
CHAPTER

7

COMPUTATIONAL LEARNING THEORY

This chapter presents a theoretical characterization of the difficulty of several types of machine learning problems and the capabilities of several types of machine learning algorithms. This theory seeks to answer questions such as “Under what conditions is successful learning possible and impossible?” and “Under what conditions is a particular learning algorithm assured of learning successfully?” Two specific frameworks for analyzing learning algorithms are considered. Within the probably approximately correct (PAC) framework, we identify classes of hypotheses that can and cannot be learned from a polynomial number of training examples and we define a natural measure of complexity for hypothesis spaces that allows bounding the number of training examples required for inductive learning. Within the mistake bound framework, we examine the number of training errors that will be made by a learner before it determines the correct hypothesis.

7.1 INTRODUCTION

When studying machine learning it is natural to wonder what general laws may govern machine (and nonmachine) learners. Is it possible to identify classes of learning problems that are inherently difficult or easy, independent of the learning algorithm? Can one characterize the number of training examples necessary or sufficient to assure successful learning? How is this number affected if the learner is allowed to pose queries to the trainer, versus observing a random sample of training examples? Can one characterize the number of mistakes that a learner

will make before learning the target function? Can one characterize the inherent computational complexity of classes of learning problems?

Although general answers to all these questions are not yet known, fragments of a computational theory of learning have begun to emerge. This chapter presents key results from this theory, providing answers to these questions within particular problem settings. We focus here on the problem of inductively learning an unknown target function, given only training examples of this target function and a space of candidate hypotheses. Within this setting, we will be chiefly concerned with questions such as how many training examples are sufficient to successfully learn the target function, and how many mistakes will the learner make before succeeding. As we shall see, it is possible to set quantitative bounds on these measures, depending on attributes of the learning problem such as:

- the size or complexity of the hypothesis space considered by the learner
- the accuracy to which the target concept must be approximated
- the probability that the learner will output a successful hypothesis
- the manner in which training examples are presented to the learner

For the most part, we will focus not on individual learning algorithms, but rather on broad classes of learning algorithms characterized by the hypothesis spaces they consider, the presentation of training examples, etc. Our goal is to answer questions such as:

- *Sample complexity.* How many training examples are needed for a learner to converge (with high probability) to a successful hypothesis?
- *Computational complexity.* How much computational effort is needed for a learner to converge (with high probability) to a successful hypothesis?
- *Mistake bound.* How many training examples will the learner misclassify before converging to a successful hypothesis?

Note there are many specific settings in which we could pursue such questions. For example, there are various ways to specify what it means for the learner to be “successful.” We might specify that to succeed, the learner must output a hypothesis identical to the target concept. Alternatively, we might simply require that it output a hypothesis that agrees with the target concept most of the time, or that it usually output such a hypothesis. Similarly, we must specify how training examples are to be obtained by the learner. We might specify that training examples are presented by a helpful teacher, or obtained by the learner performing experiments, or simply generated at random according to some process outside the learner’s control. As we might expect, the answers to the above questions depend on the particular setting, or learning model, we have in mind.

The remainder of this chapter is organized as follows. Section 7.2 introduces the probably approximately correct (PAC) learning setting. Section 7.3 then analyzes the sample complexity and computational complexity for several learning

problems within this PAC setting. Section 7.4 introduces an important measure of hypothesis space complexity called the VC-dimension and extends our PAC analysis to problems in which the hypothesis space is infinite. Section 7.5 introduces the mistake-bound model and provides a bound on the number of mistakes made by several learning algorithms discussed in earlier chapters. Finally, we introduce the WEIGHTED-MAJORITY algorithm, a practical algorithm for combining the predictions of multiple competing learning algorithms, along with a theoretical mistake bound for this algorithm.

7.2 PROBABLY LEARNING AN APPROXIMATELY CORRECT HYPOTHESIS

In this section we consider a particular setting for the learning problem, called the *probably approximately correct* (PAC) learning model. We begin by specifying the problem setting that defines the PAC learning model, then consider the questions of how many training examples and how much computation are required in order to learn various classes of target functions within this PAC model. For the sake of simplicity, we restrict the discussion to the case of learning boolean-valued concepts from noise-free training data. However, many of the results can be extended to the more general scenario of learning real-valued target functions (see, for example, Natarajan 1991), and some can be extended to learning from certain types of noisy data (see, for example, Laird 1988; Kearns and Vazirani 1994).

7.2.1 The Problem Setting

As in earlier chapters, let X refer to the set of all possible instances over which target functions may be defined. For example, X might represent the set of all people, each described by the attributes *age* (e.g., *young* or *old*) and *height* (*short* or *tall*). Let C refer to some set of target concepts that our learner might be called upon to learn. Each target concept c in C corresponds to some subset of X , or equivalently to some boolean-valued function $c : X \rightarrow \{0, 1\}$. For example, one target concept c in C might be the concept “people who are skiers.” If x is a positive example of c , then we will write $c(x) = 1$; if x is a negative example, $c(x) = 0$.

We assume instances are generated at random from X according to some probability distribution \mathcal{D} . For example, \mathcal{D} might be the distribution of instances generated by observing people who walk out of the largest sports store in Switzerland. In general, \mathcal{D} may be any distribution, and it will not generally be known to the learner. All that we require of \mathcal{D} is that it be stationary; that is, that the distribution not change over time. Training examples are generated by drawing an instance x at random according to \mathcal{D} , then presenting x along with its target value, $c(x)$, to the learner.

The learner L considers some set H of possible hypotheses when attempting to learn the target concept. For example, H might be the set of all hypotheses

describable by conjunctions of the attributes *age* and *height*. After observing a sequence of training examples of the target concept c , L must output some hypothesis h from H , which is its estimate of c . To be fair, we evaluate the success of L by the performance of h over new instances drawn randomly from X according to \mathcal{D} , the same probability distribution used to generate the training data.

Within this setting, we are interested in characterizing the performance of various learners L using various hypothesis spaces H , when learning individual target concepts drawn from various classes C . Because we demand that L be general enough to learn any target concept from C regardless of the distribution of training examples, we will often be interested in worst-case analyses over all possible target concepts from C and all possible instance distributions \mathcal{D} .

7.2.2 Error of a Hypothesis

Because we are interested in how closely the learner's output hypothesis h approximates the actual target concept c , let us begin by defining the *true error* of a hypothesis h with respect to target concept c and instance distribution \mathcal{D} . Informally, the true error of h is just the error rate we expect when applying h to future instances drawn according to the probability distribution \mathcal{D} . In fact, we already defined the true error of h in Chapter 5. For convenience, we restate the definition here using c to represent the boolean target function.

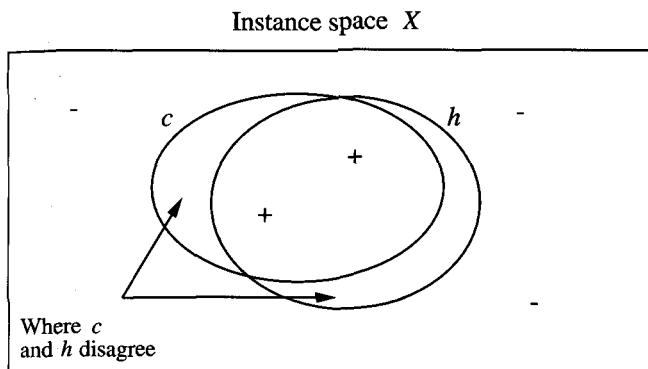
Definition: The **true error** (denoted $\text{error}_{\mathcal{D}}(h)$) of hypothesis h with respect to target concept c and distribution \mathcal{D} is the probability that h will misclassify an instance drawn at random according to \mathcal{D} .

$$\text{error}_{\mathcal{D}}(h) \equiv \Pr_{x \in \mathcal{D}} [c(x) \neq h(x)]$$

Here the notation $\Pr_{x \in \mathcal{D}}$ indicates that the probability is taken over the instance distribution \mathcal{D} .

Figure 7.1 shows this definition of error in graphical form. The concepts c and h are depicted by the sets of instances within X that they label as positive. The error of h with respect to c is the probability that a randomly drawn instance will fall into the region where h and c disagree (i.e., their set difference). Note we have chosen to define error over the *entire distribution* of instances—not simply over the training examples—because this is the true error we expect to encounter when actually using the learned hypothesis h on subsequent instances drawn from \mathcal{D} .

Note that error depends strongly on the unknown probability distribution \mathcal{D} . For example, if \mathcal{D} is a uniform probability distribution that assigns the same probability to every instance in X , then the error for the hypothesis in Figure 7.1 will be the fraction of the total instance space that falls into the region where h and c disagree. However, the same h and c will have a much higher error if \mathcal{D} happens to assign very high probability to instances for which h and c disagree. In the extreme, if \mathcal{D} happens to assign zero probability to the instances for which

**FIGURE 7.1**

The error of hypothesis h with respect to target concept c . The error of h with respect to c is the probability that a randomly drawn instance will fall into the region where h and c disagree on its classification. The + and - points indicate positive and negative training examples. Note h has a nonzero error with respect to c despite the fact that h and c agree on all five training examples observed thus far.

$h(x) = c(x)$, then the error for the h in Figure 7.1 will be 1, despite the fact the h and c agree on a very large number of (zero probability) instances.

Finally, note that the error of h with respect to c is not directly observable to the learner. L can only observe the performance of h over the *training examples*, and it must choose its output hypothesis on this basis only. We will use the term *training error* to refer to the fraction of training examples misclassified by h , in contrast to the *true error* defined above. Much of our analysis of the complexity of learning centers around the question “how probable is it that the observed *training error* for h gives a misleading estimate of the *true error* $D(h)$?”

Notice the close relationship between this question and the questions considered in Chapter 5. Recall that in Chapter 5 we defined the *sample error* of h with respect to a set S of examples to be the fraction of S misclassified by h . The training error defined above is just the sample error when S is the set of training examples. In Chapter 5 we determined the probability that the sample error will provide a misleading estimate of the true error, under the assumption that the data sample S is drawn independent of h . However, when S is the set of training data, the learned hypothesis h depends very much on S ! Therefore, in this chapter we provide an analysis that addresses this important special case.

7.2.3 PAC Learnability

Our aim is to characterize classes of target concepts that can be reliably learned from a reasonable number of randomly drawn training examples and a reasonable amount of computation.

What kinds of statements about learnability should we guess hold true? We might try to characterize the number of training examples needed to learn

a hypothesis h for which $\text{error}_{\mathcal{D}}(h) = 0$. Unfortunately, it turns out this is futile in the setting we are considering, for two reasons. First, unless we provide training examples corresponding to every possible instance in X (an unrealistic assumption), there may be multiple hypotheses consistent with the provided training examples, and the learner cannot be certain to pick the one corresponding to the target concept. Second, given that the training examples are drawn randomly, there will always be some nonzero probability that the training examples encountered by the learner will be misleading. (For example, although we might frequently see skiers of different heights, on any given day there is some small chance that all observed training examples will happen to be 2 meters tall.)

To accommodate these two difficulties, we weaken our demands on the learner in two ways. First, we will not require that the learner output a zero error hypothesis—we will require only that its error be bounded by some constant, ϵ , that can be made arbitrarily small. Second, we will not require that the learner succeed for *every* sequence of randomly drawn training examples—we will require only that its probability of failure be bounded by some constant, δ , that can be made arbitrarily small. In short, we require only that the learner *probably* learn a hypothesis that is *approximately correct*—hence the term probably approximately correct learning, or PAC learning for short.

Consider some class C of possible target concepts and a learner L using hypothesis space H . Loosely speaking, we will say that the concept class C is PAC-learnable by L using H if, for any target concept c in C , L will with probability $(1 - \delta)$ output a hypothesis h with $\text{error}_{\mathcal{D}}(h) < \epsilon$, after observing a reasonable number of training examples and performing a reasonable amount of computation. More precisely,

Definition: Consider a concept class C defined over a set of instances X of length n and a learner L using hypothesis space H . C is **PAC-learnable** by L using H if for all $c \in C$, distributions \mathcal{D} over X , ϵ such that $0 < \epsilon < 1/2$, and δ such that $0 < \delta < 1/2$, learner L will with probability at least $(1 - \delta)$ output a hypothesis $h \in H$ such that $\text{error}_{\mathcal{D}}(h) \leq \epsilon$, in time that is polynomial in $1/\epsilon$, $1/\delta$, n , and $\text{size}(c)$.

Our definition requires two things from L . First, L must, with arbitrarily high probability $(1 - \delta)$, output a hypothesis having arbitrarily low error (ϵ). Second, it must do so efficiently—in time that grows at most polynomially with $1/\epsilon$ and $1/\delta$, which define the strength of our demands on the output hypothesis, and with n and $\text{size}(c)$ that define the inherent complexity of the underlying instance space X and concept class C . Here, n is the size of instances in X . For example, if instances in X are conjunctions of k boolean features, then $n = k$. The second space parameter, $\text{size}(c)$, is the encoding length of c in C , assuming some representation for C . For example, if concepts in C are conjunctions of up to k boolean features, each described by listing the indices of the features in the conjunction, then $\text{size}(c)$ is the number of boolean features actually used to describe c .

Our definition of PAC learning may at first appear to be concerned only with the computational resources required for learning, whereas in practice we are

usually more concerned with the number of training examples required. However, the two are very closely related: If L requires some minimum processing time per training example, then for C to be PAC-learnable by L , L must learn from a polynomial number of training examples. In fact, a typical approach to showing that some class C of target concepts is PAC-learnable, is to first show that each target concept in C can be learned from a polynomial number of training examples and then show that the processing time per example is also polynomially bounded.

Before moving on, we should point out a restrictive assumption implicit in our definition of PAC-learnable. This definition implicitly assumes that the learner's hypothesis space H contains a hypothesis with arbitrarily small error for every target concept in C . This follows from the requirement in the above definition that the learner succeed when the error bound ϵ is arbitrarily close to zero. Of course this is difficult to assure if one does not know C in advance (what is C for a program that must learn to recognize faces from images?), unless H is taken to be the power set of X . As pointed out in Chapter 2, such an unbiased H will not support accurate generalization from a reasonable number of training examples. Nevertheless, the results based on the PAC learning model provide useful insights regarding the relative complexity of different learning problems and regarding the rate at which generalization accuracy improves with additional training examples. Furthermore, in Section 7.3.1 we will lift this restrictive assumption, to consider the case in which the learner makes no prior assumption about the form of the target concept.

7.3 SAMPLE COMPLEXITY FOR FINITE HYPOTHESIS SPACES

As noted above, PAC-learnability is largely determined by the number of training examples required by the learner. The growth in the number of required training examples with problem size, called the *sample complexity* of the learning problem, is the characteristic that is usually of greatest interest. The reason is that in most practical settings the factor that most limits success of the learner is the limited availability of training data.

Here we present a general bound on the sample complexity for a very broad class of learners, called *consistent learners*. A learner is *consistent* if it outputs hypotheses that perfectly fit the training data, whenever possible. It is quite reasonable to ask that a learning algorithm be consistent, given that we typically prefer a hypothesis that fits the training data over one that does not. Note that many of the learning algorithms discussed in earlier chapters, including all the learning algorithms described in Chapter 2, are consistent learners.

Can we derive a bound on the number of training examples required by *any* consistent learner, independent of the specific algorithm it uses to derive a consistent hypothesis? The answer is yes. To accomplish this, it is useful to recall the definition of version space from Chapter 2. There we defined the version space, $VS_{H,D}$, to be the set of all hypotheses $h \in H$ that correctly classify the training examples D .

$$VS_{H,D} = \{h \in H | (\forall \langle x, c(x) \rangle \in D) (h(x) = c(x))\}$$

The significance of the version space here is that *every consistent learner outputs a hypothesis belonging to the version space*, regardless of the instance space X , hypothesis space H , or training data D . The reason is simply that by definition the version space $VS_{H,D}$ contains every consistent hypothesis in H . Therefore, *to bound the number of examples needed by any consistent learner, we need only bound the number of examples needed to assure that the version space contains no unacceptable hypotheses*. The following definition, after Haussler (1988), states this condition precisely.

Definition: Consider a hypothesis space H , target concept c , instance distribution \mathcal{D} , and set of training examples D of c . The version space $VS_{H,D}$ is said to be **ϵ -exhausted** with respect to c and \mathcal{D} , if every hypothesis h in $VS_{H,D}$ has error less than ϵ with respect to c and \mathcal{D} .

$$(\forall h \in VS_{H,D}) \text{ error}_{\mathcal{D}}(h) < \epsilon$$

This definition is illustrated in Figure 7.2. The version space is ϵ -exhausted just in the case that all the hypotheses consistent with the observed training examples (i.e., those with zero training error) happen to have true error less than ϵ . Of course from the learner's viewpoint all that can be known is that these hypotheses fit the training data equally well—they all have zero training error. Only an observer who knew the identity of the target concept could determine with certainty whether the version space is ϵ -exhausted. Surprisingly, a probabilistic argument allows us to bound the probability that the version space will be ϵ -exhausted after a given number of training examples, even without knowing the identity of the target concept or the distribution from which training examples

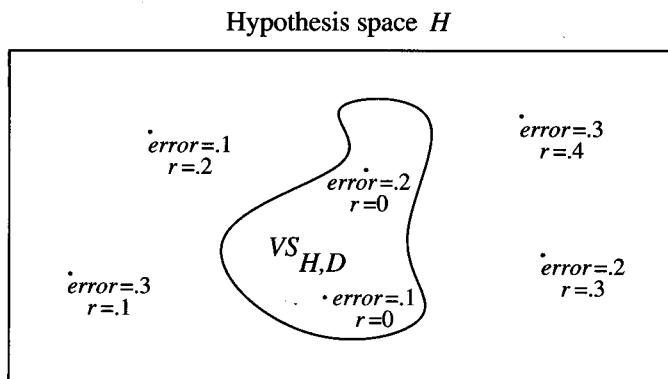


FIGURE 7.2

Exhausting the version space. The version space $VS_{H,D}$ is the subset of hypotheses $h \in H$, which have zero training error (denoted by $r = 0$ in the figure). Of course the true $\text{error}_{\mathcal{D}}(h)$ (denoted by error in the figure) may be nonzero, even for hypotheses that commit zero errors over the training data. The version space is said to be ϵ -exhausted when all hypotheses h remaining in $VS_{H,D}$ have $\text{error}_{\mathcal{D}}(h) < \epsilon$.

are drawn. Haussler (1988) provides such a bound, in the form of the following theorem.

Theorem 7.1. *ϵ -exhausting the version space.* If the hypothesis space H is finite, and D is a sequence of $m \geq 1$ independent randomly drawn examples of some target concept c , then for any $0 \leq \epsilon \leq 1$, the probability that the version space $VS_{H,D}$ is not ϵ -exhausted (with respect to c) is less than or equal to

$$|H|e^{-\epsilon m}$$

Proof. Let h_1, h_2, \dots, h_k be all the hypotheses in H that have true error greater than ϵ with respect to c . We fail to ϵ -exhaust the version space if and only if at least one of these k hypotheses happens to be consistent with all m independent random training examples. The probability that any single hypothesis having true error greater than ϵ would be consistent with one randomly drawn example is at most $(1 - \epsilon)$. Therefore the probability that this hypothesis will be consistent with m independently drawn examples is at most $(1 - \epsilon)^m$. Given that we have k hypotheses with error greater than ϵ , the probability that at least one of these will be consistent with all m training examples is at most

$$k(1 - \epsilon)^m$$

And since $k \leq |H|$, this is at most $|H|(1 - \epsilon)^m$. Finally, we use a general inequality stating that if $0 \leq \epsilon \leq 1$ then $(1 - \epsilon) \leq e^{-\epsilon}$. Thus,

$$k(1 - \epsilon)^m \leq |H|(1 - \epsilon)^m \leq |H|e^{-\epsilon m}$$

which proves the theorem. □

We have just proved an upper bound on the probability that the version space is not ϵ -exhausted, based on the number of training examples m , the allowed error ϵ , and the size of H . Put another way, this bounds the probability that m training examples will fail to eliminate all “bad” hypotheses (i.e., hypotheses with true error greater than ϵ), for any consistent learner using hypothesis space H .

Let us use this result to determine the number of training examples required to reduce this probability of failure below some desired level δ .

$$|H|e^{-\epsilon m} \leq \delta \tag{7.1}$$

Rearranging terms to solve for m , we find

$$m \geq \frac{1}{\epsilon}(\ln |H| + \ln(1/\delta)) \tag{7.2}$$

To summarize, the inequality shown in Equation (7.2) provides a general bound on the number of training examples sufficient for *any consistent learner* to successfully learn any target concept in H , for any desired values of δ and ϵ . This number m of training examples is sufficient to assure that any consistent hypothesis will be probably (with probability $(1 - \delta)$) approximately (within error ϵ) correct. Notice m grows linearly in $1/\epsilon$ and logarithmically in $1/\delta$. It also grows logarithmically in the size of the hypothesis space H .

Note that the above bound can be a substantial overestimate. For example, although the probability of failing to exhaust the version space must lie in the interval $[0, 1]$, the bound given by the theorem grows linearly with $|H|$. For sufficiently large hypothesis spaces, this bound can easily be greater than one. As a result, the bound given by the inequality in Equation (7.2) can substantially overestimate the number of training examples required. The weakness of this bound is mainly due to the $|H|$ term, which arises in the proof when summing the probability that a single hypothesis could be unacceptable, over all possible hypotheses. In fact, a much tighter bound is possible in many cases, as well as a bound that covers infinitely large hypothesis spaces. This will be the subject of Section 7.4.

7.3.1 Agnostic Learning and Inconsistent Hypotheses

Equation (7.2) is important because it tells us how many training examples suffice to ensure (with probability $(1 - \delta)$) that every hypothesis in H having zero training error will have a true error of at most ϵ . Unfortunately, if H does not contain the target concept c , then a zero-error hypothesis cannot always be found. In this case, the most we might ask of our learner is to output the hypothesis from H that has the *minimum* error over the training examples. A learner that makes no assumption that the target concept is representable by H and that simply finds the hypothesis with minimum training error, is often called an *agnostic* learner, because it makes no prior commitment about whether or not $C \subseteq H$.

Although Equation (7.2) is based on the assumption that the learner outputs a zero-error hypothesis, a similar bound can be found for this more general case in which the learner entertains hypotheses with nonzero training error. To state this precisely, let D denote the particular set of training examples available to the learner, in contrast to \mathcal{D} , which denotes the probability distribution over the entire set of instances. Let $\text{error}_D(h)$ denote the training error of hypothesis h . In particular, $\text{error}_D(h)$ is defined as the fraction of the training examples in D that are misclassified by h . Note the $\text{error}_D(h)$ over the particular sample of training data D may differ from the true error $\text{error}_{\mathcal{D}}(h)$ over the entire probability distribution \mathcal{D} . Now let h_{best} denote the hypothesis from H having lowest training error over the training examples. How many training examples suffice to ensure (with high probability) that its true error $\text{error}_{\mathcal{D}}(h_{\text{best}})$ will be no more than $\epsilon + \text{error}_D(h_{\text{best}})$? Notice the question considered in the previous section is just a special case of this question, when $\text{error}_D(h_{\text{best}})$ happens to be zero.

This question can be answered (see Exercise 7.3) using an argument analogous to the proof of Theorem 7.1. It is useful here to invoke the general Hoeffding bounds (sometimes called the additive Chernoff bounds). The Hoeffding bounds characterize the deviation between the true probability of some event and its observed frequency over m independent trials. More precisely, these bounds apply to experiments involving m distinct Bernoulli trials (e.g., m independent flips of a coin with some probability of turning up heads). This is exactly analogous to the setting we consider when estimating the error of a hypothesis in Chapter 5: The

probability of the coin being heads corresponds to the probability that the hypothesis will misclassify a randomly drawn instance. The m independent coin flips correspond to the m independently drawn instances. The frequency of heads over the m examples corresponds to the frequency of misclassifications over the m instances.

The Hoeffding bounds state that if the training error $\text{error}_D(h)$ is measured over the set D containing m randomly drawn examples, then

$$\Pr[\text{error}_D(h) > \text{error}_D(h) + \epsilon] \leq e^{-2m\epsilon^2}$$

This gives us a bound on the probability that an arbitrarily chosen single hypothesis has a very misleading training error. To assure that the *best* hypothesis found by L has an error bounded in this way, we must consider the probability that any one of the $|H|$ hypotheses could have a large error

$$\Pr[(\exists h \in H)(\text{error}_D(h) > \text{error}_D(h) + \epsilon)] \leq |H|e^{-2m\epsilon^2}$$

If we call this probability δ , and ask how many examples m suffice to hold δ to some desired value, we now obtain

$$m \geq \frac{1}{2\epsilon^2} (\ln |H| + \ln(1/\delta)) \quad (7.3)$$

This is the generalization of Equation (7.2) to the case in which the learner still picks the best hypothesis $h \in H$, but where the best hypothesis may have nonzero training error. Notice that m depends logarithmically on H and on $1/\delta$, as it did in the more restrictive case of Equation (7.2). However, in this less restrictive situation m now grows as the square of $1/\epsilon$, rather than linearly with $1/\epsilon$.

7.3.2 Conjunctions of Boolean Literals Are PAC-Learnable

Now that we have a bound indicating the number of training examples sufficient to probably approximately learn the target concept, we can use it to determine the sample complexity and PAC-learnability of some specific concept classes.

Consider the class C of target concepts described by conjunctions of boolean literals. A boolean *literal* is any boolean variable (e.g., *Old*), or its negation (e.g., $\neg \text{Old}$). Thus, conjunctions of boolean literals include target concepts such as "*Old* \wedge $\neg \text{Tall}$ ". Is C PAC-learnable? We can show that the answer is yes by first showing that any consistent learner will require only a polynomial number of training examples to learn any c in C , and then suggesting a specific algorithm that uses polynomial time per training example.

Consider any consistent learner L using a hypothesis space H identical to C . We can use Equation (7.2) to compute the number m of random training examples sufficient to ensure that L will, with probability $(1 - \delta)$, output a hypothesis with maximum error ϵ . To accomplish this, we need only determine the size $|H|$ of the hypothesis space.

Now consider the hypothesis space H defined by conjunctions of literals based on n boolean variables. The size $|H|$ of this hypothesis space is 3^n . To see this, consider the fact that there are only three possibilities for each variable in

any given hypothesis: Include the variable as a literal in the hypothesis, include its negation as a literal, or ignore it. Given n such variables, there are 3^n distinct hypotheses.

Substituting $|H| = 3^n$ into Equation (7.2) gives the following bound for the sample complexity of learning conjunctions of up to n boolean literals.

$$m \geq \frac{1}{\epsilon} (n \ln 3 + \ln(1/\delta)) \quad (7.4)$$

For example, if a consistent learner attempts to learn a target concept described by conjunctions of up to 10 boolean literals, and we desire a 95% probability that it will learn a hypothesis with error less than .1, then it suffices to present m randomly drawn training examples, where $m = \frac{1}{.1} (10 \ln 3 + \ln(1/.05)) = 140$.

Notice that m grows linearly in the number of literals n , linearly in $1/\epsilon$, and logarithmically in $1/\delta$. What about the overall computational effort? That will depend, of course, on the specific learning algorithm. However, as long as our learning algorithm requires no more than polynomial computation per training example, and no more than a polynomial number of training examples, then the total computation required will be polynomial as well.

In the case of learning conjunctions of boolean literals, one algorithm that meets this requirement has already been presented in Chapter 2. It is the FIND-S algorithm, which incrementally computes the most specific hypothesis consistent with the training examples. For each new positive training example, this algorithm computes the intersection of the literals shared by the current hypothesis and the new training example, using time linear in n . Therefore, the FIND-S algorithm PAC-learns the concept class of conjunctions of n boolean literals with negations.

Theorem 7.2. PAC-learnability of boolean conjunctions. The class C of conjunctions of boolean literals is PAC-learnable by the FIND-S algorithm using $H = C$.

Proof. Equation (7.4) shows that the sample complexity for this concept class is polynomial in n , $1/\delta$, and $1/\epsilon$, and independent of $\text{size}(c)$. To incrementally process each training example, the FIND-S algorithm requires effort linear in n and independent of $1/\delta$, $1/\epsilon$, and $\text{size}(c)$. Therefore, this concept class is PAC-learnable by the FIND-S algorithm. \square

7.3.3 PAC-Learnability of Other Concept Classes

As we just saw, Equation (7.2) provides a general basis for bounding the sample complexity for learning target concepts in some given class C . Above we applied it to the class of conjunctions of boolean literals. It can also be used to show that many other concept classes have polynomial sample complexity (e.g., see Exercise 7.2).

7.3.3.1 UNBIASED LEARNERS

Not all concept classes have polynomially bounded sample complexity according to the bound of Equation (7.2). For example, consider the *unbiased* concept class

C that contains every teachable concept relative to X . The set C of all definable target concepts corresponds to the power set of X —the set of all subsets of X —which contains $|C| = 2^{|X|}$ concepts. Suppose that instances in X are defined by n boolean features. In this case, there will be $|X| = 2^n$ distinct instances, and therefore $|C| = 2^{|X|} = 2^{2^n}$ distinct concepts. Of course to learn such an unbiased concept class, the learner must itself use an unbiased hypothesis space $H = C$. Substituting $|H| = 2^{2^n}$ into Equation (7.2) gives the sample complexity for learning the unbiased concept class relative to X .

$$m \geq \frac{1}{\epsilon} (2^n \ln 2 + \ln(1/\delta)) \quad (7.5)$$

Thus, this unbiased class of target concepts has exponential sample complexity under the PAC model, according to Equation (7.2). Although Equations (7.2) and (7.5) are not tight upper bounds, it can in fact be proven that the sample complexity for the unbiased concept class is exponential in n .

7.3.3.2 K -TERM DNF AND K -CNF CONCEPTS

It is also possible to find concept classes that have polynomial sample complexity, but nevertheless cannot be learned in polynomial time. One interesting example is the concept class C of k -term disjunctive normal form (k -term DNF) expressions. k -term DNF expressions are of the form $T_1 \vee T_2 \vee \dots \vee T_k$, where each term T_i is a conjunction of n boolean attributes and their negations. Assuming $H = C$, it is easy to show that $|H|$ is at most 3^{nk} (because there are k terms, each of which may take on 3^n possible values). Note 3^{nk} is an overestimate of H , because it is double counting the cases where $T_i = T_j$ and where T_i is *more general than* T_j . Still, we can use this upper bound on $|H|$ to obtain an upper bound on the sample complexity, substituting this into Equation (7.2).

$$m \geq \frac{1}{\epsilon} (nk \ln 3 + \ln(1/\delta)) \quad (7.6)$$

which indicates that the sample complexity of k -term DNF is polynomial in $1/\epsilon$, $1/\delta$, n , and k . Despite having polynomial sample complexity, the computational complexity is not polynomial, because this learning problem can be shown to be equivalent to other problems that are known to be unsolvable in polynomial time (unless $RP = NP$). Thus, although k -term DNF has polynomial sample complexity, it does not have polynomial computational complexity for a learner using $H = C$.

The surprising fact about k -term DNF is that although it is not PAC-learnable, there is a strictly larger concept class that is! This is possible because the larger concept class has polynomial computation complexity per example and still has polynomial sample complexity. This larger class is the class of k -CNF expressions: conjunctions of arbitrary length of the form $T_1 \wedge T_2 \wedge \dots \wedge T_j$, where each T_i is a disjunction of up to k boolean attributes. It is straightforward to show that k -CNF subsumes k -DNF, because any k -term DNF expression can easily be

rewritten as a k -CNF expression (but not vice versa). Although k -CNF is more expressive than k -term DNF, it has both polynomial sample complexity and polynomial time complexity. Hence, the concept class k -term DNF is PAC learnable by an efficient algorithm using $H = k$ -CNF. See Kearns and Vazirani (1994) for a more detailed discussion.

7.4 SAMPLE COMPLEXITY FOR INFINITE HYPOTHESIS SPACES

In the above section we showed that sample complexity for PAC learning grows as the logarithm of the size of the hypothesis space. While Equation (7.2) is quite useful, there are two drawbacks to characterizing sample complexity in terms of $|H|$. First, it can lead to quite weak bounds (recall that the bound on δ can be significantly greater than 1 for large $|H|$). Second, in the case of infinite hypothesis spaces we cannot apply Equation (7.2) at all!

Here we consider a second measure of the complexity of H , called the Vapnik-Chervonenkis dimension of H (VC dimension, or $VC(H)$, for short). As we shall see, we can state bounds on sample complexity that use $VC(H)$ rather than $|H|$. In many cases, the sample complexity bounds based on $VC(H)$ will be tighter than those from Equation (7.2). In addition, these bounds allow us to characterize the sample complexity of many infinite hypothesis spaces, and can be shown to be fairly tight.

7.4.1 Shattering a Set of Instances

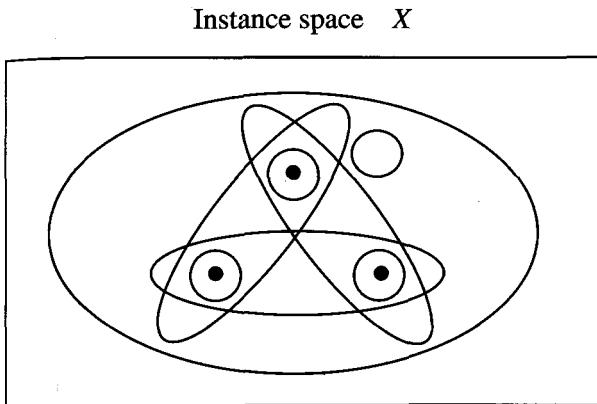
The VC dimension measures the complexity of the hypothesis space H , not by the number of distinct hypotheses $|H|$, but instead by the number of distinct instances from X that can be completely discriminated using H .

To make this notion more precise, we first define the notion of *shattering* a set of instances. Consider some subset of instances $S \subseteq X$. For example, Figure 7.3 shows a subset of three instances from X . Each hypothesis h from H imposes some dichotomy on S ; that is, h partitions S into the two subsets $\{x \in S | h(x) = 1\}$ and $\{x \in S | h(x) = 0\}$. Given some instance set S , there are $2^{|S|}$ possible dichotomies, though H may be unable to represent some of these. We say that H shatters S if every possible dichotomy of S can be represented by some hypothesis from H .

Definition: A set of instances S is **shattered** by hypothesis space H if and only if for every dichotomy of S there exists some hypothesis in H consistent with this dichotomy.

Figure 7.3 illustrates a set S of three instances that is shattered by the hypothesis space. Notice that each of the 2^3 dichotomies of these three instances is covered by some hypothesis.

Note that if a set of instances is not shattered by a hypothesis space, then there must be some concept (dichotomy) that can be defined over the instances, but that cannot be represented by the hypothesis space. The ability of H to shatter

**FIGURE 7.3**

A set of three instances shattered by eight hypotheses. For every possible dichotomy of the instances, there exists a corresponding hypothesis.

a set of instances is thus a measure of its capacity to represent target concepts defined over these instances.

7.4.2 The Vapnik-Chervonenkis Dimension

The ability to shatter a set of instances is closely related to the inductive bias of a hypothesis space. Recall from Chapter 2 that an unbiased hypothesis space is one capable of representing every possible concept (dichotomy) definable over the instance space X . Put briefly, an unbiased hypothesis space H is one that shatters the instance space X . What if H cannot shatter X , but can shatter some large subset S of X ? Intuitively, it seems reasonable to say that the larger the subset of X that can be shattered, the more expressive H . The VC dimension of H is precisely this measure.

Definition: The **Vapnik-Chervonenkis dimension**, $VC(H)$, of hypothesis space H defined over instance space X is the size of the largest finite subset of X shattered by H . If arbitrarily large finite sets of X can be shattered by H , then $VC(H) \equiv \infty$.

Note that for any finite H , $VC(H) \leq \log_2 |H|$. To see this, suppose that $VC(H) = d$. Then H will require 2^d distinct hypotheses to shatter d instances. Hence, $2^d \leq |H|$, and $d = VC(H) \leq \log_2 |H|$.

7.4.2.1 ILLUSTRATIVE EXAMPLES

In order to develop an intuitive feeling for $VC(H)$, consider a few example hypothesis spaces. To get started, suppose the instance space X is the set of real numbers $X = \mathbb{R}$ (e.g., describing the *height* of people), and H the set of intervals on the real number line. In other words, H is the set of hypotheses of the

form $a < x < b$, where a and b may be any real constants. What is $VC(H)$? To answer this question, we must find the largest subset of X that can be shattered by H . Consider a particular subset containing two distinct instances, say $S = \{3.1, 5.7\}$. Can S be shattered by H ? Yes. For example, the four hypotheses $(1 < x < 2)$, $(1 < x < 4)$, $(4 < x < 7)$, and $(1 < x < 7)$ will do. Together, they represent each of the four dichotomies over S , covering neither instance, either one of the instances, and both of the instances, respectively. Since we have found a set of size two that can be shattered by H , we know the VC dimension of H is at least two. Is there a set of size three that can be shattered? Consider a set $S = \{x_0, x_1, x_2\}$ containing three arbitrary instances. Without loss of generality, assume $x_0 < x_1 < x_2$. Clearly this set cannot be shattered, because the dichotomy that includes x_0 and x_2 , but not x_1 , cannot be represented by a single closed interval. Therefore, no subset S of size three can be shattered, and $VC(H) = 2$. Note here that H is infinite, but $VC(H)$ finite.

Next consider the set X of instances corresponding to points on the x, y plane (see Figure 7.4). Let H be the set of all linear decision surfaces in the plane. In other words, H is the hypothesis space corresponding to a single perceptron unit with two inputs (see Chapter 4 for a general discussion of perceptrons). What is the VC dimension of this H ? It is easy to see that any two distinct points in the plane can be shattered by H , because we can find four linear surfaces that include neither, either, or both points. What about sets of three points? As long as the points are not colinear, we will be able to find 2^3 linear surfaces that shatter them. Of course three colinear points cannot be shattered (for the same reason that the three points on the real line could not be shattered in the previous example). What is $VC(H)$ in this case—two or three? It is at least three. The definition of VC dimension indicates that if we find *any* set of instances of size d that can be shattered, then $VC(H) \geq d$. To show that $VC(H) < d$, we must show that no set of size d can be shattered. In this example, no sets of size four can be shattered, so $VC(H) = 3$. More generally, it can be shown that the VC dimension of linear decision surfaces in an r dimensional space (i.e., the VC dimension of a perceptron with r inputs) is $r + 1$.

As one final example, suppose each instance in X is described by the conjunction of exactly three boolean literals, and suppose that each hypothesis in H is described by the conjunction of up to three boolean literals. What is $VC(H)$? We

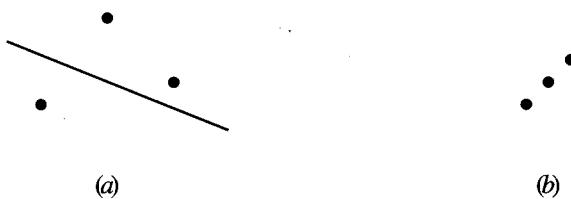


FIGURE 7.4

The VC dimension for linear decision surfaces in the x, y plane is 3. (a) A set of three points that can be shattered using linear decision surfaces. (b) A set of three that cannot be shattered.

can show that it is at least 3, as follows. Represent each instance by a 3-bit string corresponding to the values of each of its three literals l_1 , l_2 , and l_3 . Consider the following set of three instances:

$instance_1: 100$

$instance_2: 010$

$instance_3: 001$

This set of three instances can be shattered by H , because a hypothesis can be constructed for any desired dichotomy as follows: If the dichotomy is to exclude $instance_1$, add the literal $\neg l_1$ to the hypothesis. For example, suppose we wish to include $instance_2$, but exclude $instance_1$ and $instance_3$. Then we use the hypothesis $\neg l_1 \wedge \neg l_3$. This argument easily extends from three features to n . Thus, the VC dimension for conjunctions of n boolean literals is at least n . In fact, it is exactly n , though showing this is more difficult, because it requires demonstrating that no set of $n + 1$ instances can be shattered.

7.4.3 Sample Complexity and the VC Dimension

Earlier we considered the question “How many randomly drawn training examples suffice to probably approximately learn any target concept in C ? ” (i.e., how many examples suffice to ϵ -exhaust the version space with probability $(1 - \delta)$?). Using $VC(H)$ as a measure for the complexity of H , it is possible to derive an alternative answer to this question, analogous to the earlier bound of Equation (7.2). This new bound (see Blumer et al. 1989) is

$$m \geq \frac{1}{\epsilon} (4 \log_2(2/\delta) + 8VC(H) \log_2(13/\epsilon)) \quad (7.7)$$

Note that just as in the bound from Equation (7.2), the number of required training examples m grows logarithmically in $1/\delta$. It now grows log times linear in $1/\epsilon$, rather than linearly. Significantly, the $\ln |H|$ term in the earlier bound has now been replaced by the alternative measure of hypothesis space complexity, $VC(H)$ (recall $VC(H) \leq \log_2 |H|$).

Equation (7.7) provides an upper bound on the number of training examples sufficient to probably approximately learn any target concept in C , for any desired ϵ and δ . It is also possible to obtain a lower bound, as summarized in the following theorem (see Ehrenfeucht et al. 1989).

Theorem 7.3. Lower bound on sample complexity. Consider any concept class C such that $VC(C) \geq 2$, any learner L , and any $0 < \epsilon < \frac{1}{8}$, and $0 < \delta < \frac{1}{100}$. Then there exists a distribution \mathcal{D} and target concept in C such that if L observes fewer examples than

$$\max \left[\frac{1}{\epsilon} \log(1/\delta), \frac{VC(C) - 1}{32\epsilon} \right]$$

then with probability at least δ , L outputs a hypothesis h having $error_{\mathcal{D}}(h) > \epsilon$.

This theorem states that if the number of training examples is too few, then no learner can PAC-learn every target concept in any nontrivial C . Thus, this theorem provides a lower bound on the number of training examples *necessary* for successful learning, complementing the earlier upper bound that gives a *sufficient* number. Notice this lower bound is determined by the complexity of the concept class C , whereas our earlier upper bounds were determined by H . (Why?)[†]

This lower bound shows that the upper bound of the inequality in Equation (7.7) is fairly tight. Both bounds are logarithmic in $1/\delta$ and linear in $VC(H)$. The only difference in the order of these two bounds is the extra $\log(1/\epsilon)$ dependence in the upper bound.

7.4.4 VC Dimension for Neural Networks

Given the discussion of artificial neural network learning in Chapter 4, it is interesting to consider how we might calculate the VC dimension of a network of interconnected units such as the feedforward networks trained by the BACKPROPAGATION procedure. This section presents a general result that allows computing the VC dimension of layered acyclic networks, based on the structure of the network and the VC dimension of its individual units. This VC dimension can then be used to bound the number of training examples sufficient to probably approximately correctly learn a feedforward network to desired values of ϵ and δ . This section may be skipped on a first reading without loss of continuity.

Consider a network, G , of units, which forms a layered directed acyclic graph. A *directed acyclic* graph is one for which the edges have a direction (e.g., the units have inputs and outputs), and in which there are no directed cycles. A *layered* graph is one whose nodes can be partitioned into layers such that all directed edges from nodes at layer l go to nodes at layer $l + 1$. The layered feedforward neural networks discussed throughout Chapter 4 are examples of such layered directed acyclic graphs.

It turns out that we can bound the VC dimension of such networks based on their graph structure and the VC dimension of the primitive units from which they are constructed. To formalize this, we must first define a few more terms. Let n be the number of inputs to the network G , and let us assume that there is just one output node. Let each internal unit N_i of G (i.e., each node that is not an input) have at most r inputs and implement a boolean-valued function $c_i : \mathcal{W} \rightarrow \{0, 1\}$ from some function class C . For example, if the internal nodes are perceptrons, then C will be the class of linear threshold functions defined over \mathcal{W} .

We can now define the *G -composition* of C to be the class of all functions that can be implemented by the network G assuming individual units in G take on functions from the class C . In brief, the G -composition of C is the hypothesis space representable by the network G .

[†]Hint: If we were to substitute H for C in the lower bound, this would result in a tighter bound on m in the case $H \supset C$.

The following theorem bounds the VC dimension of the G -composition of C , based on the VC dimension of C and the structure of G .

Theorem 7.4. VC-dimension of directed acyclic layered networks. (See Kearns and Vazirani 1994.) Let G be a layered directed acyclic graph with n input nodes and $s \geq 2$ internal nodes, each having at most r inputs. Let C be a concept class over \Re^r of VC dimension d , corresponding to the set of functions that can be described by each of the s internal nodes. Let C_G be the G -composition of C , corresponding to the set of functions that can be represented by G . Then $VC(C_G) \leq 2ds \log(es)$, where e is the base of the natural logarithm.

Note this bound on the VC dimension of the network G grows linearly with the VC dimension d of its individual units and log times linear in s , the number of threshold units in the network.

Suppose we consider acyclic layered networks whose individual nodes are perceptrons. Recall from Chapter 4 that an r input perceptron uses linear decision surfaces to represent boolean functions over \Re^r . As noted in Section 7.4.2.1, the VC dimension of linear decision surfaces over \Re^r is $r + 1$. Therefore, a single perceptron with r inputs has VC dimension $r + 1$. We can use this fact, together with the above theorem, to bound the VC dimension of acyclic layered networks containing s perceptrons, each with r inputs, as

$$VC(C_G^{perceptrons}) \leq 2(r + 1)s \log(es)$$

We can now bound the number m of training examples sufficient to learn (with probability at least $(1 - \delta)$) any target concept from $C_G^{perceptrons}$ to within error ϵ . Substituting the above expression for the network VC dimension into Equation (7.7), we have

$$\begin{aligned} m &\geq \frac{1}{\epsilon}(4 \log(2/\delta) + 8VC(H) \log(13/\epsilon)) \\ &\geq \frac{1}{\epsilon}(4 \log(2/\delta) + 16(r + 1)s \log(es) \log(13/\epsilon)) \end{aligned} \quad (7.8)$$

As illustrated by this perceptron network example, the above theorem is interesting because it provides a general method for bounding the VC dimension of layered, acyclic networks of units, based on the network structure and the VC dimension of the individual units. Unfortunately the above result does not directly apply to networks trained using BACKPROPAGATION, for two reasons. First, this result applies to networks of perceptrons rather than networks of *sigmoid units* to which the BACKPROPAGATION algorithm applies. Nevertheless, notice that the VC dimension of sigmoid units will be at least as great as that of perceptrons, because a sigmoid unit can approximate a perceptron to arbitrary accuracy by using sufficiently large weights. Therefore, the above bound on m will be at least as large for acyclic layered networks of sigmoid units. The second shortcoming of the above result is that it fails to account for the fact that BACKPROPAGATION

trains a network by beginning with near-zero weights, then iteratively modifying these weights until an acceptable hypothesis is found. Thus, BACKPROPAGATION with a cross-validation stopping criterion exhibits an inductive bias in favor of networks with small weights. This inductive bias, which reduces the effective VC dimension, is not captured by the above analysis.

7.5 THE MISTAKE BOUND MODEL OF LEARNING

While we have focused thus far on the PAC learning model, computational learning theory considers a variety of different settings and questions. Different learning settings that have been studied vary by how the training examples are generated (e.g., passive observation of random examples, active querying by the learner), noise in the data (e.g., noisy or error-free), the definition of success (e.g., the target concept must be learned exactly, or only probably and approximately), assumptions made by the learner (e.g., regarding the distribution of instances and whether $C \subseteq H$), and the measure according to which the learner is evaluated (e.g., number of training examples, number of mistakes, total time).

In this section we consider the *mistake bound* model of learning, in which the learner is evaluated by the total number of mistakes it makes before it converges to the correct hypothesis. As in the PAC setting, we assume the learner receives a sequence of training examples. However, here we demand that upon receiving each example x , the learner must predict the target value $c(x)$, before it is shown the correct target value by the trainer. The question considered is “How many *mistakes* will the learner make in its predictions before it learns the target concept?” This question is significant in practical settings where learning must be done while the system is in actual use, rather than during some off-line training stage. For example, if the system is to learn to predict which credit card purchases should be approved and which are fraudulent, based on data collected during use, then we are interested in minimizing the total number of mistakes it will make before converging to the correct target function. Here the total number of mistakes can be even more important than the total number of training examples.

This mistake bound learning problem may be studied in various specific settings. For example, we might count the number of mistakes made before PAC learning the target concept. In the examples below, we consider instead the number of mistakes made before learning the target concept *exactly*. Learning the target concept exactly means converging to a hypothesis such that $(\forall x)h(x) = c(x)$.

7.5.1 Mistake Bound for the FIND-S Algorithm

To illustrate, consider again the hypothesis space H consisting of conjunctions of up to n boolean literals $l_1 \dots l_n$ and their negations (e.g., $\text{Rich} \wedge \neg\text{Handsome}$). Recall the FIND-S algorithm from Chapter 2, which incrementally computes the maximally specific hypothesis consistent with the training examples. A straightforward implementation of FIND-S for the hypothesis space H is as follows:

FIND-S:

- Initialize h to the most specific hypothesis $l_1 \wedge \neg l_1 \wedge l_2 \wedge \neg l_2 \dots l_n \wedge \neg l_n$
- For each positive training instance x
 - Remove from h any literal that is not satisfied by x
- Output hypothesis h .

FIND-S converges in the limit to a hypothesis that makes no errors, provided $C \subseteq H$ and provided the training data is noise-free. FIND-S begins with the most specific hypothesis (which classifies every instance a negative example), then incrementally generalizes this hypothesis as needed to cover observed positive training examples. For the hypothesis representation used here, this generalization step consists of deleting unsatisfied literals.

Can we prove a bound on the total number of mistakes that FIND-S will make before exactly learning the target concept c ? The answer is yes. To see this, note first that if $c \in H$, then FIND-S can never mistakenly classify a negative example as positive. The reason is that its current hypothesis h is always at least as specific as the target concept c . Therefore, to calculate the number of mistakes it will make, we need only count the number of mistakes it will make misclassifying truly positive examples as negative. How many such mistakes can occur before FIND-S learns c exactly? Consider the first positive example encountered by FIND-S. The learner will certainly make a mistake classifying this example, because its initial hypothesis labels every instance negative. However, the result will be that half of the $2n$ terms in its initial hypothesis will be eliminated, leaving only n terms. For each subsequent positive example that is mistakenly classified by the current hypothesis, at least one more of the remaining n terms must be eliminated from the hypothesis. Therefore, the total number of mistakes can be at most $n + 1$. This number of mistakes will be required in the worst case, corresponding to learning the most general possible target concept $(\forall x)c(x) = 1$ and corresponding to a worst case sequence of instances that removes only one literal per mistake.

7.5.2 Mistake Bound for the HALVING Algorithm

As a second example, consider an algorithm that learns by maintaining a description of the version space, incrementally refining the version space as each new training example is encountered. The CANDIDATE-ELIMINATION algorithm and the LIST-THEN-ELIMINATE algorithm from Chapter 2 are examples of such algorithms. In this section we derive a worst-case bound on the number of mistakes that will be made by such a learner, for any finite hypothesis space H , assuming again that the target concept must be learned exactly.

To analyze the number of mistakes made while learning we must first specify precisely how the learner will make predictions given a new instance x . Let us assume this prediction is made by taking a majority vote among the hypotheses in the current version space. If the majority of version space hypotheses classify the new instance as positive, then this prediction is output by the learner. Otherwise a negative prediction is output.

This combination of learning the version space, together with using a majority vote to make subsequent predictions, is often called the HALVING algorithm. What is the maximum number of mistakes that can be made by the HALVING algorithm, for an arbitrary finite H , before it exactly learns the target concept? Notice that learning the target concept “exactly” corresponds to reaching a state where the version space contains only a single hypothesis (as usual, we assume the target concept c is in H).

To derive the mistake bound, note that the only time the HALVING algorithm can make a mistake is when the majority of hypotheses in its current version space incorrectly classify the new example. In this case, once the correct classification is revealed to the learner, the version space will be reduced to at most half its current size (i.e., only those hypotheses that voted with the minority will be retained). Given that each mistake reduces the size of the version space by at least half, and given that the initial version space contains only $|H|$ members, the maximum number of mistakes possible before the version space contains just one member is $\log_2 |H|$. In fact one can show the bound is $\lfloor \log_2 |H| \rfloor$. Consider, for example, the case in which $|H| = 7$. The first mistake must reduce $|H|$ to at most 3, and the second mistake will then reduce it to 1.

Note that $\lfloor \log_2 |H| \rfloor$ is a worst-case bound, and that it is possible for the HALVING algorithm to learn the target concept exactly without making any mistakes at all! This can occur because even when the majority vote is correct, the algorithm will remove the incorrect, minority hypotheses. If this occurs over the entire training sequence, then the version space may be reduced to a single member while making no mistakes along the way.

One interesting extension to the HALVING algorithm is to allow the hypotheses to vote with different weights. Chapter 6 describes the Bayes optimal classifier, which takes such a weighted vote among hypotheses. In the Bayes optimal classifier, the weight assigned to each hypothesis is the estimated posterior probability that it describes the target concept, given the training data. Later in this section we describe a different algorithm based on weighted voting, called the WEIGHTED-MAJORITY algorithm.

7.5.3 Optimal Mistake Bounds

The above analyses give worst-case mistake bounds for two specific algorithms: FIND-S and CANDIDATE-ELIMINATION. It is interesting to ask what is the optimal mistake bound for an arbitrary concept class C , assuming $H = C$. By optimal mistake bound we mean the lowest worst-case mistake bound over all possible learning algorithms. To be more precise, for any learning algorithm A and any target concept c , let $M_A(c)$ denote the maximum over all possible sequences of training examples of the number of mistakes made by A to exactly learn c . Now for any nonempty concept class C , let $M_A(C) \equiv \max_{c \in C} M_A(c)$. Note that above we showed $M_{\text{Find-S}}(C) = n + 1$ when C is the concept class described by up to n boolean literals. We also showed $M_{\text{Halving}}(C) \leq \log_2(|C|)$ for any concept class C .

We define the optimal mistake bound for a concept class C below.

Definition: Let C be an arbitrary nonempty concept class. The **optimal mistake bound** for C , denoted $Opt(C)$, is the minimum over all possible learning algorithms A of $M_A(C)$.

$$Opt(C) \equiv \min_{A \text{ learning algorithms}} M_A(C)$$

Speaking informally, this definition states that $Opt(C)$ is the number of mistakes made for the hardest target concept in C , using the hardest training sequence, by the best algorithm. Littlestone (1987) shows that for any concept class C , there is an interesting relationship among the optimal mistake bound for C , the bound of the HALVING algorithm, and the VC dimension of C , namely

$$VC(C) \leq Opt(C) \leq M_{\text{Halving}}(C) \leq \log_2(|C|)$$

Furthermore, there exist concept classes for which the four quantities above are exactly equal. One such concept class is the powerset C_P of any finite set of instances X . In this case, $VC(C_P) = |X| = \log_2(|C_P|)$, so all four quantities must be equal. Littlestone (1987) provides examples of other concept classes for which $VC(C)$ is strictly less than $Opt(C)$ and for which $Opt(C)$ is strictly less than $M_{\text{Halving}}(C)$.

7.5.4 WEIGHTED-MAJORITY Algorithm

In this section we consider a generalization of the HALVING algorithm called the WEIGHTED-MAJORITY algorithm. The WEIGHTED-MAJORITY algorithm makes predictions by taking a weighted vote among a pool of prediction algorithms and learns by altering the weight associated with each prediction algorithm. These prediction algorithms can be taken to be the alternative hypotheses in H , or they can be taken to be alternative learning algorithms that themselves vary over time. All that we require of a prediction algorithm is that it predict the value of the target concept, given an instance. One interesting property of the WEIGHTED-MAJORITY algorithm is that it is able to accommodate inconsistent training data. This is because it does not eliminate a hypothesis that is found to be inconsistent with some training example, but rather reduces its weight. A second interesting property is that we can bound the number of mistakes made by WEIGHTED-MAJORITY in terms of the number of mistakes committed by the best of the pool of prediction algorithms.

The WEIGHTED-MAJORITY algorithm begins by assigning a weight of 1 to each prediction algorithm, then considers the training examples. Whenever a prediction algorithm misclassifies a new training example its weight is decreased by multiplying it by some number β , where $0 < \beta < 1$. The exact definition of the WEIGHTED-MAJORITY algorithm is given in Table 7.1.

Notice if $\beta = 0$ then WEIGHTED-MAJORITY is identical to the HALVING algorithm. On the other hand, if we choose some other value for β , no prediction

a_i denotes the i^{th} prediction algorithm in the pool A of algorithms. w_i denotes the weight associated with a_i .

- For all i initialize $w_i \leftarrow 1$
- For each training example $(x, c(x))$
 - Initialize q_0 and q_1 to 0
 - For each prediction algorithm a_i
 - If $a_i(x) = 0$ then $q_0 \leftarrow q_0 + w_i$
 - If $a_i(x) = 1$ then $q_1 \leftarrow q_1 + w_i$
 - If $q_1 > q_0$ then predict $c(x) = 1$
 - If $q_0 > q_1$ then predict $c(x) = 0$
 - If $q_1 = q_0$ then predict 0 or 1 at random for $c(x)$
- For each prediction algorithm a_i in A do
 - If $a_i(x) \neq c(x)$ then $w_i \leftarrow \beta w_i$

TABLE 7.1

WEIGHTED-MAJORITY algorithm.

algorithm will ever be eliminated completely. If an algorithm misclassifies a training example, it will simply receive a smaller vote in the future.

We now show that the number of mistakes committed by the WEIGHTED-MAJORITY algorithm can be bounded in terms of the number of mistakes made by the best prediction algorithm in the voting pool.

Theorem 7.5. Relative mistake bound for WEIGHTED-MAJORITY. Let D be any sequence of training examples, let A be any set of n prediction algorithms, and let k be the minimum number of mistakes made by any algorithm in A for the training sequence D . Then the number of mistakes over D made by the WEIGHTED-MAJORITY algorithm using $\beta = \frac{1}{2}$ is at most

$$2.4(k + \log_2 n)$$

Proof. We prove the theorem by comparing the final weight of the best prediction algorithm to the sum of weights over all algorithms. Let a_j denote an algorithm from A that commits the optimal number k of mistakes. The final weight w_j associated with a_j will be $(\frac{1}{2})^k$, because its initial weight is 1 and it is multiplied by $\frac{1}{2}$ for each mistake. Now consider the sum $W = \sum_{i=1}^n w_i$ of the weights associated with all n algorithms in A . W is initially n . For each mistake made by WEIGHTED-MAJORITY, W is reduced to at most $\frac{3}{4}W$. This is the case because the algorithms voting in the weighted majority must hold at least half of the total weight W , and this portion of W will be reduced by a factor of $\frac{1}{2}$. Let M denote the total number of mistakes committed by WEIGHTED-MAJORITY for the training sequence D . Then the final total weight W is at most $n(\frac{3}{4})^M$. Because the final weight w_j cannot be greater than the final total weight, we have

$$\left(\frac{1}{2}\right)^k \leq n \left(\frac{3}{4}\right)^M$$

Rearranging terms yields

$$M \leq \frac{(k + \log_2 n)}{-\log_2 \left(\frac{3}{4}\right)} \leq 2.4(k + \log_2 n)$$

which proves the theorem. \square

To summarize, the above theorem states that the number of mistakes made by the WEIGHTED-MAJORITY algorithm will never be greater than a constant factor times the number of mistakes made by the best member of the pool, plus a term that grows only logarithmically in the size of the pool.

This theorem is generalized by Littlestone and Warmuth (1991), who show that for an arbitrary $0 \leq \beta < 1$ the above bound is

$$\frac{k \log_2 \frac{1}{\beta} + \log_2 n}{\log_2 \frac{2}{1+\beta}}$$

7.6 SUMMARY AND FURTHER READING

The main points of this chapter include:

- The probably approximately correct (PAC) model considers algorithms that learn target concepts from some concept class C , using training examples drawn at random according to an unknown, but fixed, probability distribution. It requires that the learner probably (with probability at least $[1 - \delta]$) learn a hypothesis that is approximately (within error ϵ) correct, given computational effort and training examples that grow only polynomially with $1/\epsilon$, $1/\delta$, the size of the instances, and the size of the target concept.
- Within the setting of the PAC learning model, any consistent learner using a finite hypothesis space H where $C \subseteq H$ will, with probability $(1 - \delta)$, output a hypothesis within error ϵ of the target concept, after observing m randomly drawn training examples, as long as

$$m \geq \frac{1}{\epsilon} (\ln(1/\delta) + \ln |H|)$$

This gives a bound on the number of training examples sufficient for successful learning under the PAC model.

- One constraining assumption of the PAC learning model is that the learner knows in advance some restricted concept class C that contains the target concept to be learned. In contrast, the *agnostic learning* model considers the more general setting in which the learner makes no assumption about the class from which the target concept is drawn. Instead, the learner outputs the hypothesis from H that has the least error (possibly nonzero) over the training data. Under this less restrictive agnostic learning model, the learner is assured with probability $(1 - \delta)$ to output a hypothesis within error ϵ of the

best possible hypothesis in H , after observing m randomly drawn training examples, provided

$$m \geq \frac{1}{2\epsilon^2} (\ln(1/\delta) + \ln |H|)$$

- The number of training examples required for successful learning is strongly influenced by the complexity of the hypothesis space considered by the learner. One useful measure of the complexity of a hypothesis space H is its Vapnik-Chervonenkis dimension, $VC(H)$. $VC(H)$ is the size of the largest subset of instances that can be shattered (split in all possible ways) by H .
- An alternative upper bound on the number of training examples sufficient for successful learning under the PAC model, stated in terms of $VC(H)$ is

$$m \geq \frac{1}{\epsilon} (4 \log_2(2/\delta) + 8VC(H) \log_2(13/\epsilon))$$

A lower bound is

$$m \geq \max \left[\frac{1}{\epsilon} \log(1/\delta), \frac{VC(C) - 1}{32\epsilon} \right]$$

- An alternative learning model, called the *mistake bound model*, is used to analyze the number of training examples a learner will misclassify before it exactly learns the target concept. For example, the HALVING algorithm will make at most $\lfloor \log_2 |H| \rfloor$ mistakes before exactly learning any target concept drawn from H . For an arbitrary concept class C , the best worst-case algorithm will make $Opt(C)$ mistakes, where

$$VC(C) \leq Opt(C) \leq \log_2(|C|)$$

- The WEIGHTED-MAJORITY algorithm combines the weighted votes of multiple prediction algorithms to classify new instances. It learns weights for each of these prediction algorithms based on errors made over a sequence of examples. Interestingly, the number of mistakes made by WEIGHTED-MAJORITY can be bounded in terms of the number of mistakes made by the best prediction algorithm in the pool.

Much early work on computational learning theory dealt with the question of whether the learner could identify the target concept in the limit, given an indefinitely long sequence of training examples. The identification in the limit model was introduced by Gold (1967). A good overview of results in this area is (Angluin 1992). Vapnik (1982) examines in detail the problem of uniform convergence, and the closely related PAC-learning model was introduced by Valiant (1984). The discussion in this chapter of ϵ -exhausting the version space is based on Haussler's (1988) exposition. A useful collection of results under the PAC model can be found in Blumer et al. (1989). Kearns and Vazirani (1994) provide an excellent exposition of many results from computational learning theory. Earlier texts in this area include Anthony and Biggs (1992) and Natarajan (1991).

Current research on computational learning theory covers a broad range of learning models and learning algorithms. Much of this research can be found in the proceedings of the annual conference on Computational Learning Theory (COLT). Several special issues of the journal *Machine Learning* have also been devoted to this topic.

EXERCISES

- 7.1. Consider training a two-input perceptron. Give an upper bound on the number of training examples sufficient to assure with 90% confidence that the learned perceptron will have true error of at most 5%. Does this bound seem realistic?
- 7.2. Consider the class C of concepts of the form $(a \leq x \leq b) \wedge (c \leq y \leq d)$, where a, b, c , and d are integers in the interval $(0, 99)$. Note each concept in this class corresponds to a rectangle with integer-valued boundaries on a portion of the x, y plane. Hint: Given a region in the plane bounded by the points $(0, 0)$ and $(n - 1, n - 1)$, the number of distinct rectangles with integer-valued boundaries within this region is $\left(\frac{n(n+1)}{2}\right)^2$.
 - (a) Give an upper bound on the number of randomly drawn training examples sufficient to assure that for any target concept c in C , any consistent learner using $H = C$ will, with probability 95%, output a hypothesis with error at most .15.
 - (b) Now suppose the rectangle boundaries a, b, c , and d take on *real* values instead of integer values. Update your answer to the first part of this question.
- 7.3. In this chapter we derived an expression for the number of training examples sufficient to ensure that every hypothesis will have true error no worse than ϵ plus its observed training error $\text{error}_D(h)$. In particular, we used Hoeffding bounds to derive Equation (7.3). Derive an alternative expression for the number of training examples sufficient to ensure that every hypothesis will have true error no worse than $(1 + \gamma)\text{error}_D(h)$. You can use the general Chernoff bounds to derive such a result.

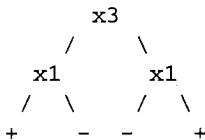
Chernoff bounds: Suppose X_1, \dots, X_m are the outcomes of m independent coin flips (Bernoulli trials), where the probability of heads on any single trial is $\Pr[X_i = 1] = p$ and the probability of tails is $\Pr[X_i = 0] = 1 - p$. Define $S = X_1 + X_2 + \dots + X_m$ to be the sum of the outcomes of these m trials. The expected value of S/m is $E[S/m] = p$. The Chernoff bounds govern the probability that S/m will differ from p by some factor $0 \leq \gamma \leq 1$.

$$\Pr[S/m > (1 + \gamma)p] \leq e^{-mp\gamma^2/3}$$

$$\Pr[S/m < (1 - \gamma)p] \leq e^{-mp\gamma^2/2}$$

- 7.4. Consider a learning problem in which $X = \mathbb{R}$ is the set of real numbers, and $C = H$ is the set of intervals over the reals, $H = \{(a < x < b) \mid a, b \in \mathbb{R}\}$. What is the probability that a hypothesis consistent with m examples of this target concept will have error at least ϵ ? Solve this using the VC dimension. Can you find a second way to solve this, based on first principles and ignoring the VC dimension?

- 7.5. Consider the space of instances X corresponding to all points in the x, y plane. Give the VC dimension of the following hypothesis spaces:
- H_r = the set of all rectangles in the x, y plane. That is, $H = \{((a < x < b) \wedge (c < y < d)) | a, b, c, d \in \mathbb{R}\}$.
 - H_c = circles in the x, y plane. Points inside the circle are classified as positive examples
 - H_t = triangles in the x, y plane. Points inside the triangle are classified as positive examples
- 7.6. Write a consistent learner for H_r from Exercise 7.5. Generate a variety of target concept rectangles at random, corresponding to different rectangles in the plane. Generate random examples of each of these target concepts, based on a uniform distribution of instances within the rectangle from $\langle 0, 0 \rangle$ to $\langle 100, 100 \rangle$. Plot the generalization error as a function of the number of training examples, m . On the same graph, plot the theoretical relationship between ϵ and m , for $\delta = .95$. Does theory fit experiment?
- 7.7. Consider the hypothesis class H_{rd2} of “regular, depth-2 decision trees” over n Boolean variables. A “regular, depth-2 decision tree” is a depth-2 decision tree (a tree with four leaves, all distance 2 from the root) in which the left and right child of the root are *required to contain the same variable*. For instance, the following tree is in H_{rd2} .



- As a function of n , how many syntactically distinct trees are there in H_{rd2} ?
 - Give an upper bound for the number of examples needed in the PAC model to learn H_{rd2} with error ϵ and confidence δ .
 - Consider the following WEIGHTED-MAJORITY algorithm, for the class H_{rd2} . You begin with all hypotheses in H_{rd2} assigned an initial weight equal to 1. Every time you see a new example, you predict based on a weighted majority vote over all hypotheses in H_{rd2} . Then, instead of eliminating the inconsistent trees, you cut down their weight by a factor of 2. How many mistakes will this procedure make at most, as a function of n and the number of mistakes of the best tree in H_{rd2} ?
- 7.8. This question considers the relationship between the PAC analysis considered in this chapter and the evaluation of hypotheses discussed in Chapter 5. Consider a learning task in which instances are described by n boolean variables (e.g., $x_1 \wedge \bar{x}_2 \wedge x_3 \dots \bar{x}_n$) and are drawn according to a fixed but unknown probability distribution \mathcal{D} . The target concept is known to be describable by a conjunction of boolean attributes and their negations (e.g., $x_2 \wedge \bar{x}_5$), and the learning algorithm uses this concept class as its hypothesis space H . A consistent learner is provided a set of 100 training examples drawn according to \mathcal{D} . It outputs a hypothesis h from H that is consistent with all 100 examples (i.e., the error of h over these training examples is zero).
- We are interested in the true error of h , that is, the probability that it will misclassify future instances drawn randomly according to \mathcal{D} . Based on the above information, can you give an interval into which this true error will fall with at least 95% probability? If so, state it and justify it briefly. If not, explain the difficulty.

- (b) You now draw a new set of 100 instances, drawn independently according to the same distribution \mathcal{D} . You find that h misclassifies 30 of these 100 new examples. Can you give an interval into which this true error will fall with approximately 95% probability? (Ignore the performance over the earlier training data for this part.) If so, state it and justify it briefly. If not, explain the difficulty.
- (c) It may seem a bit odd that h misclassifies 30% of the new examples even though it perfectly classified the training examples. Is this event more likely for large n or small n ? Justify your answer in a sentence.

REFERENCES

- Angluin, D. (1992). Computational learning theory: Survey and selected bibliography. *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing* (pp. 351–369). ACM Press.
- Angluin, D., Frazier, M., & Pitt, L. (1992). Learning conjunctions of horn clauses. *Machine Learning*, 9, 147–164.
- Anthony, M., & Biggs, N. (1992). *Computational learning theory: An introduction*. Cambridge, England: Cambridge University Press.
- Blumer, A., Ehrenfeucht, A., Haussler, D., & Warmuth, M. (1989). Learnability and the Vapnik-Chervonenkis dimension. *Journal of the ACM*, 36(4) (October), 929–965.
- Ehrenfeucht, A., Haussler, D., Kearns, M., & Valiant, L. (1989). A general lower bound on the number of examples needed for learning. *Information and Computation*, 82, 247–261.
- Gold, E. M. (1967). Language identification in the limit. *Information and Control*, 10, 447–474.
- Goldman, S. (Ed.). (1995). Special issue on computational learning theory. *Machine Learning*, 18(2/3), February.
- Haussler, D. (1988). Quantifying inductive bias: AI learning algorithms and Valiant's learning framework. *Artificial Intelligence*, 36, 177–221.
- Kearns, M. J., & Vazirani, U. V. (1994). *An introduction to computational learning theory*. Cambridge, MA: MIT Press.
- Laird, P. (1988). *Learning from good and bad data*. Dordrecht: Kluwer Academic Publishers.
- Li, M., & Valiant, L. G. (Eds.). (1994). Special issue on computational learning theory. *Machine Learning*, 14(1).
- Littlestone, N. (1987). Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2, 285–318.
- Littlestone, N., & Warmuth, M. (1991). *The weighted majority algorithm* (Technical report UCSC-CRL-91-28). Univ. of California Santa Cruz, Computer Engineering and Information Sciences Dept., Santa Cruz, CA.
- Littlestone, N., & Warmuth, M. (1994). The weighted majority algorithm. *Information and Computation* (108), 212–261.
- Pitt, L. (Ed.). (1990). Special issue on computational learning theory. *Machine Learning*, 5(2).
- Natarajan, B. K. (1991). *Machine learning: A theoretical approach*. San Mateo, CA: Morgan Kaufmann.
- Valiant, L. (1984). A theory of the learnable. *Communications of the ACM*, 27(11), 1134–1142.
- Vapnik, V. N. (1982). *Estimation of dependences based on empirical data*. New York: Springer-Verlag.
- Vapnik, V. N., & Chervonenkis, A. (1971). On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and Its Applications*, 16, 264–280.

CHAPTER

8

INSTANCE-BASED LEARNING

In contrast to learning methods that construct a general, explicit description of the target function when training examples are provided, instance-based learning methods simply store the training examples. Generalizing beyond these examples is postponed until a new instance must be classified. Each time a new query instance is encountered, its relationship to the previously stored examples is examined in order to assign a target function value for the new instance. Instance-based learning includes nearest neighbor and locally weighted regression methods that assume instances can be represented as points in a Euclidean space. It also includes case-based reasoning methods that use more complex, symbolic representations for instances. Instance-based methods are sometimes referred to as “lazy” learning methods because they delay processing until a new instance must be classified. A key advantage of this kind of delayed, or lazy, learning is that instead of estimating the target function once for the entire instance space, these methods can estimate it locally and differently for each new instance to be classified.

8.1 INTRODUCTION

Instance-based learning methods such as nearest neighbor and locally weighted regression are conceptually straightforward approaches to approximating real-valued or discrete-valued target functions. Learning in these algorithms consists of simply storing the presented training data. When a new query instance is encountered, a set of similar related instances is retrieved from memory and used to classify the

new query instance. One key difference between these approaches and the methods discussed in other chapters is that instance-based approaches can construct a different approximation to the target function for each distinct query instance that must be classified. In fact, many techniques construct only a local approximation to the target function that applies in the neighborhood of the new query instance, and never construct an approximation designed to perform well over the entire instance space. This has significant advantages when the target function is very complex, but can still be described by a collection of less complex local approximations.

Instance-based methods can also use more complex, symbolic representations for instances. In case-based learning, instances are represented in this fashion and the process for identifying “neighboring” instances is elaborated accordingly. Case-based reasoning has been applied to tasks such as storing and reusing past experience at a help desk, reasoning about legal cases by referring to previous cases, and solving complex scheduling problems by reusing relevant portions of previously solved problems.

One disadvantage of instance-based approaches is that the cost of classifying new instances can be high. This is due to the fact that nearly all computation takes place at classification time rather than when the training examples are first encountered. Therefore, techniques for efficiently indexing training examples are a significant practical issue in reducing the computation required at query time. A second disadvantage to many instance-based approaches, especially nearest-neighbor approaches, is that they typically consider *all* attributes of the instances when attempting to retrieve similar training examples from memory. If the target concept depends on only a few of the many available attributes, then the instances that are truly most “similar” may well be a large distance apart.

In the next section we introduce the *k*-NEAREST NEIGHBOR learning algorithm, including several variants of this widely-used approach. The subsequent section discusses locally weighted regression, a learning method that constructs local approximations to the target function and that can be viewed as a generalization of *k*-NEAREST NEIGHBOR algorithms. We then describe radial basis function networks, which provide an interesting bridge between instance-based and neural network learning algorithms. The next section discusses case-based reasoning, an instance-based approach that employs symbolic representations and knowledge-based inference. This section includes an example application of case-based reasoning to a problem in engineering design. Finally, we discuss the fundamental differences in capabilities that distinguish lazy learning methods discussed in this chapter from eager learning methods discussed in the other chapters of this book.

8.2 *k*-NEAREST NEIGHBOR LEARNING

The most basic instance-based method is the *k*-NEAREST NEIGHBOR algorithm. This algorithm assumes all instances correspond to points in the *n*-dimensional space \Re^n . The nearest neighbors of an instance are defined in terms of the standard

Euclidean distance. More precisely, let an arbitrary instance x be described by the feature vector

$$\langle a_1(x), a_2(x), \dots, a_n(x) \rangle$$

where $a_r(x)$ denotes the value of the r th attribute of instance x . Then the distance between two instances x_i and x_j is defined to be $d(x_i, x_j)$, where

$$d(x_i, x_j) \equiv \sqrt{\sum_{r=1}^n (a_r(x_i) - a_r(x_j))^2}$$

In nearest-neighbor learning the target function may be either discrete-valued or real-valued. Let us first consider learning discrete-valued target functions of the form $f : \mathbb{R}^n \rightarrow V$, where V is the finite set $\{v_1, \dots, v_s\}$. The k -NEAREST NEIGHBOR algorithm for approximating a discrete-valued target function is given in Table 8.1. As shown there, the value $\hat{f}(x_q)$ returned by this algorithm as its estimate of $f(x_q)$ is just the most common value of f among the k training examples nearest to x_q . If we choose $k = 1$, then the 1-NEAREST NEIGHBOR algorithm assigns to $\hat{f}(x_q)$ the value $f(x_i)$ where x_i is the training instance nearest to x_q . For larger values of k , the algorithm assigns the most common value among the k nearest training examples.

Figure 8.1 illustrates the operation of the k -NEAREST NEIGHBOR algorithm for the case where the instances are points in a two-dimensional space and where the target function is boolean valued. The positive and negative training examples are shown by “+” and “-” respectively. A query point x_q is shown as well. Note the 1-NEAREST NEIGHBOR algorithm classifies x_q as a positive example in this figure, whereas the 5-NEAREST NEIGHBOR algorithm classifies it as a negative example.

What is the nature of the hypothesis space H implicitly considered by the k -NEAREST NEIGHBOR algorithm? Note the k -NEAREST NEIGHBOR algorithm never forms an explicit general hypothesis \hat{f} regarding the target function f . It simply computes the classification of each new query instance as needed. Nevertheless,

Training algorithm:

- For each training example $(x, f(x))$, add the example to the list *training_examples*

Classification algorithm:

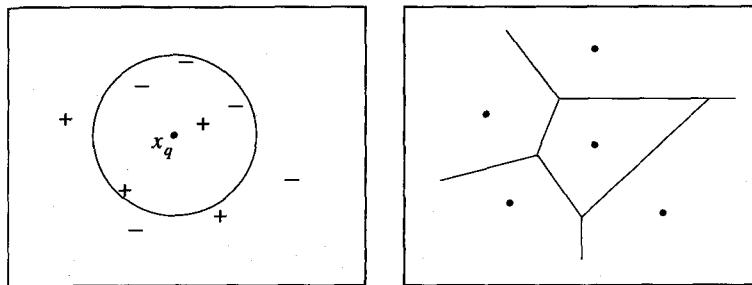
- Given a query instance x_q to be classified,
 - Let $x_1 \dots x_k$ denote the k instances from *training_examples* that are nearest to x_q
 - Return

$$\hat{f}(x_q) \leftarrow \operatorname{argmax}_{v \in V} \sum_{i=1}^k \delta(v, f(x_i))$$

where $\delta(a, b) = 1$ if $a = b$ and where $\delta(a, b) = 0$ otherwise.

TABLE 8.1

The k -NEAREST NEIGHBOR algorithm for approximating a discrete-valued function $f : \mathbb{R}^n \rightarrow V$.

**FIGURE 8.1**

k-NEAREST NEIGHBOR. A set of positive and negative training examples is shown on the left, along with a query instance x_q to be classified. The 1-NEAREST NEIGHBOR algorithm classifies x_q positive, whereas 5-NEAREST NEIGHBOR classifies it as negative. On the right is the decision surface induced by the 1-NEAREST NEIGHBOR algorithm for a typical set of training examples. The convex polygon surrounding each training example indicates the region of instance space closest to that point (i.e., the instances for which the 1-NEAREST NEIGHBOR algorithm will assign the classification belonging to that training example).

we can still ask what the implicit general function is, or what classifications would be assigned if we were to hold the training examples constant and query the algorithm with every possible instance in X . The diagram on the right side of Figure 8.1 shows the shape of this decision surface induced by 1-NEAREST NEIGHBOR over the entire instance space. The decision surface is a combination of convex polyhedra surrounding each of the training examples. For every training example, the polyhedron indicates the set of query points whose classification will be completely determined by that training example. Query points outside the polyhedron are closer to some other training example. This kind of diagram is often called the *Voronoi diagram* of the set of training examples.

The *k*-NEAREST NEIGHBOR algorithm is easily adapted to approximating continuous-valued target functions. To accomplish this, we have the algorithm calculate the mean value of the *k* nearest training examples rather than calculate their most common value. More precisely, to approximate a real-valued target function $f : \Re^n \rightarrow \Re$ we replace the final line of the above algorithm by the line

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^k f(x_i)}{k} \quad (8.1)$$

8.2.1 Distance-Weighted NEAREST NEIGHBOR Algorithm

One obvious refinement to the *k*-NEAREST NEIGHBOR algorithm is to weight the contribution of each of the *k* neighbors according to their distance to the query point x_q , giving greater weight to closer neighbors. For example, in the algorithm of Table 8.1, which approximates discrete-valued target functions, we might weight the vote of each neighbor according to the inverse square of its distance from x_q .

This can be accomplished by replacing the final line of the algorithm by

$$\hat{f}(x_q) \leftarrow \operatorname{argmax}_{v \in V} \sum_{i=1}^k w_i \delta(v, f(x_i)) \quad (8.2)$$

where

$$w_i \equiv \frac{1}{d(x_q, x_i)^2} \quad (8.3)$$

To accommodate the case where the query point x_q exactly matches one of the training instances x_i and the denominator $d(x_q, x_i)^2$ is therefore zero, we assign $\hat{f}(x_q)$ to be $f(x_i)$ in this case. If there are several such training examples, we assign the majority classification among them.

We can distance-weight the instances for real-valued target functions in a similar fashion, replacing the final line of the algorithm in this case by

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^k w_i f(x_i)}{\sum_{i=1}^k w_i} \quad (8.4)$$

where w_i is as defined in Equation (8.3). Note the denominator in Equation (8.4) is a constant that normalizes the contributions of the various weights (e.g., it assures that if $f(x_i) = c$ for all training examples, then $\hat{f}(x_q) \leftarrow c$ as well).

Note all of the above variants of the k -NEAREST NEIGHBOR algorithm consider only the k nearest neighbors to classify the query point. Once we add distance weighting, there is really no harm in allowing all training examples to have an influence on the classification of the x_q , because very distant examples will have very little effect on $\hat{f}(x_q)$. The only disadvantage of considering all examples is that our classifier will run more slowly. If all training examples are considered when classifying a new query instance, we call the algorithm a *global* method. If only the nearest training examples are considered, we call it a *local* method. When the rule in Equation (8.4) is applied as a global method, using all training examples, it is known as Shepard's method (Shepard 1968).

8.2.2 Remarks on k -NEAREST NEIGHBOR Algorithm

The distance-weighted k -NEAREST NEIGHBOR algorithm is a highly effective inductive inference method for many practical problems. It is robust to noisy training data and quite effective when it is provided a sufficiently large set of training data. Note that by taking the weighted average of the k neighbors nearest to the query point, it can smooth out the impact of isolated noisy training examples.

What is the inductive bias of k -NEAREST NEIGHBOR? The basis for classifying new query points is easily understood based on the diagrams in Figure 8.1. The inductive bias corresponds to an assumption that the classification of an instance x_q will be most similar to the classification of other instances that are nearby in Euclidean distance.

One practical issue in applying k -NEAREST NEIGHBOR algorithms is that the distance between instances is calculated based on *all* attributes of the instance

(i.e., on all axes in the Euclidean space containing the instances). This lies in contrast to methods such as rule and decision tree learning systems that select only a subset of the instance attributes when forming the hypothesis. To see the effect of this policy, consider applying *k*-NEAREST NEIGHBOR to a problem in which each instance is described by 20 attributes, but where only 2 of these attributes are relevant to determining the classification for the particular target function. In this case, instances that have identical values for the 2 relevant attributes may nevertheless be distant from one another in the 20-dimensional instance space. As a result, the similarity metric used by *k*-NEAREST NEIGHBOR—depending on all 20 attributes—will be misleading. The distance between neighbors will be dominated by the large number of irrelevant attributes. This difficulty, which arises when many irrelevant attributes are present, is sometimes referred to as the *curse of dimensionality*. Nearest-neighbor approaches are especially sensitive to this problem.

One interesting approach to overcoming this problem is to weight each attribute differently when calculating the distance between two instances. This corresponds to stretching the axes in the Euclidean space, shortening the axes that correspond to less relevant attributes, and lengthening the axes that correspond to more relevant attributes. The amount by which each axis should be stretched can be determined automatically using a cross-validation approach. To see how, first note that we wish to stretch (multiply) the j th axis by some factor z_j , where the values $z_1 \dots z_n$ are chosen to minimize the true classification error of the learning algorithm. Second, note that this true error can be estimated using cross-validation. Hence, one algorithm is to select a random subset of the available data to use as training examples, then determine the values of $z_1 \dots z_n$ that lead to the minimum error in classifying the remaining examples. By repeating this process multiple times the estimate for these weighting factors can be made more accurate. This process of stretching the axes in order to optimize the performance of *k*-NEAREST NEIGHBOR provides a mechanism for suppressing the impact of irrelevant attributes.

An even more drastic alternative is to completely eliminate the least relevant attributes from the instance space. This is equivalent to setting some of the z_i scaling factors to zero. Moore and Lee (1994) discuss efficient cross-validation methods for selecting relevant subsets of the attributes for *k*-NEAREST NEIGHBOR algorithms. In particular, they explore methods based on leave-one-out cross-validation, in which the set of m training instances is repeatedly divided into a training set of size $m - 1$ and test set of size 1, in all possible ways. This leave-one-out approach is easily implemented in *k*-NEAREST NEIGHBOR algorithms because no additional training effort is required each time the training set is redefined. Note both of the above approaches can be seen as stretching each axis by some constant factor. Alternatively, we could stretch each axis by a value that varies over the instance space. However, as we increase the number of degrees of freedom available to the algorithm for redefining its distance metric in such a fashion, we also increase the risk of overfitting. Therefore, the approach of locally stretching the axes is much less common.

One additional practical issue in applying *k*-NEAREST NEIGHBOR is efficient memory indexing. Because this algorithm delays all processing until a new query is received, significant computation can be required to process each new query. Various methods have been developed for indexing the stored training examples so that the nearest neighbors can be identified more efficiently at some additional cost in memory. One such indexing method is the *kd*-tree (Bentley 1975; Friedman et al. 1977), in which instances are stored at the leaves of a tree, with nearby instances stored at the same or nearby nodes. The internal nodes of the tree sort the new query x_q to the relevant leaf by testing selected attributes of x_q .

8.2.3 A Note on Terminology

Much of the literature on nearest-neighbor methods and weighted local regression uses a terminology that has arisen from the field of statistical pattern recognition. In reading that literature, it is useful to know the following terms:

- *Regression* means approximating a real-valued target function.
- *Residual* is the error $\hat{f}(x) - f(x)$ in approximating the target function.
- *Kernel function* is the function of distance that is used to determine the weight of each training example. In other words, the kernel function is the function K such that $w_i = K(d(x_i, x_q))$.

8.3 LOCALLY WEIGHTED REGRESSION

The nearest-neighbor approaches described in the previous section can be thought of as approximating the target function $f(x)$ at the single query point $x = x_q$. Locally weighted regression is a generalization of this approach. It constructs an explicit approximation to f over a local region surrounding x_q . Locally weighted regression uses nearby or distance-weighted training examples to form this local approximation to f . For example, we might approximate the target function in the neighborhood surrounding x_q using a linear function, a quadratic function, a multilayer neural network, or some other functional form. The phrase “locally weighted regression” is called *local* because the function is approximated based only on data near the query point, *weighted* because the contribution of each training example is weighted by its distance from the query point, and *regression* because this is the term used widely in the statistical learning community for the problem of approximating real-valued functions.

Given a new query instance x_q , the general approach in locally weighted regression is to construct an approximation \hat{f} that fits the training examples in the neighborhood surrounding x_q . This approximation is then used to calculate the value $\hat{f}(x_q)$, which is output as the estimated target value for the query instance. The description of \hat{f} may then be deleted, because a different local approximation will be calculated for each distinct query instance.

8.3.1 Locally Weighted Linear Regression

Let us consider the case of locally weighted regression in which the target function f is approximated near x_q using a linear function of the form

$$\hat{f}(x) = w_0 + w_1 a_1(x) + \cdots + w_n a_n(x)$$

As before, $a_i(x)$ denotes the value of the i th attribute of the instance x .

Recall that in Chapter 4 we discussed methods such as gradient descent to find the coefficients $w_0 \dots w_n$ to minimize the error in fitting such linear functions to a given set of training examples. In that chapter we were interested in a global approximation to the target function. Therefore, we derived methods to choose weights that minimize the squared error summed over the set D of training examples

$$E \equiv \frac{1}{2} \sum_{x \in D} (f(x) - \hat{f}(x))^2 \quad (8.5)$$

which led us to the gradient descent training rule

$$\Delta w_j = \eta \sum_{x \in D} (f(x) - \hat{f}(x)) a_j(x) \quad (8.6)$$

where η is a constant learning rate, and where the training rule has been re-expressed from the notation of Chapter 4 to fit our current notation (i.e., $t \rightarrow f(x)$, $o \rightarrow \hat{f}(x)$, and $x_j \rightarrow a_j(x)$).

How shall we modify this procedure to derive a local approximation rather than a global one? The simple way is to redefine the error criterion E to emphasize fitting the local training examples. Three possible criteria are given below. Note we write the error $E(x_q)$ to emphasize the fact that now the error is being defined as a function of the query point x_q .

1. Minimize the squared error over just the k nearest neighbors:

$$E_1(x_q) \equiv \frac{1}{2} \sum_{x \in k \text{ nearest nbrs of } x_q} (f(x) - \hat{f}(x))^2$$

2. Minimize the squared error over the entire set D of training examples, while weighting the error of each training example by some decreasing function K of its distance from x_q :

$$E_2(x_q) \equiv \frac{1}{2} \sum_{x \in D} (f(x) - \hat{f}(x))^2 K(d(x_q, x))$$

3. Combine 1 and 2:

$$E_3(x_q) \equiv \frac{1}{2} \sum_{x \in k \text{ nearest nbrs of } x_q} (f(x) - \hat{f}(x))^2 K(d(x_q, x))$$

Criterion two is perhaps the most esthetically pleasing because it allows every training example to have an impact on the classification of x_q . However,

this approach requires computation that grows linearly with the number of training examples. Criterion three is a good approximation to criterion two and has the advantage that computational cost is independent of the total number of training examples; its cost depends only on the number k of neighbors considered.

If we choose criterion three above and rederive the gradient descent rule using the same style of argument as in Chapter 4, we obtain the following training rule (see Exercise 8.1):

$$\Delta w_j = \eta \sum_{x \in k \text{ nearest nbrs of } x_q} K(d(x_q, x)) (f(x) - \hat{f}(x)) a_j(x) \quad (8.7)$$

Notice the only differences between this new rule and the rule given by Equation (8.6) are that the contribution of instance x to the weight update is now multiplied by the distance penalty $K(d(x_q, x))$, and that the error is summed over only the k nearest training examples. In fact, if we are fitting a linear function to a fixed set of training examples, then methods much more efficient than gradient descent are available to directly solve for the desired coefficients $w_0 \dots w_n$. Atkeson et al. (1997a) and Bishop (1995) survey several such methods.

8.3.2 Remarks on Locally Weighted Regression

Above we considered using a linear function to approximate f in the neighborhood of the query instance x_q . The literature on locally weighted regression contains a broad range of alternative methods for distance weighting the training examples, and a range of methods for locally approximating the target function. In most cases, the target function is approximated by a constant, linear, or quadratic function. More complex functional forms are not often found because (1) the cost of fitting more complex functions for each query instance is prohibitively high, and (2) these simple approximations model the target function quite well over a sufficiently small subregion of the instance space.

8.4 RADIAL BASIS FUNCTIONS

One approach to function approximation that is closely related to distance-weighted regression and also to artificial neural networks is learning with radial basis functions (Powell 1987; Broomhead and Lowe 1988; Moody and Darken 1989). In this approach, the learned hypothesis is a function of the form

$$\hat{f}(x) = w_0 + \sum_{u=1}^k w_u K_u(d(x_u, x)) \quad (8.8)$$

where each x_u is an instance from X and where the kernel function $K_u(d(x_u, x))$ is defined so that it decreases as the distance $d(x_u, x)$ increases. Here k is a user-provided constant that specifies the number of kernel functions to be included. Even though $\hat{f}(x)$ is a global approximation to $f(x)$, the contribution from each of the $K_u(d(x_u, x))$ terms is localized to a region nearby the point x_u . It is common

to choose each function $K_u(d(x_u, x))$ to be a Gaussian function (see Table 5.4) centered at the point x_u with some variance σ_u^2 .

$$K_u(d(x_u, x)) = e^{-\frac{1}{2\sigma_u^2}d^2(x_u, x)}$$

We will restrict our discussion here to this common Gaussian kernel function. As shown by Hartman et al. (1990), the functional form of Equation (8.8) can approximate any function with arbitrarily small error, provided a sufficiently large number k of such Gaussian kernels and provided the width σ^2 of each kernel can be separately specified.

The function given by Equation (8.8) can be viewed as describing a two-layer network where the first layer of units computes the values of the various $K_u(d(x_u, x))$ and where the second layer computes a linear combination of these first-layer unit values. An example radial basis function (RBF) network is illustrated in Figure 8.2.

Given a set of training examples of the target function, RBF networks are typically trained in a two-stage process. First, the number k of hidden units is determined and each hidden unit u is defined by choosing the values of x_u and σ_u^2 that define its kernel function $K_u(d(x_u, x))$. Second, the weights w_u are trained to maximize the fit of the network to the training data, using the global error criterion given by Equation (8.5). Because the kernel functions are held fixed during this second stage, the linear weight values w_u can be trained very efficiently.

Several alternative methods have been proposed for choosing an appropriate number of hidden units or, equivalently, kernel functions. One approach is to allocate a Gaussian kernel function for each training example $\langle x_i, f(x_i) \rangle$, centering this Gaussian at the point x_i . Each of these kernels may be assigned the same width σ^2 . Given this approach, the RBF network learns a global approximation to the target function in which each training example $\langle x_i, f(x_i) \rangle$ can influence the value of \hat{f} only in the neighborhood of x_i . One advantage of this choice of kernel functions is that it allows the RBF network to fit the training data exactly. That is, for any set of m training examples the weights $w_0 \dots w_m$ for combining the m Gaussian kernel functions can be set so that $\hat{f}(x_i) = f(x_i)$ for each training example $\langle x_i, f(x_i) \rangle$.

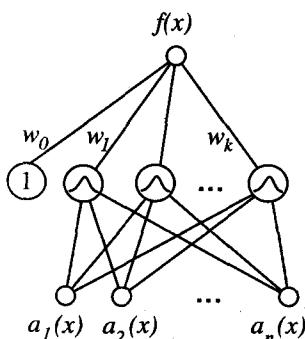


FIGURE 8.2

A radial basis function network. Each hidden unit produces an activation determined by a Gaussian function centered at some instance x_u . Therefore, its activation will be close to zero unless the input x is near x_u . The output unit produces a linear combination of the hidden unit activations. Although the network shown here has just one output, multiple output units can also be included.

A second approach is to choose a set of kernel functions that is smaller than the number of training examples. This approach can be much more efficient than the first approach, especially when the number of training examples is large. The set of kernel functions may be distributed with centers spaced uniformly throughout the instance space X . Alternatively, we may wish to distribute the centers nonuniformly, especially if the instances themselves are found to be distributed nonuniformly over X . In this later case, we can pick kernel function centers by randomly selecting a subset of the training instances, thereby sampling the underlying distribution of instances. Alternatively, we may identify prototypical clusters of instances, then add a kernel function centered at each cluster. The placement of the kernel functions in this fashion can be accomplished using unsupervised clustering algorithms that fit the training instances (but not their target values) to a mixture of Gaussians. The EM algorithm discussed in Section 6.12.1 provides one algorithm for choosing the means of a mixture of k Gaussians to best fit the observed instances. In the case of the EM algorithm, the means are chosen to maximize the probability of observing the instances x_i , given the k estimated means. Note the target function value $f(x_i)$ of the instance does not enter into the calculation of kernel centers by unsupervised clustering methods. The only role of the target values $f(x_i)$ in this case is to determine the output layer weights w_u .

To summarize, radial basis function networks provide a global approximation to the target function, represented by a linear combination of many local kernel functions. The value for any given kernel function is non-negligible only when the input x falls into the region defined by its particular center and width. Thus, the network can be viewed as a smooth linear combination of many local approximations to the target function. One key advantage to RBF networks is that they can be trained much more efficiently than feedforward networks trained with BACKPROPAGATION. This follows from the fact that the input layer and the output layer of an RBF are trained separately.

8.5 CASE-BASED REASONING

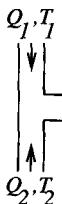
Instance-based methods such as k -NEAREST NEIGHBOR and locally weighted regression share three key properties. First, they are *lazy* learning methods in that they defer the decision of how to generalize beyond the training data until a new query instance is observed. Second, they classify new query instances by analyzing similar instances while ignoring instances that are very different from the query. Third, they represent instances as real-valued points in an n -dimensional Euclidean space. Case-based reasoning (CBR) is a learning paradigm based on the first two of these principles, but not the third. In CBR, instances are typically represented using more rich symbolic descriptions, and the methods used to retrieve similar instances are correspondingly more elaborate. CBR has been applied to problems such as conceptual design of mechanical devices based on a stored library of previous designs (Sycara et al. 1992), reasoning about new legal cases based on previous rulings (Ashley 1990), and solving planning and

scheduling problems by reusing and combining portions of previous solutions to similar problems (Veloso 1992).

Let us consider a prototypical example of a case-based reasoning system to ground our discussion. The CADET system (Sycara et al. 1992) employs case-based reasoning to assist in the conceptual design of simple mechanical devices such as water faucets. It uses a library containing approximately 75 previous designs and design fragments to suggest conceptual designs to meet the specifications of new design problems. Each instance stored in memory (e.g., a water pipe) is represented by describing both its structure and its qualitative function. New design problems are then presented by specifying the desired function and requesting the corresponding structure. This problem setting is illustrated in Figure 8.3. The top half of the figure shows the description of a typical stored case called a T-junction pipe. Its function is represented in terms of the qualitative relationships among the waterflow levels and temperatures at its inputs and outputs. In the functional description at its right, an arrow with a “+” label indicates that the variable at the arrowhead increases with the variable at its tail. For example, the output waterflow Q_3 increases with increasing input waterflow Q_1 . Similarly,

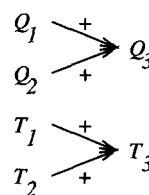
A stored case: T-junction pipe

Structure:



T = temperature
 Q = waterflow

Function:



A problem specification: Water faucet

Structure:

?

Function:

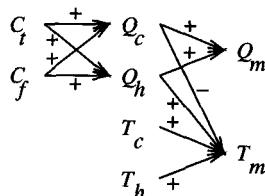


FIGURE 8.3

A stored case and a new problem. The top half of the figure describes a typical design fragment in the case library of CADET. The function is represented by the graph of qualitative dependencies among the T-junction variables (described in the text). The bottom half of the figure shows a typical design problem.

a “-” label indicates that the variable at the head decreases with the variable at the tail. The bottom half of this figure depicts a new design problem described by its desired function. This particular function describes the required behavior of one type of water faucet. Here Q_c refers to the flow of cold water into the faucet, Q_h to the input flow of hot water, and Q_m to the single mixed flow out of the faucet. Similarly, T_c , T_h , and T_m refer to the temperatures of the cold water, hot water, and mixed water respectively. The variable C_t denotes the control signal for temperature that is input to the faucet, and C_f denotes the control signal for waterflow. Note the description of the desired function specifies that these controls C_t and C_f are to influence the water flows Q_c and Q_h , thereby indirectly influencing the faucet output flow Q_m and temperature T_m .

Given this functional specification for the new design problem, CADET searches its library for stored cases whose functional descriptions match the design problem. If an exact match is found, indicating that some stored case implements exactly the desired function, then this case can be returned as a suggested solution to the design problem. If no exact match occurs, CADET may find cases that match various subgraphs of the desired functional specification. In Figure 8.3, for example, the T-junction function matches a subgraph of the water faucet function graph. More generally, CADET searches for subgraph isomorphisms between the two function graphs, so that parts of a case can be found to match parts of the design specification. Furthermore, the system may elaborate the original function specification graph in order to create functionally equivalent graphs that may match still more cases. It uses general knowledge about physical influences to create these elaborated function graphs. For example, it uses a rewrite rule that allows it to rewrite the influence

$$A \xrightarrow{+} B$$

as

$$A \xrightarrow{+} x \xrightarrow{+} B$$

This rewrite rule can be interpreted as stating that if B must increase with A , then it is sufficient to find some other quantity x such that B increases with x , and x increases with A . Here x is a universally quantified variable whose value is bound when matching the function graph against the case library. In fact, the function graph for the faucet shown in Figure 8.3 is an elaboration of the original functional specification produced by applying such rewrite rules.

By retrieving multiple cases that match different subgraphs, the entire design can sometimes be pieced together. In general, the process of producing a final solution from multiple retrieved cases can be very complex. It may require designing portions of the system from first principles, in addition to merging retrieved portions from stored cases. It may also require backtracking on earlier choices of design subgoals and, therefore, rejecting cases that were previously retrieved. CADET has very limited capabilities for combining and adapting multiple retrieved cases to form the final design and relies heavily on the user for this adaptation stage of the process. As described by Sycara et al. (1992), CADET is

a research prototype system intended to explore the potential role of case-based reasoning in conceptual design. It does not have the range of analysis algorithms needed to refine these abstract conceptual designs into final designs.

It is instructive to examine the correspondence between the problem setting of CADET and the general setting for instance-based methods such as *k*-NEAREST NEIGHBOR. In CADET each stored training example describes a function graph along with the structure that implements it. New queries correspond to new function graphs. Thus, we can map the CADET problem into our standard notation by defining the space of instances X to be the space of all function graphs. The target function f maps function graphs to the structures that implement them. Each stored training example $\langle x, f(x) \rangle$ is a pair that describes some function graph x and the structure $f(x)$ that implements x . The system must learn from the training example cases to output the structure $f(x_q)$ that successfully implements the input function graph query x_q .

The above sketch of the CADET system illustrates several generic properties of case-based reasoning systems that distinguish them from approaches such as *k*-NEAREST NEIGHBOR.

- Instances or cases may be represented by rich symbolic descriptions, such as the function graphs used in CADET. This may require a similarity metric different from Euclidean distance, such as the size of the largest shared subgraph between two function graphs.
- Multiple retrieved cases may be combined to form the solution to the new problem. This is similar to the *k*-NEAREST NEIGHBOR approach, in that multiple similar cases are used to construct a response for the new query. However, the process for combining these multiple retrieved cases can be very different, relying on knowledge-based reasoning rather than statistical methods.
- There may be a tight coupling between case retrieval, knowledge-based reasoning, and problem solving. One simple example of this is found in CADET, which uses generic knowledge about influences to rewrite function graphs during its attempt to find matching cases. Other systems have been developed that more fully integrate case-based reasoning into general search-based problem-solving systems. Two examples are ANAPRON (Golding and Rosenbloom 1991) and PRODIGY/ANALOGY (Veloso 1992).

To summarize, case-based reasoning is an instance-based learning method in which instances (cases) may be rich relational descriptions and in which the retrieval and combination of cases to solve the current query may rely on knowledge-based reasoning and search-intensive problem-solving methods. One current research issue in case-based reasoning is to develop improved methods for indexing cases. The central issue here is that syntactic similarity measures (e.g., subgraph isomorphism between function graphs) provide only an approximate indication of the relevance of a particular case to a particular problem. When the CBR system attempts to reuse the retrieved cases it may uncover difficulties that were not

captured by this syntactic similarity measure. For example, in CADET the multiple retrieved design fragments may turn out to be incompatible with one another, making it impossible to combine them into a consistent final design. When this occurs in general, the CBR system may backtrack and search for additional cases, adapt the existing cases, or resort to other problem-solving methods. Importantly, when such difficulties are detected they also provide training data for improving the similarity metric or, equivalently, the indexing structure for the case library. In particular, if a case is retrieved based on the similarity metric, but found to be irrelevant based on further analysis, then the similarity metric should be refined to reject this case for similar subsequent queries.

8.6 REMARKS ON LAZY AND EAGER LEARNING

In this chapter we considered three *lazy* learning methods: the *k*-NEAREST NEIGHBOR algorithm, locally weighted regression, and case-based reasoning. We call these methods lazy because they defer the decision of how to generalize beyond the training data until each new query instance is encountered. We also discussed one *eager* learning method: the method for learning radial basis function networks. We call this method eager because it generalizes beyond the training data before observing the new query, committing at training time to the network structure and weights that define its approximation to the target function. In this same sense, every other algorithm discussed elsewhere in this book (e.g., BACKPROPAGATION, C4.5) is an eager learning algorithm.

Are there important differences in what can be achieved by lazy versus eager learning? Let us distinguish between two kinds of differences: differences in computation time and differences in the classifications produced for new queries. There are obviously differences in computation time between eager and lazy methods. For example, lazy methods will generally require less computation during training, but more computation when they must predict the target value for a new query.

The more fundamental question is whether there are essential differences in the inductive bias that can be achieved by lazy versus eager methods. The key difference between lazy and eager methods in this regard is

- Lazy methods may consider the query instance x_q when deciding how to generalize beyond the training data D .
- Eager methods cannot. By the time they observe the query instance x_q they have already chosen their (global) approximation to the target function.

Does this distinction affect the generalization accuracy of the learner? It does if we require that the lazy and eager learner employ the same hypothesis space H . To illustrate, consider the hypothesis space consisting of linear functions. The locally weighted linear regression algorithm discussed earlier is a lazy learning method based on this hypothesis space. For each new query x_q it generalizes from the training data by choosing a new hypothesis based on the training examples near x_q . In contrast, an eager learner that uses the same hypothesis space of linear functions

must choose its approximation before the queries are observed. The eager learner must therefore commit to a single linear function hypothesis that covers the entire instance space and all future queries. The lazy method effectively uses a richer hypothesis space because it uses many different local linear functions to form its implicit global approximation to the target function. Note this same situation holds for other learners and hypothesis spaces as well. A lazy version of BACKPROPAGATION, for example, could learn a different neural network for each distinct query point, compared to the eager version of BACKPROPAGATION discussed in Chapter 4.

The key point in the above paragraph is that a lazy learner has the option of (implicitly) representing the target function by a combination of many local approximations, whereas an eager learner must commit at training time to a single global approximation. The distinction between eager and lazy learning is thus related to the distinction between global and local approximations to the target function.

Can we create eager methods that use multiple local approximations to achieve the same effects as lazy local methods? Radial basis function networks can be seen as one attempt to achieve this. The RBF learning methods we discussed are eager methods that commit to a global approximation to the target function at training time. However, an RBF network represents this global function as a linear combination of multiple local kernel functions. Nevertheless, because RBF learning methods must commit to the hypothesis before the query point is known, the local approximations they create are not specifically targeted to the query point to the same degree as in a lazy learning method. Instead, RBF networks are built eagerly from local approximations centered around the training examples, or around clusters of training examples, but not around the unknown future query points.

To summarize, lazy methods have the option of selecting a different hypothesis or local approximation to the target function for each query instance. Eager methods using the same hypothesis space are more restricted because they must commit to a single hypothesis that covers the entire instance space. Eager methods can, of course, employ hypothesis spaces that combine multiple local approximations, as in RBF networks. However, even these combined local approximations do not give eager methods the full ability of lazy methods to customize to unknown future query instances.

8.7 SUMMARY AND FURTHER READING

The main points of this chapter include:

- Instance-based learning methods differ from other approaches to function approximation because they delay processing of training examples until they must label a new query instance. As a result, they need not form an explicit hypothesis of the entire target function over the entire instance space, independent of the query instance. Instead, they may form a different local approximation to the target function for each query instance.

- Advantages of instance-based methods include the ability to model complex target functions by a collection of less complex local approximations and the fact that information present in the training examples is never lost (because the examples themselves are stored explicitly). The main practical difficulties include efficiency of labeling new instances (all processing is done at query time rather than in advance), difficulties in determining an appropriate distance metric for retrieving “related” instances (especially when examples are represented by complex symbolic descriptions), and the negative impact of irrelevant features on the distance metric.
- *k*-NEAREST NEIGHBOR is an instance-based algorithm for approximating real-valued or discrete-valued target functions, assuming instances correspond to points in an n -dimensional Euclidean space. The target function value for a new query is estimated from the known values of the k nearest training examples.
- Locally weighted regression methods are a generalization of *k*-NEAREST NEIGHBOR in which an explicit local approximation to the target function is constructed for each query instance. The local approximation to the target function may be based on a variety of functional forms such as constant, linear, or quadratic functions or on spatially localized kernel functions.
- Radial basis function (RBF) networks are a type of artificial neural network constructed from spatially localized kernel functions. These can be seen as a blend of instance-based approaches (spatially localized influence of each kernel function) and neural network approaches (a global approximation to the target function is formed at training time rather than a local approximation at query time). Radial basis function networks have been used successfully in applications such as interpreting visual scenes, in which the assumption of spatially local influences is well-justified.
- Case-based reasoning is an instance-based approach in which instances are represented by complex logical descriptions rather than points in a Euclidean space. Given these complex symbolic descriptions of instances, a rich variety of methods have been proposed for mapping from the training examples to target function values for new instances. Case-based reasoning methods have been used in applications such as modeling legal reasoning and for guiding searches in complex manufacturing and transportation planning problems.

The *k*-NEAREST NEIGHBOR algorithm is one of the most thoroughly analyzed algorithms in machine learning, due in part to its age and in part to its simplicity. Cover and Hart (1967) present early theoretical results, and Duda and Hart (1973) provide a good overview. Bishop (1995) provides a discussion of *k*-NEAREST NEIGHBOR and its relation to estimating probability densities. An excellent current survey of methods for locally weighted regression is given by Atkeson et al. (1997). The application of these methods to robot control is surveyed by Atkeson et al. (1997b).

A thorough discussion of radial basis functions is provided by Bishop (1995). Other treatments are given by Powell (1987) and Poggio and Girosi (1990). See Section 6.12 of this book for a discussion of the EM algorithm and its application to selecting the means of a mixture of Gaussians.

Kolodner (1993) provides a general introduction to case-based reasoning. Other general surveys and collections describing recent research are given by Aamodt et al. (1994), Aha et al. (1991), Haton et al. (1995), Riesbeck and Schank (1989), Schank et al. (1994), Veloso and Aamodt (1995), Watson (1995), and Wess et al. (1994).

EXERCISES

- 8.1. Derive the gradient descent rule for a distance-weighted local linear approximation to the target function, given by Equation (8.1).
- 8.2. Consider the following alternative method for accounting for distance in weighted local regression. Create a virtual set of training examples D' as follows: For each training example $(x, f(x))$ in the original data set D , create some (possibly fractional) number of copies of $(x, f(x))$ in D' , where the number of copies is $K(d(x_q, x))$. Now train a linear approximation to minimize the error criterion

$$E_4 \equiv \frac{1}{2} \sum_{x \in D'} (f(x) - \hat{f}(x))^2$$

The idea here is to make more copies of training examples that are near the query instance, and fewer of those that are distant. Derive the gradient descent rule for this criterion. Express the rule in the form of a sum over members of D rather than D' , and compare it with the rules given by Equations (8.6) and (8.7).

- 8.3. Suggest a lazy version of the eager decision tree learning algorithm ID3 (see Chapter 3). What are the advantages and disadvantages of your lazy algorithm compared to the original eager algorithm?

REFERENCES

- Aamodt, A., & Plazas, E. (1994). Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI Communications*, 7(1), 39–52.
- Aha, D., & Kibler, D. (1989). Noise-tolerant instance-based learning algorithms. *Proceedings of the IJCAI-89* (794–799).
- Aha, D., Kibler, D., & Albert, M. (1991). Instance-based learning algorithms. *Machine Learning*, 6, 37–66.
- Ashley, K. D. (1990). *Modeling legal argument: Reasoning with cases and hypotheticals*. Cambridge, MA: MIT Press.
- Atkeson, C. G., Schaal, S. A., & Moore, A. W. (1997a). Locally weighted learning. *AI Review*, (to appear).
- Atkeson, C. G., Moore, A. W., & Schaal, S. A. (1997b). Locally weighted learning for control. *AI Review*, (to appear).
- Bareiss, E. R., Porter, B., & Weir, C. C. (1988). PROTOS: An exemplar-based learning apprentice. *International Journal of Man-Machine Studies*, 29, 549–561.
- Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9), 509–517.

- Bishop, C. M. (1995). *Neural networks for pattern recognition*. Oxford, England: Oxford University Press.
- Bisio, R., & Malabocchia, F. (1995). Cost estimation of software projects through case-based reasoning. In M. Veloso and A. Aamodt (Eds.), *Lecture Notes in Artificial Intelligence* (pp. 11–22). Berlin: Springer-Verlag.
- Broomhead, D. S., & Lowe, D. (1988). Multivariable functional interpolation and adaptive networks. *Complex Systems*, 2, 321–355.
- Cover, T., & Hart, P. (1967). Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13, 21–27.
- Duda, R., & Hart, P. (1973). *Pattern classification and scene analysis*. New York: John Wiley & Sons.
- Franke, R. (1982). Scattered data interpolation: Tests of some methods. *Mathematics of Computation*, 38, 181–200.
- Friedman, J., Bentley, J., & Finkel, R. (1977). An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3), 209–226.
- Golding, A., & Rosenbloom, P. (1991). Improving rule-based systems through case-based reasoning. *Proceedings of the Ninth National Conference on Artificial Intelligence* (pp. 22–27). Cambridge: AAAI Press/The MIT Press.
- Hartman, E. J., Keller, J. D., & Kowalski, J. M. (1990). Layered neural networks with Gaussian hidden units as universal approximations. *Neural Computation*, 2(2), 210–215.
- Haton, J.-P., Keane, M., & Manago, M. (Eds.). (1995). *Advances in case-based reasoning: Second European workshop*. Berlin: Springer-Verlag.
- Kolodner, J. L. (1993). *Case-Based Reasoning*. San Francisco: Morgan Kaufmann.
- Moody, J. E., & Darken, C. J. (1989). Fast learning in networks of locally-tuned processing units. *Neural Computation*, 1(2), 281–294.
- Moore, A. W., & Lee, M. S. (1994). Efficient algorithms for minimizing cross validation error. *Proceedings of the 11th International Conference on Machine Learning*. San Francisco: Morgan Kaufmann.
- Poggio, T., & Girosi, F. (1990). Networks for approximation and learning. *Proceedings of the IEEE*, 78(9), 1481–1497.
- Powell, M. J. D. (1987). Radial basis functions for multivariable interpolation: A review. In Mason, J., & Cox, M. (Eds.), *Algorithms for approximation* (pp. 143–167). Oxford: Clarendon Press.
- Riesbeck, C., & Schank, R. (1989). *Inside case-based reasoning*. Hillsdale, NJ: Lawrence Erlbaum.
- Schank, R. (1982). *Dynamic Memory*. Cambridge, England: Cambridge University Press.
- Schank, R., Riesbeck, C., & Kass, A. (1994). *Inside case-based explanation*. Hillsdale, NJ: Lawrence Erlbaum.
- Shepard, D. (1968). A two-dimensional interpolation function for irregularly spaced data. *Proceedings of the 23rd National Conference of the ACM* (pp. 517–523).
- Stanfill, C., & Waltz, D. (1986). Toward memory-based reasoning. *Communications of the ACM*, 29(12), 1213–1228.
- Sycara, K., Guttal, R., Koning, J., Narasimhan, S., & Navinchandra, D. (1992). CADET: A case-based synthesis tool for engineering design. *International Journal of Expert Systems*, 4(2), 157–188.
- Veloso, M. M. (1992). *Planning and learning by analogical reasoning*. Berlin: Springer-Verlag.
- Veloso, M. M., & Aamodt, A. (Eds.). (1995). *Case-based reasoning research and development*. Lecture Notes in Artificial Intelligence. Berlin: Springer-Verlag.
- Watson, I. (Ed.). (1995). *Progress in case-based reasoning: First United Kingdom workshop*. Berlin: Springer-Verlag.
- Wess, S., Althoff, K., & Richter, M. (Eds.). (1994). *Topics in case-based reasoning*. Berlin: Springer-Verlag.

CHAPTER

9

GENETIC ALGORITHMS

Genetic algorithms provide an approach to learning that is based loosely on simulated evolution. Hypotheses are often described by bit strings whose interpretation depends on the application, though hypotheses may also be described by symbolic expressions or even computer programs. The search for an appropriate hypothesis begins with a population, or collection, of initial hypotheses. Members of the current population give rise to the next generation population by means of operations such as random mutation and crossover, which are patterned after processes in biological evolution. At each step, the hypotheses in the current population are evaluated relative to a given measure of fitness, with the most fit hypotheses selected probabilistically as seeds for producing the next generation. Genetic algorithms have been applied successfully to a variety of learning tasks and to other optimization problems. For example, they have been used to learn collections of rules for robot control and to optimize the topology and learning parameters for artificial neural networks. This chapter covers both genetic algorithms, in which hypotheses are typically described by bit strings, and genetic programming, in which hypotheses are described by computer programs.

9.1 MOTIVATION

Genetic algorithms (GAs) provide a learning method motivated by an analogy to biological evolution. Rather than search from general-to-specific hypotheses, or from simple-to-complex, GAs generate successor hypotheses by repeatedly mutating and recombining parts of the best currently known hypotheses. At each step,

a collection of hypotheses called the *current population* is updated by replacing some fraction of the population by offspring of the most fit current hypotheses. The process forms a generate-and-test beam-search of hypotheses, in which variants of the best current hypotheses are most likely to be considered next. The popularity of GAs is motivated by a number of factors including:

- Evolution is known to be a successful, robust method for adaptation within biological systems.
- GAs can search spaces of hypotheses containing complex interacting parts, where the impact of each part on overall hypothesis fitness may be difficult to model.
- Genetic algorithms are easily parallelized and can take advantage of the decreasing costs of powerful computer hardware.

This chapter describes the genetic algorithm approach, illustrates its use, and examines the nature of its hypothesis space search. We also describe a variant called genetic programming, in which entire computer programs are evolved to certain fitness criteria. Genetic algorithms and genetic programming are two of the more popular approaches in a field that is sometimes called evolutionary computation. In the final section we touch on selected topics in the study of biological evolution, including the Baldwin effect, which describes an interesting interplay between the learning capabilities of single individuals and the rate of evolution of the entire population.

9.2 GENETIC ALGORITHMS

The problem addressed by GAs is to search a space of candidate hypotheses to identify the best hypothesis. In GAs the “best hypothesis” is defined as the one that optimizes a predefined numerical measure for the problem at hand, called the hypothesis *fitness*. For example, if the learning task is the problem of approximating an unknown function given training examples of its input and output, then fitness could be defined as the accuracy of the hypothesis over this training data. If the task is to learn a strategy for playing chess, fitness could be defined as the number of games won by the individual when playing against other individuals in the current population.

Although different implementations of genetic algorithms vary in their details, they typically share the following structure: The algorithm operates by iteratively updating a pool of hypotheses, called the population. On each iteration, all members of the population are evaluated according to the fitness function. A new population is then generated by probabilistically selecting the most fit individuals from the current population. Some of these selected individuals are carried forward into the next generation population intact. Others are used as the basis for creating new offspring individuals by applying genetic operations such as crossover and mutation.

GA(*Fitness*, *Fitness_threshold*, *p*, *r*, *m*)

Fitness: A function that assigns an evaluation score, given a hypothesis.

Fitness_threshold: A threshold specifying the termination criterion.

p: The number of hypotheses to be included in the population.

r: The fraction of the population to be replaced by Crossover at each step.

m: The mutation rate.

- *Initialize population*: $P \leftarrow$ Generate *p* hypotheses at random

- *Evaluate*: For each h in P , compute $\text{Fitness}(h)$

- While $[\max_h \text{Fitness}(h)] < \text{Fitness_threshold}$ do

Create a new generation, P_S :

1. *Select*: Probabilistically select $(1 - r)p$ members of P to add to P_S . The probability $\Pr(h_i)$ of selecting hypothesis h_i from P is given by

$$\Pr(h_i) = \frac{\text{Fitness}(h_i)}{\sum_{j=1}^p \text{Fitness}(h_j)}$$

2. *Crossover*: Probabilistically select $\frac{r \cdot p}{2}$ pairs of hypotheses from P , according to $\Pr(h_i)$ given above. For each pair, $\langle h_1, h_2 \rangle$, produce two offspring by applying the Crossover operator. Add all offspring to P_S .

3. *Mutate*: Choose *m* percent of the members of P_S with uniform probability. For each, invert one randomly selected bit in its representation.

4. *Update*: $P \leftarrow P_S$.

5. *Evaluate*: for each h in P , compute $\text{Fitness}(h)$

- Return the hypothesis from P that has the highest fitness.
-

TABLE 9.1

A prototypical genetic algorithm. A population containing *p* hypotheses is maintained. On each iteration, the successor population P_S is formed by probabilistically selecting current hypotheses according to their fitness and by adding new hypotheses. New hypotheses are created by applying a crossover operator to pairs of most fit hypotheses and by creating single point mutations in the resulting generation of hypotheses. This process is iterated until sufficiently fit hypotheses are discovered. Typical crossover and mutation operators are defined in a subsequent table.

A prototypical genetic algorithm is described in Table 9.1. The inputs to this algorithm include the fitness function for ranking candidate hypotheses, a threshold defining an acceptable level of fitness for terminating the algorithm, the size of the population to be maintained, and parameters that determine how successor populations are to be generated: the fraction of the population to be replaced at each generation and the mutation rate.

Notice in this algorithm each iteration through the main loop produces a new generation of hypotheses based on the current population. First, a certain number of hypotheses from the current population are selected for inclusion in the next generation. These are selected *probabilistically*, where the probability of selecting hypothesis h_i is given by

$$\Pr(h_i) = \frac{\text{Fitness}(h_i)}{\sum_{j=1}^p \text{Fitness}(h_j)} \quad (9.1)$$

Thus, the probability that a hypothesis will be selected is proportional to its own fitness and is inversely proportional to the fitness of the other competing hypotheses in the current population.

Once these members of the current generation have been selected for inclusion in the next generation population, additional members are generated using a crossover operation. Crossover, defined in detail in the next section, takes two parent hypotheses from the current generation and creates two offspring hypotheses by recombining portions of both parents. The parent hypotheses are chosen probabilistically from the current population, again using the probability function given by Equation (9.1). After new members have been created by this crossover operation, the new generation population now contains the desired number of members. At this point, a certain fraction m of these members are chosen at random, and random mutations all performed to alter these members.

This GA algorithm thus performs a randomized, parallel beam search for hypotheses that perform well according to the fitness function. In the following subsections, we describe in more detail the representation of hypotheses and genetic operators used in this algorithm.

9.2.1 Representing Hypotheses

Hypotheses in GAs are often represented by bit strings, so that they can be easily manipulated by genetic operators such as mutation and crossover. The hypotheses represented by these bit strings can be quite complex. For example, sets of if-then rules can easily be represented in this way, by choosing an encoding of rules that allocates specific substrings for each rule precondition and postcondition. Examples of such rule representations in GA systems are described by Holland (1986); Grefenstette (1988); and DeJong et al. (1993).

To see how if-then rules can be encoded by bit strings, first consider how we might use a bit string to describe a constraint on the value of a single attribute. To pick an example, consider the attribute *Outlook*, which can take on any of the three values *Sunny*, *Overcast*, or *Rain*. One obvious way to represent a constraint on *Outlook* is to use a bit string of length three, in which each bit position corresponds to one of its three possible values. Placing a 1 in some position indicates that the attribute is allowed to take on the corresponding value. For example, the string 010 represents the constraint that *Outlook* must take on the second of these values, or *Outlook* = *Overcast*. Similarly, the string 011 represents the more general constraint that allows two possible values, or (*Outlook* = *Overcast* \vee *Rain*). Note 111 represents the most general possible constraint, indicating that we don't care which of its possible values the attribute takes on.

Given this method for representing constraints on a single attribute, conjunctions of constraints on multiple attributes can easily be represented by concatenating the corresponding bit strings. For example, consider a second attribute, *Wind*, that can take on the value *Strong* or *Weak*. A rule precondition such as

$$(\text{Outlook} = \text{Overcast} \vee \text{Rain}) \wedge (\text{Wind} = \text{Strong})$$

can then be represented by the following bit string of length five:

<i>Outlook</i>	<i>Wind</i>
011	10

Rule postconditions (such as *PlayTennis* = *yes*) can be represented in a similar fashion. Thus, an entire rule can be described by concatenating the bit strings describing the rule preconditions, together with the bit string describing the rule postcondition. For example, the rule

IF *Wind* = *Strong* THEN *PlayTennis* = *yes*

would be represented by the string

<i>Outlook</i>	<i>Wind</i>	<i>PlayTennis</i>
111	10	10

where the first three bits describe the “don’t care” constraint on *Outlook*, the next two bits describe the constraint on *Wind*, and the final two bits describe the rule postcondition (here we assume *PlayTennis* can take on the values *Yes* or *No*). Note the bit string representing the rule contains a substring for each attribute in the hypothesis space, even if that attribute is not constrained by the rule preconditions. This yields a fixed length bit-string representation for rules, in which substrings at specific locations describe constraints on specific attributes. Given this representation for single rules, we can represent sets of rules by similarly concatenating the bit string representations of the individual rules.

In designing a bit string encoding for some hypothesis space, it is useful to arrange for every syntactically legal bit string to represent a well-defined hypothesis. To illustrate, note in the rule encoding in the above paragraph the bit string 111 10 11 represents a rule whose postcondition does not constrain the target attribute *PlayTennis*. If we wish to avoid considering this hypothesis, we may employ a different encoding (e.g., allocate just one bit to the *PlayTennis* postcondition to indicate whether the value is *Yes* or *No*), alter the genetic operators so that they explicitly avoid constructing such bit strings, or simply assign a very low fitness to such bit strings.

In some GAs, hypotheses are represented by symbolic descriptions rather than bit strings. For example, in Section 9.5 we discuss a genetic algorithm that encodes hypotheses as computer programs.

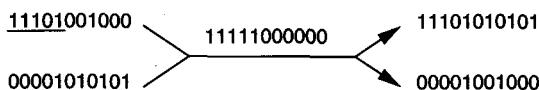
9.2.2 Genetic Operators

The generation of successors in a GA is determined by a set of operators that recombine and mutate selected members of the current population. Typical GA operators for manipulating bit string hypotheses are illustrated in Table 9.1. These operators correspond to idealized versions of the genetic operations found in biological evolution. The two most common operators are *crossover* and *mutation*.

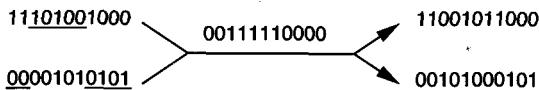
The *crossover operator* produces two new offspring from two parent strings, by copying selected bits from each parent. The bit at position i in each offspring is copied from the bit at position i in one of the two parents. The choice of which parent contributes the bit for position i is determined by an additional string called the *crossover mask*. To illustrate, consider the *single-point crossover* operator at the top of Table 9.2. Consider the topmost of the two offspring in this case. This offspring takes its first five bits from the first parent and its remaining six bits from the second parent, because the crossover mask 11111000000 specifies these choices for each of the bit positions. The second offspring uses the same crossover mask, but switches the roles of the two parents. Therefore, it contains the bits that were not used by the first offspring. In single-point crossover, the crossover mask is always constructed so that it begins with a string containing n contiguous 1s, followed by the necessary number of 0s to complete the string. This results in offspring in which the first n bits are contributed by one parent and the remaining bits by the second parent. Each time the single-point crossover operator is applied,

Initial strings	Crossover Mask	Offspring
-----------------	----------------	-----------

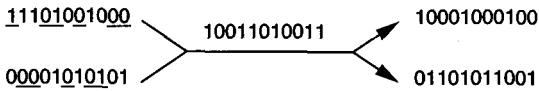
Single-point crossover:



Two-point crossover:



Uniform crossover:



Point mutation:



TABLE 9.2

Common operators for genetic algorithms. These operators form offspring of hypotheses represented by bit strings. The crossover operators create two descendants from two parents, using the crossover mask to determine which parent contributes which bits. Mutation creates a single descendant from a single parent by changing the value of a randomly chosen bit.

the crossover point n is chosen at random, and the crossover mask is then created and applied.

In *two-point crossover*, offspring are created by substituting intermediate segments of one parent into the middle of the second parent string. Put another way, the crossover mask is a string beginning with n_0 zeros, followed by a contiguous string of n_1 ones, followed by the necessary number of zeros to complete the string. Each time the two-point crossover operator is applied, a mask is generated by randomly choosing the integers n_0 and n_1 . For instance, in the example shown in Table 9.2 the offspring are created using a mask for which $n_0 = 2$ and $n_1 = 5$. Again, the two offspring are created by switching the roles played by the two parents.

Uniform crossover combines bits sampled uniformly from the two parents, as illustrated in Table 9.2. In this case the crossover mask is generated as a random bit string with each bit chosen at random and independent of the others.

In addition to recombination operators that produce offspring by combining parts of two parents, a second type of operator produces offspring from a single parent. In particular, the *mutation* operator produces small random changes to the bit string by choosing a single bit at random, then changing its value. Mutation is often performed after crossover has been applied as in our prototypical algorithm from Table 9.1.

Some GA systems employ additional operators, especially operators that are specialized to the particular hypothesis representation used by the system. For example, Grefenstette et al. (1991) describe a system that learns sets of rules for robot control. It uses mutation and crossover, together with an operator for specializing rules. Janikow (1993) describes a system that learns sets of rules using operators that generalize and specialize rules in a variety of directed ways (e.g., by explicitly replacing the condition on an attribute by “don’t care”).

9.2.3 Fitness Function and Selection

The fitness function defines the criterion for ranking potential hypotheses and for probabilistically selecting them for inclusion in the next generation population. If the task is to learn classification rules, then the fitness function typically has a component that scores the classification accuracy of the rule over a set of provided training examples. Often other criteria may be included as well, such as the complexity or generality of the rule. More generally, when the bit-string hypothesis is interpreted as a complex procedure (e.g., when the bit string represents a collection of if-then rules that will be chained together to control a robotic device), the fitness function may measure the overall performance of the resulting procedure rather than performance of individual rules.

In our prototypical GA shown in Table 9.1, the probability that a hypothesis will be selected is given by the ratio of its fitness to the fitness of other members of the current population as seen in Equation (9.1). This method is sometimes called *fitness proportionate selection*, or roulette wheel selection. Other methods for using fitness to select hypotheses have also been proposed. For example, in

tournament selection, two hypotheses are first chosen at random from the current population. With some predefined probability p the more fit of these two is then selected, and with probability $(1 - p)$ the less fit hypothesis is selected. Tournament selection often yields a more diverse population than fitness proportionate selection (Goldberg and Deb 1991). In another method called *rank selection*, the hypotheses in the current population are first sorted by fitness. The probability that a hypothesis will be selected is then proportional to its rank in this sorted list, rather than its fitness.

9.3 AN ILLUSTRATIVE EXAMPLE

A genetic algorithm can be viewed as a general optimization method that searches a large space of candidate objects seeking one that performs best according to the fitness function. Although not guaranteed to find an optimal object, GAs often succeed in finding an object with high fitness. GAs have been applied to a number of optimization problems outside machine learning, including problems such as circuit layout and job-shop scheduling. Within machine learning, they have been applied both to function-approximation problems and to tasks such as choosing the network topology for artificial neural network learning systems.

To illustrate the use of GAs for concept learning, we briefly summarize the GABIL system described by DeJong et al. (1993). GABIL uses a GA to learn boolean concepts represented by a disjunctive set of propositional rules. In experiments over several concept learning problems, GABIL was found to be roughly comparable in generalization accuracy to other learning algorithms such as the decision tree learning algorithm C4.5 and the rule learning system AQ14. The learning tasks in this study included both artificial learning tasks designed to explore the systems' generalization accuracy and the real world problem of breast cancer diagnosis.

The algorithm used by GABIL is exactly the algorithm described in Table 9.1. In experiments reported by DeJong et al. (1993), the parameter r , which determines the fraction of the parent population replaced by crossover, was set to 0.6. The parameter m , which determines the mutation rate, was set to 0.001. These are typical settings for these parameters. The population size p was varied from 100 to 1000, depending on the specific learning task.

The specific instantiation of the GA algorithm in GABIL can be summarized as follows:

- **Representation.** Each hypothesis in GABIL corresponds to a disjunctive set of propositional rules, encoded as described in Section 9.2.1. In particular, the hypothesis space of rule preconditions consists of a conjunction of constraints on a fixed set of attributes, as described in that earlier section. To represent a set of rules, the bit-string representations of individual rules are concatenated. To illustrate, consider a hypothesis space in which rule preconditions are conjunctions of constraints over two boolean attributes, a_1 and a_2 . The rule postcondition is described by a single bit that indicates the predicted

value of the target attribute c . Thus, the hypothesis consisting of the two rules

IF $a_1 = T \wedge a_2 = F$ THEN $c = T$; IF $a_2 = T$ THEN $c = F$

would be represented by the string

$$\begin{array}{ccc} a_1 & a_2 & c \\ 10 & 01 & 1 \end{array} \quad \begin{array}{ccc} a_1 & a_2 & c \\ 11 & 10 & 0 \end{array}$$

Note the length of the bit string grows with the number of rules in the hypothesis. This variable bit-string length requires a slight modification to the crossover operator, as described below.

- **Genetic operators.** GABIL uses the standard mutation operator of Table 9.2, in which a single bit is chosen at random and replaced by its complement. The crossover operator that it uses is a fairly standard extension to the two-point crossover operator described in Table 9.2. In particular, to accommodate the variable-length bit strings that encode rule sets, and to constrain the system so that crossover occurs only between like sections of the bit strings that encode rules, the following approach is taken. To perform a crossover operation on two parents, two crossover points are first chosen at random in the first parent string. Let d_1 (d_2) denote the distance from the leftmost (rightmost) of these two crossover points to the rule boundary immediately to its left. The crossover points in the second parent are now randomly chosen, subject to the constraint that they must have the same d_1 and d_2 value. For example, if the two parent strings are

$$h_1 : \begin{array}{ccc} a_1 & a_2 & c \\ 10 & 01 & 1 \end{array} \quad \begin{array}{ccc} a_1 & a_2 & c \\ 11 & 10 & 0 \end{array}$$

and

$$h_2 : \begin{array}{ccc} a_1 & a_2 & c \\ 01 & 11 & 0 \end{array} \quad \begin{array}{ccc} a_1 & a_2 & c \\ 10 & 01 & 0 \end{array}$$

and the crossover points chosen for the first parent are the points following bit positions 1 and 8,

$$h_1 : \begin{array}{ccc} a_1 & a_2 & c \\ 1[0 & 01 & 1 \end{array} \quad \begin{array}{ccc} a_1 & a_2 & c \\ 11 & 1]0 & 0 \end{array}$$

where “[” and “]” indicate crossover points, then $d_1 = 1$ and $d_2 = 3$. Hence the allowed pairs of crossover points for the second parent include the pairs of bit positions $\langle 1, 3 \rangle$, $\langle 1, 8 \rangle$, and $\langle 6, 8 \rangle$. If the pair $\langle 1, 3 \rangle$ happens to be chosen,

$$h_2 : \begin{array}{ccc} a_1 & a_2 & c \\ 0[1 & 1]1 & 0 \end{array} \quad \begin{array}{ccc} a_1 & a_2 & c \\ 10 & 01 & 0 \end{array}$$

then the two resulting offspring will be

$$h_3 : \begin{array}{ccc} a_1 & a_2 & c \\ 11 & 10 & 0 \end{array}$$

and

$$h_4 : \begin{array}{ccc} a_1 & a_2 & c \\ 00 & 01 & 1 \end{array} \quad \begin{array}{ccc} a_1 & a_2 & c \\ 11 & 11 & 0 \end{array} \quad \begin{array}{ccc} a_1 & a_2 & c \\ 10 & 01 & 0 \end{array}$$

As this example illustrates, this crossover operation enables offspring to contain a different number of rules than their parents, while assuring that all bit strings generated in this fashion represent well-defined rule sets.

- **Fitness function.** The fitness of each hypothesized rule set is based on its classification accuracy over the training data. In particular, the function used to measure fitness is

$$Fitness(h) = (correct(h))^2$$

where $correct(h)$ is the percent of all training examples correctly classified by hypothesis h .

In experiments comparing the behavior of GABIL to decision tree learning algorithms such as C4.5 and ID5R, and to the rule learning algorithm AQ14, DeJong et al. (1993) report roughly comparable performance among these systems, tested on a variety of learning problems. For example, over a set of 12 synthetic problems, GABIL achieved an average generalization accuracy of 92.1 %, whereas the performance of the other systems ranged from 91.2 % to 96.6 %.

9.3.1 Extensions

DeJong et al. (1993) also explore two interesting extensions to the basic design of GABIL. In one set of experiments they explored the addition of two new genetic operators that were motivated by the generalization operators common in many symbolic learning methods. The first of these operators, *AddAlternative*, generalizes the constraint on a specific attribute by changing a 0 to a 1 in the substring corresponding to the attribute. For example, if the constraint on an attribute is represented by the string 10010, this operator might change it to 10110. This operator was applied with probability .01 to selected members of the population on each generation. The second operator, *DropCondition* performs a more drastic generalization step, by replacing all bits for a particular attribute by a 1. This operator corresponds to generalizing the rule by completely dropping the constraint on the attribute, and was applied on each generation with probability .60. The authors report this revised system achieved an average performance of 95.2% over the above set of synthetic learning tasks, compared to 92.1% for the basic GA algorithm.

In the above experiment, the two new operators were applied with the same probability to each hypothesis in the population on each generation. In a second experiment, the bit-string representation for hypotheses was extended to include two bits that determine which of these operators may be applied to the hypothesis. In this extended representation, the bit string for a typical rule set hypothesis would be

a_1	a_2	c	a_1	a_2	c	AA	DC
01	11	0	10	01	0	1	0

where the final two bits indicate in this case that the *Add Alternative* operator may be applied to this bit string, but that the *Drop Condition* operator may not. These two new bits define part of the search strategy used by the GA and are themselves altered and evolved using the same crossover and mutation operators that operate on other bits in the string. While the authors report mixed results with this approach (i.e., improved performance on some problems, decreased performance on others), it provides an interesting illustration of how GAs might in principle be used to evolve their own hypothesis search methods.

9.4 HYPOTHESIS SPACE SEARCH

As illustrated above, GAs employ a randomized beam search method to seek a maximally fit hypothesis. This search is quite different from that of other learning methods we have considered in this book. To contrast the hypothesis space search of GAs with that of neural network BACKPROPAGATION, for example, the gradient descent search in BACKPROPAGATION moves smoothly from one hypothesis to a new hypothesis that is very similar. In contrast, the GA search can move much more abruptly, replacing a parent hypothesis by an offspring that may be radically different from the parent. Note the GA search is therefore less likely to fall into the same kind of local minima that can plague gradient descent methods.

One practical difficulty in some GA applications is the problem of *crowding*. Crowding is a phenomenon in which some individual that is more highly fit than others in the population quickly reproduces, so that copies of this individual and very similar individuals take over a large fraction of the population. The negative impact of crowding is that it reduces the diversity of the population, thereby slowing further progress by the GA. Several strategies have been explored for reducing crowding. One approach is to alter the selection function, using criteria such as tournament selection or rank selection in place of fitness proportionate roulette wheel selection. A related strategy is “fitness sharing,” in which the measured fitness of an individual is reduced by the presence of other, similar individuals in the population. A third approach is to restrict the kinds of individuals allowed to recombine to form offspring. For example, by allowing only the most similar individuals to recombine, we can encourage the formation of clusters of similar individuals, or multiple “subspecies” within the population. A related approach is to spatially distribute individuals and allow only nearby individuals to recombine. Many of these techniques are inspired by the analogy to biological evolution.

9.4.1 Population Evolution and the Schema Theorem

It is interesting to ask whether one can mathematically characterize the evolution over time of the population within a GA. The schema theorem of Holland (1975) provides one such characterization. It is based on the concept of *schemas*, or patterns that describe sets of bit strings. To be precise, a schema is any string composed of 0s, 1s, and *'s. Each schema represents the set of bit strings containing the indicated 0s and 1s, with each "*" interpreted as a "don't care." For example, the schema 0*10 represents the set of bit strings that includes exactly 0010 and 0110.

An individual bit string can be viewed as a representative of each of the different schemas that it matches. For example, the bit string 0010 can be thought of as a representative of 2^4 distinct schemas including 00**, 0*10, ****, etc. Similarly, a population of bit strings can be viewed in terms of the set of schemas that it represents and the number of individuals associated with each of these schema.

The schema theorem characterizes the evolution of the population within a GA in terms of the number of instances representing each schema. Let $m(s, t)$ denote the number of instances of schema s in the population at time t (i.e., during the t th generation). The schema theorem describes the expected value of $m(s, t + 1)$ in terms of $m(s, t)$ and other properties of the schema, population, and GA algorithm parameters.

The evolution of the population in the GA depends on the selection step, the recombination step, and the mutation step. Let us start by considering just the effect of the selection step. Let $f(h)$ denote the fitness of the individual bit string h and $\bar{f}(t)$ denote the average fitness of all individuals in the population at time t . Let n be the total number of individuals in the population. Let $h \in s \cap p_t$ indicate that the individual h is both a representative of schema s and a member of the population at time t . Finally, let $\hat{u}(s, t)$ denote the average fitness of instances of schema s in the population at time t .

We are interested in calculating the expected value of $m(s, t + 1)$, which we denote $E[m(s, t + 1)]$. We can calculate $E[m(s, t + 1)]$ using the probability distribution for selection given in Equation (9.1), which can be restated using our current terminology as follows:

$$\begin{aligned}\Pr(h) &= \frac{f(h)}{\sum_{i=1}^n f(h_i)} \\ &= \frac{f(h)}{n \bar{f}(t)}\end{aligned}$$

Now if we select one member for the new population according to this probability distribution, then the probability that we will select a representative of schema s is

$$\begin{aligned}\Pr(h \in s) &= \sum_{h \in s \cap p_t} \frac{f(h)}{n \bar{f}(t)} \\ &= \frac{\hat{u}(s, t)}{n \bar{f}(t)} m(s, t)\end{aligned}\tag{9.2}$$

The second step above follows from the fact that by definition,

$$\hat{u}(s, t) = \frac{\sum_{h \in s \cap p_t} f(h)}{m(s, t)}$$

Equation (9.2) gives the probability that a single hypothesis selected by the GA will be an instance of schema s . Therefore, the expected number of instances of s resulting from the n independent selection steps that create the entire new generation is just n times this probability.

$$E[m(s, t + 1)] = \frac{\hat{u}(s, t)}{\bar{f}(t)} m(s, t) \quad (9.3)$$

Equation (9.3) states that the expected number of instances of schema s at generation $t + 1$ is proportional to the average fitness $\hat{u}(s, t)$ of instances of this schema at time t , and inversely proportional to the average fitness $\bar{f}(t)$ of all members of the population at time t . Thus, we can expect schemas with above average fitness to be represented with increasing frequency on successive generations. If we view the GA as performing a virtual parallel search through the space of possible schemas at the same time it performs its explicit parallel search through the space of individuals, then Equation (9.3) indicates that more fit schemas will grow in influence over time.

While the above analysis considered only the selection step of the GA, the crossover and mutation steps must be considered as well. The schema theorem considers only the possible negative influence of these genetic operators (e.g., random mutation may decrease the number of representatives of s , independent of $\hat{u}(s, t)$), and considers only the case of single-point crossover. The full schema theorem thus provides a lower bound on the expected frequency of schema s , as follows:

$$E[m(s, t + 1)] \geq \frac{\hat{u}(s, t)}{\bar{f}(t)} m(s, t) \left(1 - p_c \frac{d(s)}{l - 1}\right) (1 - p_m)^{o(s)} \quad (9.4)$$

Here, p_c is the probability that the single-point crossover operator will be applied to an arbitrary individual, and p_m is the probability that an arbitrary bit of an arbitrary individual will be mutated by the mutation operator. $o(s)$ is the number of *defined bits* in schema s , where 0 and 1 are defined bits, but * is not. $d(s)$ is the distance between the leftmost and rightmost defined bits in s . Finally, l is the length of the individual bit strings in the population. Notice the leftmost term in Equation (9.4) is identical to the term from Equation (9.3) and describes the effect of the selection step. The middle term describes the effect of the single-point crossover operator—in particular, it describes the probability that an arbitrary individual representing s will still represent s following application of this crossover operator. The rightmost term describes the probability that an arbitrary individual representing schema s will still represent schema s following application of the mutation operator. Note that the effects of single-point crossover and mutation increase with the number of defined bits $o(s)$ in the schema and with the distance $d(s)$ between the defined bits. Thus, the schema theorem can be roughly interpreted as stating that more fit schemas will tend to grow in influence, especially schemas

containing a small number of defined bits (i.e., containing a large number of '*'s), and especially when these defined bits are near one another within the bit string.

The schema theorem is perhaps the most widely cited characterization of population evolution within a GA. One way in which it is incomplete is that it fails to consider the (presumably) positive effects of crossover and mutation. Numerous more recent theoretical analyses have been proposed, including analyses based on Markov chain models and on statistical mechanics models. See, for example, Whitley and Vose (1995) and Mitchell (1996).

9.5 GENETIC PROGRAMMING

Genetic programming (GP) is a form of evolutionary computation in which the individuals in the evolving population are computer programs rather than bit strings. Koza (1992) describes the basic genetic programming approach and presents a broad range of simple programs that can be successfully learned by GP.

9.5.1 Representing Programs

Programs manipulated by a GP are typically represented by trees corresponding to the parse tree of the program. Each function call is represented by a node in the tree, and the arguments to the function are given by its descendant nodes. For example, Figure 9.1 illustrates this tree representation for the function $\sin(x) + \sqrt{x^2 + y}$. To apply genetic programming to a particular domain, the user must define the primitive functions to be considered (e.g., sin, cos, $\sqrt{\cdot}$, $+$, $-$, exponentials), as well as the terminals (e.g., x , y , constants such as 2). The genetic programming algorithm then uses an evolutionary search to explore the vast space of programs that can be described using these primitives.

As in a genetic algorithm, the prototypical genetic programming algorithm maintains a population of individuals (in this case, program trees). On each iteration, it produces a new generation of individuals using selection, crossover, and mutation. The fitness of a given individual program in the population is typically determined by executing the program on a set of training data. Crossover operations are performed by replacing a randomly chosen subtree of one parent

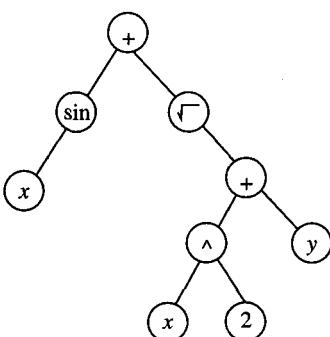
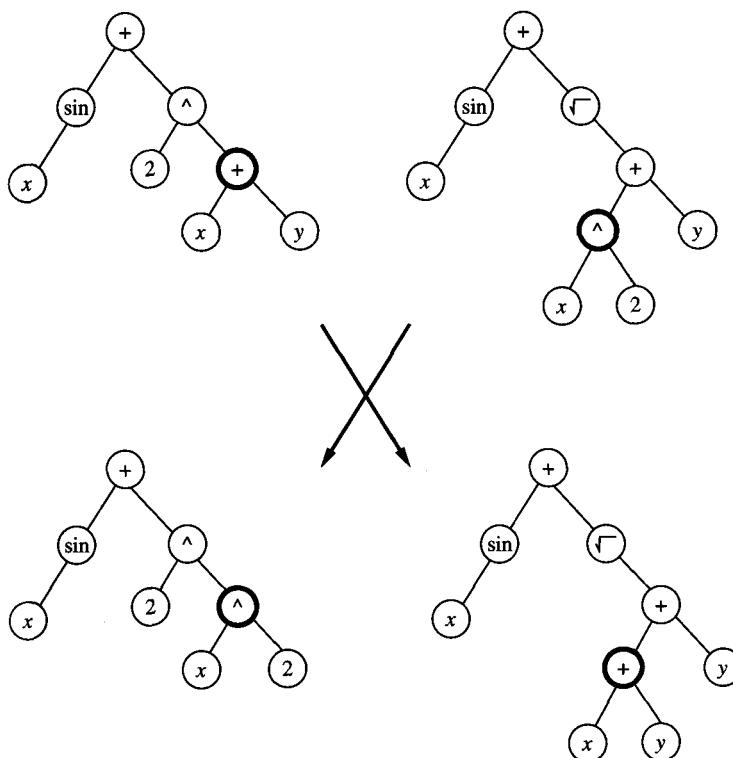


FIGURE 9.1

Program tree representation in genetic programming.
Arbitrary programs are represented by their parse trees.

**FIGURE 9.2**

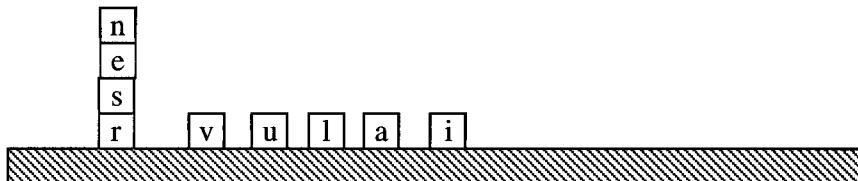
Crossover operation applied to two parent program trees (top). Crossover points (nodes shown in bold at top) are chosen at random. The subtrees rooted at these crossover points are then exchanged to create children trees (bottom).

program by a subtree from the other parent program. Figure 9.2 illustrates a typical crossover operation.

Koza (1992) describes a set of experiments applying a GP to a number of applications. In his experiments, 10% of the current population, selected probabilistically according to fitness, is retained unchanged in the next generation. The remainder of the new generation is created by applying crossover to pairs of programs from the current generation, again selected probabilistically according to their fitness. The mutation operator was not used in this particular set of experiments.

9.5.2 Illustrative Example

One illustrative example presented by Koza (1992) involves learning an algorithm for stacking the blocks shown in Figure 9.3. The task is to develop a general algorithm for stacking the blocks into a single stack that spells the word “universal,”

**FIGURE 9.3**

A block-stacking problem. The task for GP is to discover a program that can transform an arbitrary initial configuration of blocks into a stack that spells the word “universal.” A set of 166 such initial configurations was provided to evaluate fitness of candidate programs (after Koza 1992).

independent of the initial configuration of blocks in the world. The actions available for manipulating blocks allow moving only a single block at a time. In particular, the top block on the stack can be moved to the table surface, or a block on the table surface can be moved to the top of the stack.

As in most GP applications, the choice of problem representation has a significant impact on the ease of solving the problem. In Koza’s formulation, the primitive functions used to compose programs for this task include the following three terminal arguments:

- CS (current stack), which refers to the name of the top block on the stack, or F if there is no current stack.
- TB (top correct block), which refers to the name of the topmost block on the stack, such that it and those blocks beneath it are in the correct order.
- NN (next necessary), which refers to the name of the next block needed above TB in the stack, in order to spell the word “universal,” or F if no more blocks are needed.

As can be seen, this particular choice of terminal arguments provides a natural representation for describing programs for manipulating blocks for this task. Imagine, in contrast, the relative difficulty of the task if we were to instead define the terminal arguments to be the x and y coordinates of each block.

In addition to these terminal arguments, the program language in this application included the following primitive functions:

- (MS x) (move to stack), if block x is on the table, this operator moves x to the top of the stack and returns the value T . Otherwise, it does nothing and returns the value F .
- (MT x) (move to table), if block x is somewhere in the stack, this moves the block at the top of the stack to the table and returns the value T . Otherwise, it returns the value F .
- (EQ x y) (equal), which returns T if x equals y , and returns F otherwise.
- (NOT x), which returns T if $x = F$, and returns F if $x = T$.

- (DU x y) (do until), which executes the expression x repeatedly until expression y returns the value T .

To allow the system to evaluate the fitness of any given program, Koza provided a set of 166 training example problems representing a broad variety of initial block configurations, including problems of differing degrees of difficulty. The fitness of any given program was taken to be the number of these examples solved by the algorithm. The population was initialized to a set of 300 random programs. After 10 generations, the system discovered the following program, which solves all 166 problems.

$$(EQ (DU (MT CS)(NOT CS)) (DU (MS NN)(NOT NN)))$$

Notice this program contains a sequence of two DU, or “Do Until” statements. The first repeatedly moves the current top of the stack onto the table, until the stack becomes empty. The second “Do Until” statement then repeatedly moves the next necessary block from the table onto the stack. The role played by the top level EQ expression here is to provide a syntactically legal way to sequence these two “Do Until” loops.

Somewhat surprisingly, after only a few generations, this GP was able to discover a program that solves all 166 training problems. Of course the ability of the system to accomplish this depends strongly on the primitive arguments and functions provided, and on the set of training example cases used to evaluate fitness.

9.5.3 Remarks on Genetic Programming

As illustrated in the above example, genetic programming extends genetic algorithms to the evolution of complete computer programs. Despite the huge size of the hypothesis space it must search, genetic programming has been demonstrated to produce intriguing results in a number of applications. A comparison of GP to other methods for searching through the space of computer programs, such as hillclimbing and simulated annealing, is given by O'Reilly and Oppacher (1994).

While the above example of GP search is fairly simple, Koza et al. (1996) summarize the use of a GP in several more complex tasks such as designing electronic filter circuits and classifying segments of protein molecules. The filter circuit design problem provides an example of a considerably more complex problem. Here, programs are evolved that transform a simple fixed seed circuit into a final circuit design. The primitive functions used by the GP to construct its programs are functions that edit the seed circuit by inserting or deleting circuit components and wiring connections. The fitness of each program is calculated by simulating the circuit it outputs (using the SPICE circuit simulator) to determine how closely this circuit meets the design specifications for the desired filter. More precisely, the fitness score is the sum of the magnitudes of errors between the desired and actual circuit output at 101 different input frequencies. In this case, a population of size 640,000 was maintained, with selection

producing 10% of the successor population, crossover producing 89%, and mutation producing 1%. The system was executed on a 64-node parallel processor. Within the first randomly generated population, the circuits produced were so unreasonable that the SPICE simulator could not even simulate the behavior of 98% of the circuits. The percentage of unsimulatable circuits dropped to 84.9% following the first generation, to 75.0% following the second generation, and to an average of 9.6% over succeeding generations. The fitness score of the best circuit in the initial population was 159, compared to a score of 39 after 20 generations and a score of 0.8 after 137 generations. The best circuit, produced after 137 generations, exhibited performance very similar to the desired behavior.

In most cases, the performance of genetic programming depends crucially on the choice of representation and on the choice of fitness function. For this reason, an active area of current research is aimed at the automatic discovery and incorporation of subroutines that improve on the original set of primitive functions, thereby allowing the system to dynamically alter the primitives from which it constructs individuals. See, for example, Koza (1994).

9.6 MODELS OF EVOLUTION AND LEARNING

In many natural systems, individual organisms learn to adapt significantly during their lifetime. At the same time, biological and social processes allow their species to adapt over a time frame of many generations. One interesting question regarding evolutionary systems is "What is the relationship between learning during the lifetime of a single individual, and the longer time frame species-level learning afforded by evolution?"

9.6.1 Lamarckian Evolution

Lamarck was a scientist who, in the late nineteenth century, proposed that evolution over many generations was directly influenced by the experiences of individual organisms during their lifetime. In particular, he proposed that experiences of a single organism directly affected the genetic makeup of their offspring: If an individual learned during its lifetime to avoid some toxic food, it could pass this trait on genetically to its offspring, which therefore would not need to learn the trait. This is an attractive conjecture, because it would presumably allow for more efficient evolutionary progress than a generate-and-test process (like that of GAs and GPs) that ignores the experience gained during an individual's lifetime. Despite the attractiveness of this theory, current scientific evidence overwhelmingly contradicts Lamarck's model. The currently accepted view is that the genetic makeup of an individual is, in fact, unaffected by the lifetime experience of one's biological parents. Despite this apparent biological fact, recent computer studies have shown that Lamarckian processes can sometimes improve the effectiveness of computerized genetic algorithms (see Grefenstette 1991; Ackley and Littman 1994; and Hart and Belew 1995).

9.6.2 Baldwin Effect

Although Lamarckian evolution is not an accepted model of biological evolution, other mechanisms have been suggested by which individual learning can alter the course of evolution. One such mechanism is called the Baldwin effect, after J. M. Baldwin (1896), who first suggested the idea. The Baldwin effect is based on the following observations:

- If a species is evolving in a changing environment, there will be evolutionary pressure to favor individuals with the capability to learn during their lifetime. For example, if a new predator appears in the environment, then individuals capable of learning to avoid the predator will be more successful than individuals who cannot learn. In effect, the ability to learn allows an individual to perform a small local search during its lifetime to maximize its fitness. In contrast, nonlearning individuals whose fitness is fully determined by their genetic makeup will operate at a relative disadvantage.
- Those individuals who are able to learn many traits will rely less strongly on their genetic code to “hard-wire” traits. As a result, these individuals can support a more diverse gene pool, relying on individual learning to overcome the “missing” or “not quite optimized” traits in the genetic code. This more diverse gene pool can, in turn, support more rapid evolutionary adaptation. Thus, the ability of individuals to learn can have an indirect accelerating effect on the rate of evolutionary adaptation for the entire population.

To illustrate, imagine some new change in the environment of some species, such as a new predator. Such a change will selectively favor individuals capable of learning to avoid the predator. As the proportion of such self-improving individuals in the population grows, the population will be able to support a more diverse gene pool, allowing evolutionary processes (even non-Lamarckian generate-and-test processes) to adapt more rapidly. This accelerated adaptation may in turn enable standard evolutionary processes to more quickly evolve a genetic (nonlearned) trait to avoid the predator (e.g., an instinctive fear of this animal). Thus, the Baldwin effect provides an indirect mechanism for individual learning to positively impact the rate of evolutionary progress. By increasing survivability and genetic diversity of the species, individual learning supports more rapid evolutionary progress, thereby increasing the chance that the species will evolve genetic, nonlearned traits that better fit the new environment.

There have been several attempts to develop computational models to study the Baldwin effect. For example, Hinton and Nowlan (1987) experimented with evolving a population of simple neural networks, in which some network weights were fixed during the individual network “lifetime,” while others were trainable. The genetic makeup of the individual determined which weights were trainable and which were fixed. In their experiments, when no individual learning

was allowed, the population failed to improve its fitness over time. However, when individual learning was allowed, the population quickly improved its fitness. During early generations of evolution the population contained a greater proportion of individuals with many trainable weights. However, as evolution proceeded, the number of fixed, correct network weights tended to increase, as the population evolved toward genetically given weight values and toward less dependence on individual learning of weights. Additional computational studies of the Baldwin effect have been reported by Belew (1990), Harvey (1993), and French and Messinger (1994). An excellent overview of this topic can be found in Mitchell (1996). A special issue of the journal *Evolutionary Computation* on this topic (Turney et al. 1997) contains several articles on the Baldwin effect.

9.7 PARALLELIZING GENETIC ALGORITHMS

GAs are naturally suited to parallel implementation, and a number of approaches to parallelization have been explored. *Coarse grain* approaches to parallelization subdivide the population into somewhat distinct groups of individuals, called *demes*. Each deme is assigned to a different computational node, and a standard GA search is performed at each node. Communication and cross-fertilization between demes occurs on a less frequent basis than within demes. Transfer between demes occurs by a *migration* process, in which individuals from one deme are copied or transferred to other demes. This process is modeled after the kind of cross-fertilization that might occur between physically separated subpopulations of biological species. One benefit of such approaches is that it reduces the crowding problem often encountered in nonparallel GAs, in which the system falls into a local optimum due to the early appearance of a genotype that comes to dominate the entire population. Examples of coarse-grained parallel GAs are described by Tanese (1989) and by Cohoon et al. (1987).

In contrast to coarse-grained parallel implementations of GAs, fine-grained implementations typically assign one processor per individual in the population. Recombination then takes place among neighboring individuals. Several different types of neighborhoods have been proposed, ranging from planar grid to torus. Examples of such systems are described by Spiessens and Manderick (1991). An edited collection of papers on parallel GAs is available in Stender (1993).

9.8 SUMMARY AND FURTHER READING

The main points of this chapter include:

- Genetic algorithms (GAs) conduct a randomized, parallel, hill-climbing search for hypotheses that optimize a predefined fitness function.
- The search performed by GAs is based on an analogy to biological evolution. A diverse population of competing hypotheses is maintained. At each

iteration, the most fit members of the population are selected to produce new offspring that replace the least fit members of the population. Hypotheses are often encoded by strings that are combined by crossover operations, and subjected to random mutations.

- GAs illustrate how learning can be viewed as a special case of optimization. In particular, the learning task is to find the optimal hypothesis, according to the predefined fitness function. This suggests that other optimization techniques such as simulated annealing can also be applied to machine learning problems.
- GAs have most commonly been applied to optimization problems outside machine learning, such as design optimization problems. When applied to learning tasks, GAs are especially suited to tasks in which hypotheses are complex (e.g., sets of rules for robot control, or computer programs), and in which the objective to be optimized may be an indirect function of the hypothesis (e.g., that the set of acquired rules successfully controls a robot).
- Genetic programming is a variant of genetic algorithms in which the hypotheses being manipulated are computer programs rather than bit strings. Operations such as crossover and mutation are generalized to apply to programs rather than bit strings. Genetic programming has been demonstrated to learn programs for tasks such as simulated robot control (Koza 1992) and recognizing objects in visual scenes (Teller and Veloso 1994).

Evolution-based computational approaches have been explored since the early days of computer science (e.g., Box 1957 and Bledsoe 1961). Several different evolutionary approaches were introduced during the 1960s and have been further explored since that time. Evolution strategies, developed by Rechenberg (1965, 1973) to optimize numerical parameters in engineering design, were followed up by Schwefel (1975, 1977, 1995) and others. Evolutionary programming, developed by Folgel, Owens, and Walsh (1966) as a method for evolving finite-state machines, was followed up by numerous researchers (e.g., Fogel and Atmar 1993). Genetic algorithms, introduced by Holland (1962, 1975) included the notion of maintaining a large population of individuals and emphasized crossover as a key operation in such systems. Genetic programming, introduced by Koza (1992), applies the search strategy of genetic algorithms to hypotheses consisting of computer programs. As computer hardware continues to become faster and less expensive, interest in evolutionary approaches continues to grow.

One approach to using GAs to learn sets of rules was developed by K. DeJong and his students at the University of Pittsburgh (e.g., Smith 1980). In this approach, each rule set is one member in the population of competing hypotheses, as in the GABIL system discussed in this chapter. A somewhat different approach was developed at University of Michigan by Holland and his students (Holland 1986), in which each rule is a member of the population, and

the population itself is the rule set. A biological perspective on the roles of mutation, inbreeding, cross-breeding, and selection in evolution is provided by Wright (1977).

Mitchell (1996) and Goldberg (1989) are two textbooks devoted to the subject of genetic algorithms. Forrest (1993) provides an overview of the technical issues in GAs, and Goldberg (1994) provides an overview of several recent applications. Koza's (1992) monograph on genetic programming is the standard reference for this extension of genetic algorithms to manipulation of computer programs. The primary conference in which new results are published is the *International Conference on Genetic Algorithms*. Other relevant conferences include the *Conference on Simulation of Adaptive Behavior*, the *International Conference on Artificial Neural Networks and Genetic Algorithms*, and the *IEEE International Conference on Evolutionary Computation*. An annual conference is now held on genetic programming, as well (Koza et al. 1996b). The *Evolutionary Computation Journal* is one source of recent research results in the field. Several special issues of the journal *Machine Learning* have also been devoted to GAs.

EXERCISES

- 9.1. Design a genetic algorithm to learn conjunctive classification rules for the *Play-Tennis* problem described in Chapter 3. Describe precisely the bit-string encoding of hypotheses and a set of crossover operators.
- 9.2. Implement a simple GA for Exercise 9.1. Experiment with varying population size p , the fraction r of the population replaced at each generation, and the mutation rate m .
- 9.3. Represent the program discovered by the GP (described in Section 9.5.2) as a tree. Illustrate the operation of the GP crossover operator by applying it using two copies of your tree as the two parents.
- 9.4. Consider applying GAs to the task of finding an appropriate set of weights for an artificial neural network (in particular, a feedforward network identical to those trained by BACKPROPAGATION (Chapter 4)). Consider a $3 \times 2 \times 1$ layered, feedforward network. Describe an encoding of network weights as a bit string, and describe an appropriate set of crossover operators. Hint: Do not allow all possible crossover operations on bit strings. State one advantage and one disadvantage of using GAs in contrast to BACKPROPAGATION to train network weights.

REFERENCES

- Ackley, D., & Littman, M. (1994). A case for Lamarckian evolution. In C. Langton (Ed.), *Artificial life III*. Reading, MA: Addison Wesley.
- Back, T. (1996). *Evolutionary algorithms in theory and practice*. Oxford, England: Oxford University Press.
- Baldwin, J. M. (1896). A new factor in evolution. *American Naturalist*, 3, 441–451, 536–553.
<http://www.santafe.edu/sfi/publications/Bookinfo/baldwin.html>
- Belew, R. (1990). Evolution, learning, and culture: Computational metaphors for adaptive algorithms. *Complex Systems*, 4, 11–49.

- Belew, R. K., & Mitchell, M. (Eds.). (1996). *Adaptive individuals in evolving populations: Models and algorithms*. Reading, MA: Addison-Wesley.
- Bledsoe, W. (1961). The use of biological concepts in the analytical study of systems. *Proceedings of the ORSA-TIMS National Meeting*, San Francisco.
- Booker, L. B., Goldberg, D. E., & Holland, J. H. (1989). Classifier systems and genetic algorithms. *Artificial Intelligence*, 40, 235–282.
- Box, G. (1957). Evolutionary operation: A method for increasing industrial productivity. *Journal of the Royal Statistical Society*, 6(2), 81–101.
- Cohoon, J. P., Hegde, S. U., Martin, W. N., & Richards, D. (1987). Punctuated equilibria: A parallel genetic algorithm. *Proceedings of the Second International Conference on Genetic Algorithms* (pp. 148–154).
- DeJong, K. A. (1975). *An analysis of behavior of a class of genetic adaptive systems* (Ph.D. dissertation). University of Michigan.
- DeJong, K. A., Spears, W. M., & Gordon, D. F. (1993). Using genetic algorithms for concept learning. *Machine Learning*, 13, 161–188.
- Fogel, L. J., Owens, A. J., & Walsh, M. J. (1966). *Artificial intelligence through simulated evolution*. New York: John Wiley & Sons.
- Fogel, L. J., & Atmar, W. (Eds.). (1993). *Proceedings of the Second Annual Conference on Evolutionary Programming*. Evolutionary Programming Society.
- Forrest, S. (1993). Genetic algorithms: Principles of natural selection applied to computation. *Science*, 261, 872–878.
- French, R., & Messinger A. (1994). Genes, phenes, and the Baldwin effect: Learning and evolution in a simulated population. In R. Brooks and P. Maes (Eds.), *Artificial Life IV*. Cambridge, MA: MIT Press.
- Goldberg, D. (1989). *Genetic algorithms in search, optimization, and machine learning*. Reading, MA: Addison-Wesley.
- Goldberg, D. (1994). Genetic and evolutionary algorithms come of age. *Communications of the ACM*, 37(3), 113–119.
- Green, D. P., & Smith, S. F. (1993). Competition based induction of decision models from examples. *Machine Learning*, 13, 229–257.
- Grefenstette, J. J. (1988). Credit assignment in rule discovery systems based on genetic algorithms. *Machine Learning*, 3, 225–245.
- Grefenstette, J. J. (1991). Lamarckian learning in multi-agent environments. In R. Belew and L. Booker (Eds.), *Proceedings of the Fourth International Conference on Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann.
- Hart, W., & Belew, R. (1995). Optimization with genetic algorithm hybrids that use local search. In R. Below and M. Mitchell (Eds.), *Adaptive individuals in evolving populations: Models and algorithms*. Reading, MA: Addison-Wesley.
- Harvey, I. (1993). The puzzle of the persistent question marks: A case study of genetic drift. In Forrest (Ed.), *Proceedings of the Fifth International Conference on Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann.
- Hinton, G. E., & Nowlan, S. J. (1987). How learning can guide evolution. *Complex Systems*, 1, 495–502.
- Holland, J. H. (1962). Outline for a logical theory of adaptive systems. *Journal of the Association for Computing Machinery*, 3, 297–314.
- Holland, J. H. (1975). *Adaptation in natural and artificial systems*. University of Michigan Press (reprinted in 1992 by MIT Press, Cambridge, MA).

- Holland, J. H. (1986). Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In R. Michalski, J. Carbonell, & T. Mitchell (Eds.), *Machine learning: An artificial intelligence approach* (Vol. 2). San Mateo, CA: Morgan Kaufmann.
- Holland, J. H. (1989). Searching nonlinear functions for high values. *Applied Mathematics and Computation*, 32, 255–274.
- Janikow, C. Z. (1993). A knowledge-intensive GA for supervised learning. *Machine Learning*, 13, 189–228.
- Koza, J. (1992). *Genetic programming: On the programming of computers by means of natural selection*. Cambridge, MA: MIT Press.
- Koza, J. R. (1994). *Genetic Programming II: Automatic discovery of reusable programs*. Cambridge, MA: The MIT Press.
- Koza, J. R., Bennett III, F. H., Andre, D., & Keane, M. A. (1996). Four problems for which a computer program evolved by genetic programming is competitive with human performance. *Proceedings of the 1996 IEEE International Conference on Evolutionary Computation* (pp. 1–10). IEEE Press.
- Koza, J. R., Goldberg, D. E., Fogel, D. B., & Riolo, R. L. (Eds.). (1996b). *Genetic programming 1996: Proceedings of the First Annual Conference*. Cambridge, MA: MIT Press.
- Machine Learning: Special Issue on Genetic Algorithms* (1988) 3:2–3, October.
- Machine Learning: Special Issue on Genetic Algorithms* (1990) 5:4, October.
- Machine Learning: Special Issue on Genetic Algorithms* (1993) 13:2,3, November.
- Mitchell, M. (1996). *An introduction to genetic algorithms*. Cambridge, MA: MIT Press.
- O'Reilly, U-M., & Oppacher, R. (1994). Program search with a hierarchical variable length representation: Genetic programming, simulated annealing, and hill climbing. In Y. Davidor et al. (Eds.), *Parallel problem solving from nature—PPSN III* (Vol. 866) (Lecture notes in computer science). Springer-Verlag.
- Rechenberg, I. (1965). *Cybernetic solution path of an experimental problem*. Ministry of aviation, Royal Aircraft Establishment, U.K.
- Rechenberg, I. (1973). *Evolutionsstrategie: Optimierung technischer systeme nach prinzipien der biologischen evolution*. Stuttgart: Frommann-Holzboog.
- Schwefel, H. P. (1975). *Evolutionsstrategie und numerische optimierung* (Ph.D. thesis). Technical University of Berlin.
- Schwefel, H. P. (1977). *Numerische optimierung von computer-modellen mittels der evolutionsstrategie*. Basel: Birkhauser.
- Schwefel, H. P. (1995). *Evolution and optimum seeking*. New York: John Wiley & Sons.
- Spiessens, P., & Manderick, B. (1991). A massively parallel genetic algorithm: Implementation and first analysis. *Proceedings of the 4th International Conference on Genetic Algorithms* (pp. 279–286).
- Smith, S. (1980). *A learning system based on genetic adaptive algorithms* (Ph.D. dissertation). Computer Science, University of Pittsburgh.
- Stender, J. (Ed.) (1993). *Parallel genetic algorithms*. Amsterdam: IOS Publishing.
- Tanese, R. (1989). Distributed genetic algorithms. *Proceedings of the 3rd International Conference on Genetic Algorithms* (pp. 434–439).
- Teller, A., & Veloso, M. (1994). PADO: A new learning architecture for object recognition. In K. Ikeuchi & M. Veloso (Eds.), *Symbolic visual learning* (pp. 81–116). Oxford, England: Oxford Univ. Press.
- Turney, P. D. (1995). Cost-sensitive classification: Empirical evaluation of a hybrid genetic decision tree induction algorithm. *Journal of AI Research*, 2, 369–409. <http://www.cs.washington.edu/research/jair/home.html>.

- Turney, P. D., Whitley, D., & Anderson, R. (1997). *Evolutionary Computation*. Special issue: The Baldwin effect, 4(3). Cambridge, MA: MIT Press. <http://www-mitpress.mit.edu/jrnls-catalog/evolution-abstracts/evol.html>.
- Whitley, L. D., & Vose, M. D. (Eds.). (1995). *Foundations of genetic algorithms 3*. Morgan Kaufmann.
- Wright, S. (1977). *Evolution and the genetics of populations. Vol. 4: Variability within and among Natural Populations*. Chicago: University of Chicago Press.
- Zbigniew, M. (1992). *Genetic algorithms + data structures = evolution programs*. Berlin: Springer-Verlag.

CHAPTER

10

LEARNING SETS OF RULES

One of the most expressive and human readable representations for learned hypotheses is sets of if-then rules. This chapter explores several algorithms for learning such sets of rules. One important special case involves learning sets of rules containing variables, called first-order Horn clauses. Because sets of first-order Horn clauses can be interpreted as programs in the logic programming language PROLOG, learning them is often called inductive logic programming (ILP). This chapter examines several approaches to learning sets of rules, including an approach based on inverting the deductive operators of mechanical theorem provers.

10.1 INTRODUCTION

In many cases it is useful to learn the target function represented as a set of if-then rules that jointly define the function. As shown in Chapter 3, one way to learn sets of rules is to first learn a decision tree, then translate the tree into an equivalent set of rules—one rule for each leaf node in the tree. A second method, illustrated in Chapter 9, is to use a genetic algorithm that encodes each rule set as a bit string and uses genetic search operators to explore this hypothesis space. In this chapter we explore a variety of algorithms that directly learn rule sets and that differ from these algorithms in two key respects. First, they are designed to learn sets of first-order rules that contain variables. This is significant because first-order rules are much more expressive than propositional rules. Second, the algorithms discussed here use sequential covering algorithms that learn one rule at a time to incrementally grow the final set of rules.

As an example of first-order rule sets, consider the following two rules that jointly describe the target concept *Ancestor*. Here we use the predicate *Parent(x, y)* to indicate that *y* is the mother or father of *x*, and the predicate *Ancestor(x, y)* to indicate that *y* is an ancestor of *x* related by an arbitrary number of family generations.

IF <i>Parent(x, y)</i>	THEN <i>Ancestor(x, y)</i>
IF <i>Parent(x, z) \wedge Ancestor(z, y)</i>	THEN <i>Ancestor(x, y)</i>

Note these two rules compactly describe a recursive function that would be very difficult to represent using a decision tree or other propositional representation. One way to see the representational power of first-order rules is to consider the general purpose programming language PROLOG. In PROLOG, programs are sets of first-order rules such as the two shown above (rules of this form are also called *Horn clauses*). In fact, when stated in a slightly different syntax the above rules form a valid PROLOG program for computing the *Ancestor* relation. In this light, a general purpose algorithm capable of learning such rule sets may be viewed as an algorithm for automatically inferring PROLOG programs from examples. In this chapter we explore learning algorithms capable of learning such rules, given appropriate sets of training examples.

In practice, learning systems based on first-order representations have been successfully applied to problems such as learning which chemical bonds fragment in a mass spectrometer (Buchanan 1976; Lindsay 1980), learning which chemical substructures produce mutagenic activity (a property related to carcinogenicity) (Srinivasan et al. 1994), and learning to design finite element meshes to analyze stresses in physical structures (Dolsak and Muggleton 1992). In each of these applications, the hypotheses that must be represented involve relational assertions that can be conveniently expressed using first-order representations, while they are very difficult to describe using propositional representations.

In this chapter we begin by considering algorithms that learn sets of propositional rules; that is, rules without variables. Algorithms for searching the hypothesis space to learn disjunctive sets of rules are most easily understood in this setting. We then consider extensions of these algorithms to learn first-order rules. Two general approaches to inductive logic programming are then considered, and the fundamental relationship between inductive and deductive inference is explored.

10.2 SEQUENTIAL COVERING ALGORITHMS

Here we consider a family of algorithms for learning rule sets based on the strategy of learning one rule, removing the data it covers, then iterating this process. Such algorithms are called *sequential covering* algorithms. To elaborate, imagine we have a subroutine **LEARN-ONE-RULE** that accepts a set of positive and negative training examples as input, then outputs a single rule that covers many of the

positive examples and few of the negative examples. We require that this output rule have high accuracy, but not necessarily high coverage. By high accuracy, we mean the predictions it makes should be correct. By accepting low coverage, we mean it need not make predictions for every training example.

Given this LEARN-ONE-RULE subroutine for learning a single rule, one obvious approach to learning a set of rules is to invoke LEARN-ONE-RULE on all the available training examples, remove any positive examples covered by the rule it learns, then invoke it again to learn a second rule based on the remaining training examples. This procedure can be iterated as many times as desired to learn a disjunctive set of rules that together cover any desired fraction of the positive examples. This is called a *sequential covering* algorithm because it sequentially learns a set of rules that together cover the full set of positive examples. The final set of rules can then be sorted so that more accurate rules will be considered first when a new instance must be classified. A prototypical sequential covering algorithm is described in Table 10.1.

This sequential covering algorithm is one of the most widespread approaches to learning disjunctive sets of rules. It reduces the problem of learning a disjunctive set of rules to a sequence of simpler problems, each requiring that a single conjunctive rule be learned. Because it performs a greedy search, formulating a sequence of rules without backtracking, it is not guaranteed to find the smallest or best set of rules that cover the training examples.

How shall we design LEARN-ONE-RULE to meet the needs of the sequential covering algorithm? We require an algorithm that can formulate a single rule with high accuracy, but that need not cover all of the positive examples. In this section we present a variety of algorithms and describe the main variations that have been explored in the research literature. In this section we consider learning only propositional rules. In later sections, we extend these algorithms to learn first-order Horn clauses.

SEQUENTIAL-COVERING(*Target_attribute*, *Attributes*, *Examples*, *Threshold*)

- *Learned_rules* $\leftarrow \{\}$
- *Rule* \leftarrow LEARN-ONE-RULE(*Target_attribute*, *Attributes*, *Examples*)
- while PERFORMANCE(*Rule*, *Examples*) $>$ *Threshold*, do
 - *Learned_rules* \leftarrow *Learned_rules* + *Rule*
 - *Examples* \leftarrow *Examples* - {examples correctly classified by *Rule*}
 - *Rule* \leftarrow LEARN-ONE-RULE(*Target_attribute*, *Attributes*, *Examples*)
- *Learned_rules* \leftarrow sort *Learned_rules* accord to PERFORMANCE over *Examples*
- return *Learned_rules*

TABLE 10.1

The sequential covering algorithm for learning a disjunctive set of rules. LEARN-ONE-RULE must return a single rule that covers at least some of the *Examples*. PERFORMANCE is a user-provided subroutine to evaluate rule quality. This covering algorithm learns rules until it can no longer learn a rule whose performance is above the given *Threshold*.

10.2.1 General to Specific Beam Search

One effective approach to implementing LEARN-ONE-RULE is to organize the hypothesis space search in the same general fashion as the ID3 algorithm, but to follow only the most promising branch in the tree at each step. As illustrated in the search tree of Figure 10.1, the search begins by considering the most general rule precondition possible (the empty test that matches every instance), then greedily adding the attribute test that most improves rule performance measured over the training examples. Once this test has been added, the process is repeated by greedily adding a second attribute test, and so on. Like ID3, this process grows the hypothesis by greedily adding new attribute tests until the hypothesis reaches an acceptable level of performance. Unlike ID3, this implementation of LEARN-ONE-RULE follows only a single descendant at each search step—the attribute-value pair yielding the best performance—rather than growing a subtree that covers all possible values of the selected attribute.

This approach to implementing LEARN-ONE-RULE performs a general-to-specific search through the space of possible rules in search of a rule with high accuracy, though perhaps incomplete coverage of the data. As in decision tree learning, there are many ways to define a measure to select the “best” descendant. To follow the lead of ID3 let us for now define the best descendant as the one whose covered examples have the lowest entropy (recall Equation [3.3]).

The general-to-specific search suggested above for the LEARN-ONE-RULE algorithm is a greedy depth-first search with no backtracking. As with any greedy

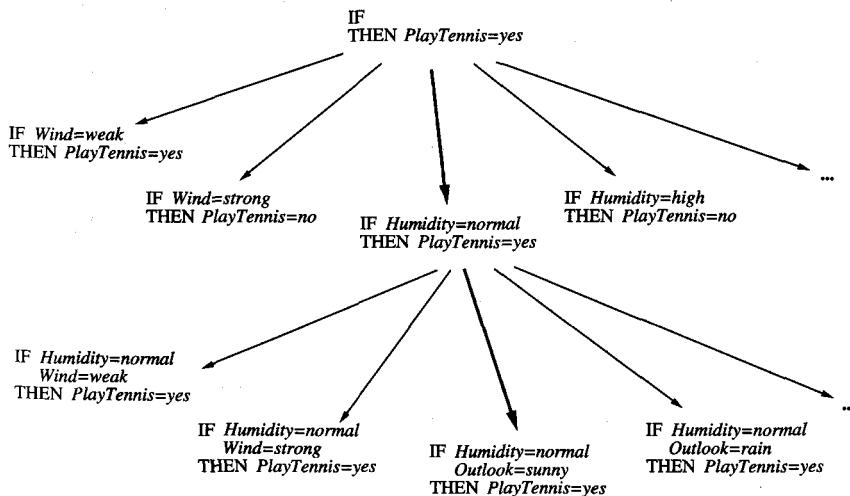


FIGURE 10.1

The search for rule preconditions as LEARN-ONE-RULE proceeds from general to specific. At each step, the preconditions of the best rule are specialized in all possible ways. Rule postconditions are determined by the examples found to satisfy the preconditions. This figure illustrates a beam search of width 1.

search, there is a danger that a suboptimal choice will be made at any step. To reduce this risk, we can extend the algorithm to perform a *beam search*; that is, a search in which the algorithm maintains a list of the k best candidates at each step, rather than a single best candidate. On each search step, descendants (specializations) are generated for each of these k best candidates, and the resulting set is again reduced to the k most promising members. Beam search keeps track of the most promising alternatives to the current top-rated hypothesis, so that all of their successors can be considered at each search step. This general-to-specific beam search algorithm is used by the CN2 program described by Clark and Niblett (1989). The algorithm is described in Table 10.2.

LEARN-ONE-RULE(*Target_attribute*, *Attributes*, *Examples*, k)

Returns a single rule that covers some of the Examples. Conducts a general-to-specific greedy beam search for the best rule, guided by the PERFORMANCE metric.

- Initialize *Best_hypothesis* to the most general hypothesis \emptyset
- Initialize *Candidate_hypotheses* to the set $\{\text{Best_hypothesis}\}$
- While *Candidate_hypotheses* is not empty, Do
 1. Generate the next more specific *Candidate_hypotheses*
 - *All_constraints* \leftarrow the set of all constraints of the form $(a = v)$, where a is a member of *Attributes*, and v is a value of a that occurs in the current set of *Examples*
 - *New_candidate_hypotheses* \leftarrow
 - for each h in *Candidate_hypotheses*,
 - for each c in *All_constraints*,
 - create a specialization of h by adding the constraint c
 - Remove from *New_candidate_hypotheses* any hypotheses that are duplicates, inconsistent, or not maximally specific
 2. Update *Best_hypothesis*
 - For all h in *New_candidate_hypotheses* do
 - If $(\text{PERFORMANCE}(h, \text{Examples}, \text{Target_attribute}) > \text{PERFORMANCE}(\text{Best_hypothesis}, \text{Examples}, \text{Target_attribute}))$
 - Then $\text{Best_hypothesis} \leftarrow h$
 3. Update *Candidate_hypotheses*
 - *Candidate_hypotheses* \leftarrow the k best members of *New_candidate_hypotheses*, according to the PERFORMANCE measure.
- Return a rule of the form
“IF *Best_hypothesis* THEN *prediction*”
where *prediction* is the most frequent value of *Target_attribute* among those *Examples* that match *Best_hypothesis*.

PERFORMANCE(h , *Examples*, *Target_attribute*)

- *h_examples* \leftarrow the subset of *Examples* that match h
- return $-\text{Entropy}(h_examples)$, where entropy is with respect to *Target_attribute*

TABLE 10.2

One implementation for LEARN-ONE-RULE is a general-to-specific beam search. The frontier of current hypotheses is represented by the variable *Candidate_hypotheses*. This algorithm is similar to that used by the CN2 program, described by Clark and Niblett (1989).

A few remarks on the LEARN-ONE-RULE algorithm of Table 10.2 are in order. First, note that each hypothesis considered in the main loop of the algorithm is a conjunction of attribute-value constraints. Each of these conjunctive hypotheses corresponds to a candidate set of preconditions for the rule to be learned and is evaluated by the entropy of the examples it covers. The search considers increasingly specific candidate hypotheses until it reaches a maximally specific hypothesis that contains all available attributes. The rule that is output by the algorithm is the rule encountered during the search whose PERFORMANCE is greatest—not necessarily the final hypothesis generated in the search. The postcondition for the output rule is chosen only in the final step of the algorithm, after its precondition (represented by the variable *Best_hypothesis*) has been determined. The algorithm constructs the rule postcondition to predict the value of the target attribute that is most common among the examples covered by the rule precondition. Finally, note that despite the use of beam search to reduce the risk, the greedy search may still produce suboptimal rules. However, even when this occurs the SEQUENTIAL-COVERING algorithm can still learn a collection of rules that together cover the training examples, because it repeatedly calls LEARN-ONE-RULE on the remaining uncovered examples.

10.2.2 Variations

The SEQUENTIAL-COVERING algorithm, together with the LEARN-ONE-RULE algorithm, learns a set of if-then rules that covers the training examples. Many variations on this approach have been explored. For example, in some cases it might be desirable to have the program learn only rules that cover positive examples and to include a “default” that assigns a negative classification to instances not covered by any rule. This approach might be desirable, say, if one is attempting to learn a target concept such as “pregnant women who are likely to have twins.” In this case, the fraction of positive examples in the entire population is small, so the rule set will be more compact and intelligible to humans if it identifies only classes of positive examples, with the default classification of all other examples as negative. This approach also corresponds to the “negation-as-failure” strategy of PROLOG, in which any expression that cannot be proven to be true is by default assumed to be false. In order to learn such rules that predict just a single target value, the LEARN-ONE-RULE algorithm can be modified to accept an additional input argument specifying the target value of interest. The general-to-specific beam search is conducted just as before, changing only the PERFORMANCE subroutine that evaluates hypotheses. Note the definition of PERFORMANCE as negative entropy is no longer appropriate in this new setting, because it assigns a maximal score to hypotheses that cover exclusively negative examples, as well as those that cover exclusively positive examples. Using a measure that evaluates the fraction of positive examples covered by the hypothesis would be more appropriate in this case.

Another variation is provided by a family of algorithms called AQ (Michalski 1969, Michalski et al. 1986), that predate the CN2 algorithm on which the

above discussion is based. Like CN2, AQ learns a disjunctive set of rules that together cover the target function. However, AQ differs in several ways from the algorithms given here. First, the covering algorithm of AQ differs from the SEQUENTIAL-COVERING algorithm because it explicitly seeks rules that cover a particular target value, learning a disjunctive set of rules for each target value in turn. Second, AQ's algorithm for learning a single rule differs from LEARN-ONE-RULE. While it conducts a general-to-specific beam search for each rule, it uses a single positive example to focus this search. In particular, it considers only those attributes satisfied by the positive example as it searches for progressively more specific hypotheses. Each time it learns a new rule it selects a new positive example from those that are not yet covered, to act as a seed to guide the search for this new disjunct.

10.3 LEARNING RULE SETS: SUMMARY

The SEQUENTIAL-COVERING algorithm described above and the decision tree learning algorithms of Chapter 3 suggest a variety of possible methods for learning sets of rules. This section considers several key dimensions in the design space of such rule learning algorithms.

First, *sequential covering* algorithms learn one rule at a time, removing the covered examples and repeating the process on the remaining examples. In contrast, decision tree algorithms such as ID3 learn the entire set of disjuncts simultaneously as part of the single search for an acceptable decision tree. We might, therefore, call algorithms such as ID3 *simultaneous covering* algorithms, in contrast to sequential covering algorithms such as CN2. Which should we prefer? The key difference occurs in the choice made at the most primitive step in the search. At each search step ID3 chooses among alternative *attributes* by comparing the *partitions* of the data they generate. In contrast, CN2 chooses among alternative *attribute-value* pairs, by comparing the *subsets* of data they cover. One way to see the significance of this difference is to compare the number of distinct choices made by the two algorithms in order to learn the same set of rules. To learn a set of n rules, each containing k attribute-value tests in their preconditions, sequential covering algorithms will perform $n \cdot k$ primitive search steps, making an independent decision to select each precondition of each rule. In contrast, simultaneous covering algorithms will make many fewer independent choices, because each choice of a decision node in the decision tree corresponds to choosing the precondition for the multiple rules associated with that node. In other words, if the decision node tests an attribute that has m possible values, the choice of the decision node corresponds to choosing a precondition for each of the m corresponding rules (see Exercise 10.1). Thus, sequential covering algorithms such as CN2 make a larger number of independent choices than simultaneous covering algorithms such as ID3. Still, the question remains, which should we prefer? The answer may depend on how much training data is available. If data is plentiful, then it may support the larger number of independent decisions required by the sequential covering algorithm, whereas if data is scarce, the “sharing” of

decisions regarding preconditions of different rules may be more effective. An additional consideration is the task-specific question of whether it is desirable that different rules test the same attributes. In the simultaneous covering decision tree learning algorithms, they will. In sequential covering algorithms, they need not.

A second dimension along which approaches vary is the direction of the search in LEARN-ONE-RULE. In the algorithm described above, the search is from *general to specific* hypotheses. Other algorithms we have discussed (e.g., FIND-S from Chapter 2) search from *specific to general*. One advantage of general to specific search in this case is that there is a single maximally general hypothesis from which to begin the search, whereas there are very many specific hypotheses in most hypothesis spaces (i.e., one for each possible instance). Given many maximally specific hypotheses, it is unclear which to select as the starting point of the search. One program that conducts a specific-to-general search, called GOLEM (Muggleton and Feng 1990), addresses this issue by choosing several positive examples at random to initialize and to guide the search. The best hypothesis obtained through multiple random choices is then selected.

A third dimension is whether the LEARN-ONE-RULE search is a *generate then test* search through the syntactically legal hypotheses, as it is in our suggested implementation, or whether it is *example-driven* so that individual training examples constrain the generation of hypotheses. Prototypical example-driven search algorithms include the FIND-S and CANDIDATE-ELIMINATION algorithms of Chapter 2, the AQ algorithm, and the CIGOL algorithm discussed later in this chapter. In each of these algorithms, the generation or revision of hypotheses is driven by the analysis of an individual training example, and the result is a revised hypothesis designed to correct performance for this single example. This contrasts to the generate and test search of LEARN-ONE-RULE in Table 10.2, in which successor hypotheses are generated based only on the syntax of the hypothesis representation. The training data is considered only after these candidate hypotheses are generated and is used to choose among the candidates based on their performance over the entire collection of training examples. One important advantage of the generate and test approach is that each choice in the search is based on the hypothesis performance over *many* examples, so that the impact of noisy data is minimized. In contrast, example-driven algorithms that refine the hypothesis based on individual examples are more easily misled by a single noisy training example and are therefore less robust to errors in the training data.

A fourth dimension is whether and how rules are post-pruned. As in decision tree learning, it is possible for LEARN-ONE-RULE to formulate rules that perform very well on the training data, but less well on subsequent data. As in decision tree learning, one way to address this issue is to post-prune each rule after it is learned from the training data. In particular, preconditions can be removed from the rule whenever this leads to improved performance over a set of pruning examples distinct from the training examples. A more detailed discussion of rule post-pruning is provided in Section 3.7.1.2.

A final dimension is the particular definition of rule PERFORMANCE used to guide the search in LEARN-ONE-RULE. Various evaluation functions have been used. Some common evaluation functions include:

- *Relative frequency.* Let n denote the number of examples the rule matches and let n_c denote the number of these that it classifies correctly. The relative frequency estimate of rule performance is

$$\frac{n_c}{n}$$

Relative frequency is used to evaluate rules in the AQ program.

- *m-estimate of accuracy.* This accuracy estimate is biased toward the default accuracy expected of the rule. It is often preferred when data is scarce and the rule must be evaluated based on few examples. As above, let n and n_c denote the number of examples matched and correctly predicted by the rule. Let p be the prior probability that a randomly drawn example from the entire data set will have the classification assigned by the rule (e.g., if 12 out of 100 examples have the value predicted by the rule, then $p = .12$). Finally, let m be the weight, or equivalent number of examples for weighting this prior p . The m -estimate of rule accuracy is

$$\frac{n_c + mp}{n + m}$$

Note if m is set to zero, then the m -estimate becomes the above relative frequency estimate. As m is increased, a larger number of examples is needed to override the prior assumed accuracy p . The m -estimate measure is advocated by Cestnik and Bratko (1991) and has been used in some versions of the CN2 algorithm. It is also used in the naive Bayes classifier discussed in Section 6.9.1.

- *Entropy.* This is the measure used by the PERFORMANCE subroutine in the algorithm of Table 10.2. Let S be the set of examples that match the rule preconditions. Entropy measures the uniformity of the target function values for this set of examples. We take the negative of the entropy so that better rules will have higher scores.

$$-\text{Entropy}(S) = \sum_{i=1}^c p_i \log_2 p_i$$

where c is the number of distinct values the target function may take on, and where p_i is the proportion of examples from S for which the target function takes on the i th value. This entropy measure, combined with a test for statistical significance, is used in the CN2 algorithm of Clark and Niblett (1989). It is also the basis for the information gain measure used by many decision tree learning algorithms.

10.4 LEARNING FIRST-ORDER RULES

In the previous sections we discussed algorithms for learning sets of propositional (i.e., variable-free) rules. In this section, we consider learning rules that contain variables—in particular, learning first-order Horn theories. Our motivation for considering such rules is that they are much more expressive than propositional rules. Inductive learning of first-order rules or theories is often referred to as *inductive logic programming* (or ILP for short), because this process can be viewed as automatically inferring PROLOG programs from examples. PROLOG is a general purpose, Turing-equivalent programming language in which programs are expressed as collections of Horn clauses.

10.4.1 First-Order Horn Clauses

To see the advantages of first-order representations over propositional (variable-free) representations, consider the task of learning the simple target concept *Daughter*(x, y), defined over pairs of people x and y . The value of *Daughter*(x, y) is *True* when x is the daughter of y , and *False* otherwise. Suppose each person in the data is described by the attributes *Name*, *Mother*, *Father*, *Male*, *Female*. Hence, each training example will consist of the description of two people in terms of these attributes, along with the value of the target attribute *Daughter*. For example, the following is a positive example in which Sharon is the daughter of Bob:

$$\begin{aligned} & (\text{Name}_1 = \text{Sharon}, \quad \text{Mother}_1 = \text{Louise}, \quad \text{Father}_1 = \text{Bob}, \\ & \quad \text{Male}_1 = \text{False}, \quad \text{Female}_1 = \text{True}, \\ & \quad \text{Name}_2 = \text{Bob}, \quad \text{Mother}_2 = \text{Nora}, \quad \text{Father}_2 = \text{Victor}, \\ & \quad \text{Male}_2 = \text{True}, \quad \text{Female}_2 = \text{False}, \quad \text{Daughter}_{1,2} = \text{True}) \end{aligned}$$

where the subscript on each attribute name indicates which of the two persons is being described. Now if we were to collect a number of such training examples for the target concept *Daughter*_{1,2} and provide them to a propositional rule learner such as CN2 or C4.5, the result would be a collection of very specific rules such as

$$\begin{aligned} \text{IF } & (\text{Father}_1 = \text{Bob}) \wedge (\text{Name}_2 = \text{Bob}) \wedge (\text{Female}_1 = \text{True}) \\ \text{THEN } & \text{Daughter}_{1,2} = \text{True} \end{aligned}$$

Although it is correct, this rule is so specific that it will rarely, if ever, be useful in classifying future pairs of people. The problem is that propositional representations offer no general way to describe the essential *relations* among the values of the attributes. In contrast, a program using first-order representations could learn the following general rule:

$$\text{IF } \text{Father}(y, x) \wedge \text{Female}(y), \quad \text{THEN } \text{Daughter}(x, y)$$

where x and y are variables that can be bound to any person.

First-order Horn clauses may also refer to variables in the preconditions that do not occur in the postconditions. For example, one rule for *GrandDaughter* might be

IF $\text{Father}(y, z) \wedge \text{Mother}(z, x) \wedge \text{Female}(y)$
 THEN $\text{GrandDaughter}(x, y)$

Note the variable z in this rule, which refers to the father of y , is not present in the rule postconditions. Whenever such a variable occurs only in the preconditions, it is assumed to be existentially quantified; that is, the rule preconditions are satisfied as long as there exists at least one binding of the variable that satisfies the corresponding literal.

It is also possible to use the same predicates in the rule postconditions and preconditions, enabling the description of recursive rules. For example, the two rules at the beginning of this chapter provide a recursive definition of the concept *Ancestor*(x, y). ILP learning methods such as those described below have been demonstrated to learn a variety of simple recursive functions, such as the above *Ancestor* function, and functions for sorting the elements of a list, removing a specific element from a list, and appending two lists.

10.4.2 Terminology

Before moving on to algorithms for learning sets of Horn clauses, let us introduce some basic terminology from formal logic. All expressions are composed of *constants* (e.g., *Bob*, *Louise*), *variables* (e.g., x , y), *predicate symbols* (e.g., *Married*, *Greater Than*), and *function symbols* (e.g., *age*). The difference between predicates and functions is that predicates take on values of *True* or *False*, whereas functions may take on any constant as their value. We will use lowercase symbols for variables and capitalized symbols for constants. Also, we will use lowercase for functions and capitalized symbols for predicates.

From these symbols, we build up expressions as follows: A *term* is any constant, any variable, or any function applied to any term (e.g., *Bob*, x , *age(Bob)*). A *literal* is any predicate or its negation applied to any term (e.g., *Married(Bob, Louise)*, $\neg\text{Greater Than}(\text{age}(Sue), 20)$). If a literal contains a negation (\neg) symbol, we call it a *negative literal*, otherwise a *positive literal*.

A *clause* is any disjunction of literals, where all variables are assumed to be universally quantified. A *Horn clause* is a clause containing at most one positive literal, such as

$$H \vee \neg L_1 \vee \dots \neg L_n$$

where H is the positive literal, and $\neg L_1 \dots \neg L_n$ are negative literals. Because of the equalities $(B \vee \neg A) = (B \leftarrow A)$ and $\neg(A \wedge B) = (\neg A \vee \neg B)$, the above Horn clause can alternatively be written in the form

$$H \leftarrow (L_1 \wedge \dots \wedge L_n)$$

- Every well-formed expression is composed of *constants* (e.g., *Mary*, 23, or *Joe*), *variables* (e.g., *x*), *predicates* (e.g., *Female*, as in *Female(Mary)*), and *functions* (e.g., *age*, as in *age(Mary)*).
- A *term* is any constant, any variable, or any function applied to any term. Examples include *Mary*, *x*, *age(Mary)*, *age(x)*.
- A *literal* is any predicate (or its negation) applied to any set of terms. Examples include *Female(Mary)*, $\neg\text{Female}(x)$, *Greater_than(age(Mary), 20)*.
- A *ground literal* is a literal that does not contain any variables (e.g., $\neg\text{Female}(\text{Joe})$).
- A *negative literal* is a literal containing a negated predicate (e.g., $\neg\text{Female}(\text{Joe})$).
- A *positive literal* is a literal with no negation sign (e.g., *Female(Mary)*).
- A *clause* is any disjunction of literals $M_1 \vee \dots \vee M_n$ whose variables are universally quantified.
- A *Horn clause* is an expression of the form

$$H \leftarrow (L_1 \wedge \dots \wedge L_n)$$

where $H, L_1 \dots L_n$ are positive literals. H is called the *head* or *consequent* of the Horn clause. The conjunction of literals $L_1 \wedge L_2 \wedge \dots \wedge L_n$ is called the *body* or *antecedents* of the Horn clause.

- For any literals A and B , the expression $(A \leftarrow B)$ is equivalent to $(A \vee \neg B)$, and the expression $\neg(A \wedge B)$ is equivalent to $(\neg A \vee \neg B)$. Therefore, a Horn clause can equivalently be written as the disjunction

$$H \vee \neg L_1 \vee \dots \vee \neg L_n$$

- A *substitution* is any function that replaces variables by terms. For example, the substitution $\{x/3, y/z\}$ replaces the variable x by the term 3 and replaces the variable y by the term z . Given a substitution θ and a literal L we write $L\theta$ to denote the result of applying substitution θ to L .
- A *unifying substitution* for two literals L_1 and L_2 is any substitution θ such that $L_1\theta = L_2\theta$.

TABLE 10.3

Basic definitions from first-order logic.

which is equivalent to the following, using our earlier rule notation

IF $L_1 \wedge \dots \wedge L_n$, THEN H

Whatever the notation, the Horn clause preconditions $L_1 \wedge \dots \wedge L_n$ are called the clause *body* or, alternatively, the clause *antecedents*. The literal H that forms the postcondition is called the clause *head* or, alternatively, the clause *consequent*. For easy reference, these definitions are summarized in Table 10.3, along with other definitions introduced later in this chapter.

10.5 LEARNING SETS OF FIRST-ORDER RULES: FOIL

A variety of algorithms has been proposed for learning first-order rules, or Horn clauses. In this section we consider a program called FOIL (Quinlan 1990) that employs an approach very similar to the SEQUENTIAL-COVERING and LEARN-ONE-RULE algorithms of the previous section. In fact, the FOIL program is the natural extension of these earlier algorithms to first-order representations. Formally, the hypotheses learned by FOIL are sets of first-order rules, where each rule is similar to a Horn clause with two exceptions. First, the rules learned by FOIL are

more restricted than general Horn clauses, because the literals are not permitted to contain function symbols (this reduces the complexity of the hypothesis space search). Second, FOIL rules are more expressive than Horn clauses, because the literals appearing in the body of the rule may be negated. FOIL has been applied to a variety of problem domains. For example, it has been demonstrated to learn a recursive definition of the *QUICKSORT* algorithm and to learn to discriminate legal from illegal chess positions.

The FOIL algorithm is summarized in Table 10.4. Notice the outer loop corresponds to a variant of the *SEQUENTIAL-COVERING* algorithm discussed earlier; that is, it learns new rules one at a time, removing the positive examples covered by the latest rule before attempting to learn the next rule. The inner loop corresponds to a variant of our earlier *LEARN-ONE-RULE* algorithm, extended to accommodate first-order rules. Note also there are a few minor differences between FOIL and these earlier algorithms. In particular, FOIL seeks only rules that predict when the target literal is *True*, whereas our earlier algorithm would seek both rules that predict when it is *True* and rules that predict when it is *False*. Also, FOIL performs a simple hillclimbing search rather than a beam search (equivalently, it uses a beam of width one).

The hypothesis space search performed by FOIL is best understood by viewing it hierarchically. Each iteration through FOIL's outer loop adds a new rule to its disjunctive hypothesis, *Learned_rules*. The effect of each new rule is to gen-

FOIL(Target_predicate, Predicates, Examples)

- $Pos \leftarrow$ those *Examples* for which the *Target_predicate* is *True*
- $Neg \leftarrow$ those *Examples* for which the *Target_predicate* is *False*
- $Learned_rules \leftarrow \{\}$
- while Pos , do
 - Learn a NewRule*
 - $NewRule \leftarrow$ the rule that predicts *Target_predicate* with no preconditions
 - $NewRuleNeg \leftarrow Neg$
 - while $NewRuleNeg$, do
 - Add a new literal to specialize NewRule*
 - $Candidate_literals \leftarrow$ generate candidate new literals for *NewRule*, based on *Predicates*
 - $Best_literal \leftarrow \underset{L \in Candidate_literals}{\operatorname{argmax}} \quad Foil_Gain(L, NewRule)$
 - add *Best_literal* to preconditions of *NewRule*
 - $NewRuleNeg \leftarrow$ subset of *NewRuleNeg* that satisfies *NewRule* preconditions
 - $Learned_rules \leftarrow Learned_rules + NewRule$
 - $Pos \leftarrow Pos - \{\text{members of } Pos \text{ covered by } NewRule\}$
 - Return *Learned_rules*

TABLE 10.4

The basic FOIL algorithm. The specific method for generating *Candidate_literals* and the definition of *Foil_Gain* are given in the text. This basic algorithm can be modified slightly to better accommodate noisy data, as described in the text.

eralize the current disjunctive hypothesis (i.e., to increase the number of instances it classifies as positive), by adding a new disjunct. Viewed at this level, the search is a *specific-to-general* search through the space of hypotheses, beginning with the most specific empty disjunction and terminating when the hypothesis is sufficiently general to cover all positive training examples. The inner loop of FOIL performs a finer-grained search to determine the exact definition of each new rule. This inner loop searches a second hypothesis space, consisting of conjunctions of literals, to find a conjunction that will form the preconditions for the new rule. Within this hypothesis space, it conducts a *general-to-specific*, hill-climbing search, beginning with the most general preconditions possible (the empty precondition), then adding literals one at a time to specialize the rule until it avoids all negative examples.

The two most substantial differences between FOIL and our earlier SEQUENTIAL-COVERING and LEARN-ONE-RULE algorithm follow from the requirement that it accommodate first-order rules. These differences are:

1. In its general-to-specific search to learn each new rule, FOIL employs different detailed steps to generate candidate specializations of the rule. This difference follows from the need to accommodate variables in the rule preconditions.
2. FOIL employs a PERFORMANCE measure, *Foil_Gain*, that differs from the entropy measure shown for LEARN-ONE-RULE in Table 10.2. This difference follows from the need to distinguish between different bindings of the rule variables and from the fact that FOIL seeks only rules that cover positive examples.

The following two subsections consider these two differences in greater detail.

10.5.1 Generating Candidate Specializations in FOIL

To generate candidate specializations of the current rule, FOIL generates a variety of new literals, each of which may be individually added to the rule preconditions. More precisely, suppose the current rule being considered is

$$P(x_1, x_2, \dots, x_k) \leftarrow L_1 \dots L_n$$

where $L_1 \dots L_n$ are literals forming the current rule preconditions and where $P(x_1, x_2, \dots, x_k)$ is the literal that forms the rule head, or postconditions. FOIL generates candidate specializations of this rule by considering new literals L_{n+1} that fit one of the following forms:

- $Q(v_1, \dots, v_r)$, where Q is any predicate name occurring in *Predicates* and where the v_i are either new variables or variables already present in the rule. At least one of the v_i in the created literal must already exist as a variable in the rule.
- $Equal(x_j, x_k)$, where x_j and x_k are variables already present in the rule.

- The negation of either of the above forms of literals.

To illustrate, consider learning rules to predict the target literal $\text{GrandDaughter}(x, y)$, where the other predicates used to describe examples are Father and Female . The general-to-specific search in FOIL begins with the most general rule

$$\text{GrandDaughter}(x, y) \leftarrow$$

which asserts that $\text{GrandDaughter}(x, y)$ is true of any x and y . To specialize this initial rule, the above procedure generates the following literals as candidate additions to the rule preconditions: $\text{Equal}(x, y)$, $\text{Female}(x)$, $\text{Female}(y)$, $\text{Father}(x, y)$, $\text{Father}(y, x)$, $\text{Father}(x, z)$, $\text{Father}(z, x)$, $\text{Father}(y, z)$, $\text{Father}(z, y)$, and the negations of each of these literals (e.g., $\neg\text{Equal}(x, y)$). Note that z is a new variable here, whereas x and y exist already within the current rule.

Now suppose that among the above literals FOIL greedily selects $\text{Father}(y, z)$ as the most promising, leading to the more specific rule

$$\text{GrandDaughter}(x, y) \leftarrow \text{Father}(y, z)$$

In generating candidate literals to further specialize this rule, FOIL will now consider all of the literals mentioned in the previous step, plus the additional literals $\text{Female}(z)$, $\text{Equal}(z, x)$, $\text{Equal}(z, y)$, $\text{Father}(z, w)$, $\text{Father}(w, z)$, and their negations. These new literals are considered at this point because the variable z was added to the rule in the previous step. Because of this, FOIL now considers an additional new variable w .

If FOIL at this point were to select the literal $\text{Father}(z, x)$ and on the next iteration select the literal $\text{Female}(y)$, this would lead to the following rule, which covers only positive examples and hence terminates the search for further specializations of the rule.

$$\text{GrandDaughter}(x, y) \leftarrow \text{Father}(y, z) \wedge \text{Father}(z, x) \wedge \text{Female}(y)$$

At this point, FOIL will remove all positive examples covered by this new rule. If additional positive examples remain to be covered, then it will begin yet another general-to-specific search for an additional rule.

10.5.2 Guiding the Search in FOIL

To select the most promising literal from the candidates generated at each step, FOIL considers the performance of the rule over the training data. In doing this, it considers all possible bindings of each variable in the current rule. To illustrate this process, consider again the example in which we seek to learn a set of rules for the target literal $\text{GrandDaughter}(x, y)$. For illustration, assume the training data includes the following simple set of assertions, where we use the convention that $P(x, y)$ can be read as “The P of x is y .”

$\text{GrandDaughter}(\text{Victor}, \text{Sharon})$	$\text{Father}(\text{Sharon}, \text{Bob})$	$\text{Father}(\text{Tom}, \text{Bob})$
$\text{Female}(\text{Sharon})$		$\text{Father}(\text{Bob}, \text{Victor})$

Here let us also make the closed world assumption that any literal involving the predicate *GrandDaughter*, *Father*, or *Female* and the constants *Victor*, *Sharon*, *Bob*, and *Tom* that is not listed above can be assumed to be false (i.e., we also implicitly assert $\neg\text{GrandDaughter}(\text{Tom}, \text{Bob})$, $\neg\text{GrandDaughter}(\text{Victor}, \text{Victor})$, etc.).

To select the best specialization of the current rule, FOIL considers each distinct way in which the rule variables can bind to constants in the training examples. For example, in the initial step when the rule is

$$\text{GrandDaughter}(x, y) \leftarrow$$

the rule variables x and y are not constrained by any preconditions and may therefore bind in any combination to the four constants *Victor*, *Sharon*, *Bob*, and *Tom*. We will use the notation $\{x/\text{Bob}, y/\text{Sharon}\}$ to denote a particular variable binding; that is, a substitution mapping each variable to a constant. Given the four possible constants, there are 16 possible variable bindings for this initial rule. The binding $\{x/\text{Victor}, y/\text{Sharon}\}$ corresponds to a positive example binding, because the training data includes the assertion *GrandDaughter*(*Victor*, *Sharon*). The other 15 bindings allowed by the rule (e.g., the binding $\{x/\text{Bob}, y/\text{Tom}\}$) constitute negative evidence for the rule in the current example, because no corresponding assertion can be found in the training data.

At each stage, the rule is evaluated based on these sets of positive and negative variable bindings, with preference given to rules that possess more positive bindings and fewer negative bindings. As new literals are added to the rule, the sets of bindings will change. Note if a literal is added that introduces a new variable, then the bindings for the rule will grow in length (e.g., if *Father*(y, z) is added to the above rule, then the original binding $\{x/\text{Victor}, y/\text{Sharon}\}$ will become the more lengthy $\{x/\text{Victor}, y/\text{Sharon}, z/\text{Bob}\}$). Note also that if the new variable can bind to several different constants, then the number of bindings fitting the extended rule can be greater than the number associated with the original rule.

The evaluation function used by FOIL to estimate the utility of adding a new literal is based on the numbers of positive and negative bindings covered before and after adding the new literal. More precisely, consider some rule R , and a candidate literal L that might be added to the body of R . Let R' be the rule created by adding literal L to rule R . The value *Foil_Gain*(L, R) of adding L to R is defined as

$$\text{Foil_Gain}(L, R) \equiv t \left(\log_2 \frac{p_1}{p_1 + n_1} - \log_2 \frac{p_0}{p_0 + n_0} \right) \quad (10.1)$$

where p_0 is the number of positive bindings of rule R , n_0 is the number of negative bindings of R , p_1 is the number of positive bindings of rule R' , and n_1 is the number of negative bindings of R' . Finally, t is the number of positive bindings of rule R that are still covered after adding literal L to R . When a new variable is introduced into R by adding L , then any original binding is considered to be covered so long as some binding extending it is present in the bindings of R' .

This *Foil_Gain* function has a straightforward interpretation in terms of information theory. According to information theory, $-\log_2 \frac{p_0}{p_0+n_0}$ is the minimum number of bits needed to encode the classification of an arbitrary positive binding among the bindings covered by rule R . Similarly, $-\log_2 \frac{p_1}{p_1+n_1}$ is the number of bits required if the binding is one of those covered by rule R' . Since t is just the number of positive bindings covered by R that remain covered by R' , *Foil_Gain*(L, R) can be seen as the reduction due to L in the total number of bits needed to encode the classification of all positive bindings of R .

10.5.3 Learning Recursive Rule Sets

In the above discussion, we ignored the possibility that new literals added to the rule body could refer to the target predicate itself (i.e., the predicate occurring in the rule head). However, if we include the target predicate in the input list of *Predicates*, then FOIL will consider it as well when generating candidate literals. This will allow it to form recursive rules—rules that use the same predicate in the body and the head of the rule. For instance, recall the following rule set that provides a recursive definition of the *Ancestor* relation.

IF <i>Parent</i> (x, y)	THEN <i>Ancestor</i> (x, y)
IF <i>Parent</i> (x, z) \wedge <i>Ancestor</i> (z, y)	THEN <i>Ancestor</i> (x, y)

Given an appropriate set of training examples, these two rules can be learned following a trace similar to the one above for *GrandDaughter*. Note the second rule is among the rules that are potentially within reach of FOIL’s search, provided *Ancestor* is included in the list *Predicates* that determines which predicates may be considered when generating new literals. Of course whether this particular rule would be learned or not depends on whether these particular literals outscore competing candidates during FOIL’s greedy search for increasingly specific rules. Cameron-Jones and Quinlan (1993) discuss several examples in which FOIL has successfully discovered recursive rule sets. They also discuss important subtleties that arise, such as how to avoid learning rule sets that produce infinite recursion.

10.5.4 Summary of FOIL

To summarize, FOIL extends the sequential covering algorithm of CN2 to handle the case of learning first-order rules similar to Horn clauses. To learn each rule FOIL performs a general-to-specific search, at each step adding a single new literal to the rule preconditions. The new literal may refer to variables already mentioned in the rule preconditions or postconditions, and may introduce new variables as well. At each step, it uses the *Foil_Gain* function of Equation (10.1) to select among the candidate new literals. If new literals are allowed to refer to the target predicate, then FOIL can, in principle, learn sets of recursive rules. While this introduces the complexity of avoiding rule sets that result in infinite recursion, FOIL has been demonstrated to successfully learn recursive rule sets in several cases.

In the case of noise-free training data, FOIL may continue adding new literals to the rule until it covers no negative examples. To handle noisy data, the search is continued until some tradeoff occurs between rule accuracy, coverage, and complexity. FOIL uses a minimum description length approach to halt the growth of rules, in which new literals are added only when their description length is shorter than the description length of the training data they explain. The details of this strategy are given in Quinlan (1990). In addition, FOIL post-prunes each rule it learns, using the same rule post-pruning strategy used for decision trees (Chapter 3).

10.6 INDUCTION AS INVERTED DEDUCTION

A second, quite different approach to inductive logic programming is based on the simple observation that induction is just the inverse of deduction! In general, machine learning involves building theories that explain the observed data. Given some data D and some partial background knowledge B , learning can be described as generating a hypothesis h that, together with B , explains D . Put more precisely, assume as usual that the training data D is a set of training examples, each of the form $\langle x_i, f(x_i) \rangle$. Here x_i denotes the i th training instance and $f(x_i)$ denotes its target value. Then learning is the problem of discovering a hypothesis h , such that the classification $f(x_i)$ of each training instance x_i follows deductively from the hypothesis h , the description of x_i , and any other background knowledge B known to the system.

$$(\forall \langle x_i, f(x_i) \rangle \in D) (B \wedge h \wedge x_i) \vdash f(x_i) \quad (10.2)$$

The expression $X \vdash Y$ is read “ Y follows deductively from X ,” or alternatively “ X entails Y .” Expression (10.2) describes the constraint that must be satisfied by the learned hypothesis h ; namely, for every training instance x_i , the target classification $f(x_i)$ must follow deductively from B , h , and x_i .

As an example, consider the case where the target concept to be learned is “pairs of people $\langle u, v \rangle$ such that the child of u is v ,” represented by the predicate $Child(u, v)$. Assume we are given a single positive example $Child(Bob, Sharon)$, where the instance is described by the literals $Male(Bob)$, $Female(Sharon)$, and $Father(Sharon, Bob)$. Furthermore, suppose we have the general background knowledge $Parent(u, v) \leftarrow Father(u, v)$. We can describe this situation in the terms of Equation (10.2) as follows:

$$\begin{array}{ll} x_i : & Male(Bob), Female(Sharon), Father(Sharon, Bob) \\ f(x_i) : & Child(Bob, Sharon) \\ B : & Parent(u, v) \leftarrow Father(u, v) \end{array}$$

In this case, two of the many hypotheses that satisfy the constraint $(B \wedge h \wedge x_i) \vdash f(x_i)$ are

$$\begin{aligned} h_1 &: Child(u, v) \leftarrow Father(v, u) \\ h_2 &: Child(u, v) \leftarrow Parent(v, u) \end{aligned}$$

Note that the target literal $\text{Child}(Bob, Sharon)$ is entailed by $h_1 \wedge x_i$ with no need for the background information B . In the case of hypothesis h_2 , however, the situation is different. The target $\text{Child}(Bob, Sharon)$ follows from $B \wedge h_2 \wedge x_i$, but not from $h_2 \wedge x_i$ alone. This example illustrates the role of background knowledge in expanding the set of acceptable hypotheses for a given set of training data. It also illustrates how new predicates (e.g., Parent) can be introduced into hypotheses (e.g., h_2), even when the predicate is not present in the original description of the instance x_i . This process of augmenting the set of predicates, based on background knowledge, is often referred to as *constructive induction*.

The significance of Equation (10.2) is that it casts the learning problem in the framework of deductive inference and formal logic. In the case of propositional and first-order logics, there exist well-understood algorithms for automated deduction. Interestingly, it is possible to develop inverses of these procedures in order to automate the process of inductive generalization. The insight that induction might be performed by inverting deduction appears to have been first observed by the nineteenth century economist W. S. Jevons, who wrote:

Induction is, in fact, the inverse operation of deduction, and cannot be conceived to exist without the corresponding operation, so that the question of relative importance cannot arise. Who thinks of asking whether addition or subtraction is the more important process in arithmetic? But at the same time much difference in difficulty may exist between a direct and inverse operation; ... it must be allowed that inductive investigations are of a far higher degree of difficulty and complexity than any questions of deduction. . . . (Jevons 1874)

In the remainder of this chapter we will explore this view of induction as the inverse of deduction. The general issue we will be interested in here is designing *inverse entailment operators*. An inverse entailment operator, $O(B, D)$ takes the training data $D = \{(x_i, f(x_i))\}$ and background knowledge B as input and produces as output a hypothesis h satisfying Equation (10.2).

$$O(B, D) = h \text{ such that } (\forall (x_i, f(x_i)) \in D) (B \wedge h \wedge x_i) \vdash f(x_i)$$

Of course there will, in general, be many different hypotheses h that satisfy $(\forall (x_i, f(x_i)) \in D) (B \wedge h \wedge x_i) \vdash f(x_i)$. One common heuristic in ILP for choosing among such hypotheses is to rely on the heuristic known as the Minimum Description Length principle (see Section 6.6).

There are several attractive features to formulating the learning task as finding a hypothesis h that solves the relation $(\forall (x_i, f(x_i)) \in D) (B \wedge h \wedge x_i) \vdash f(x_i)$.

- This formulation subsumes the common definition of learning as finding some general concept that matches a given set of training examples (which corresponds to the special case where no background knowledge B is available).
- By incorporating the notion of background information B , this formulation allows a more rich definition of when a hypothesis may be said to “fit” the data. Up until now, we have always determined whether a hypothesis

(e.g., neural network) fits the data based solely on the description of the hypothesis and data, independent of the task domain under study. In contrast, this formulation allows the domain-specific background information B to become part of the definition of “fit.” In particular, h fits the training example $\langle x_i, f(x_i) \rangle$ as long as $f(x_i)$ follows deductively from $B \wedge h \wedge x_i$.

- By incorporating background information B , this formulation invites learning methods that use this background information to guide the search for h , rather than merely searching the space of syntactically legal hypotheses. The inverse resolution procedure described in the following section uses background knowledge in this fashion.

At the same time, research on inductive logic programming following this formulation has encountered several practical difficulties.

- The requirement $(\forall \langle x_i, f(x_i) \rangle \in D) (B \wedge h \wedge x_i) \vdash f(x_i)$ does not naturally accommodate noisy training data. The problem is that this expression does not allow for the possibility that there may be errors in the observed description of the instance x_i or its target value $f(x_i)$. Such errors can produce an inconsistent set of constraints on h . Unfortunately, most formal logic frameworks completely lose their ability to distinguish between truth and falsehood once they are given inconsistent sets of assertions.
- The language of first-order logic is so expressive, and the number of hypotheses that satisfy $(\forall \langle x_i, f(x_i) \rangle \in D) (B \wedge h \wedge x_i) \vdash f(x_i)$ is so large, that the search through the space of hypotheses is intractable in the general case. Much recent work has sought restricted forms of first-order expressions, or additional second-order knowledge, to improve the tractability of the hypothesis space search.
- Despite our intuition that background knowledge B should help constrain the search for a hypothesis, in most ILP systems (including all discussed in this chapter) the complexity of the hypothesis space search *increases* as background knowledge B is increased. (However, see Chapters 11 and 12 for algorithms that use background knowledge to *decrease* rather than increase sample complexity).

In the following section, we examine one quite general inverse entailment operator that constructs hypotheses by inverting a deductive inference rule.

10.7 INVERTING RESOLUTION

A general method for automated deduction is the *resolution rule* introduced by Robinson (1965). The resolution rule is a sound and complete rule for deductive inference in first-order logic. Therefore, it is sensible to ask whether we can invert the resolution rule to form an inverse entailment operator. The answer is yes, and it is just this operator that forms the basis of the CIGOL program introduced by Muggleton and Buntine (1988).

It is easiest to introduce the resolution rule in propositional form, though it is readily extended to first-order representations. Let L be an arbitrary propositional literal, and let P and R be arbitrary propositional clauses. The resolution rule is

$$\frac{\begin{array}{ccc} P & \vee & L \\ \neg L & \vee & R \end{array}}{P \vee R}$$

which should be read as follows: Given the two clauses above the line, conclude the clause below the line. Intuitively, the resolution rule is quite sensible. Given the two assertions $P \vee L$ and $\neg L \vee R$, it is obvious that either L or $\neg L$ must be false. Therefore, either P or R must be true. Thus, the conclusion $P \vee R$ of the resolution rule is intuitively satisfying.

The general form of the propositional resolution operator is described in Table 10.5. Given two clauses C_1 and C_2 , the resolution operator first identifies a literal L that occurs as a positive literal in one of these two clauses and as a negative literal in the other. It then draws the conclusion given by the above formula. For example, consider the application of the resolution operator illustrated on the left side of Figure 10.2. Given clauses C_1 and C_2 , the first step of the procedure identifies the literal $L = \neg KnowMaterial$, which is present in C_1 , and whose negation $\neg(\neg KnowMaterial) = KnowMaterial$ is present in C_2 . Thus the conclusion is the clause formed by the union of the literals $C_1 - \{L\} = PassExam$ and $C_2 - \{\neg L\} = \neg Study$. As another example, the result of applying the resolution rule to the clauses $C_1 = A \vee B \vee C \vee \neg D$ and $C_2 = \neg B \vee E \vee F$ is the clause $A \vee C \vee \neg D \vee E \vee F$.

It is easy to invert the resolution operator to form an inverse entailment operator $O(C, C_1)$ that performs inductive inference. In general, the inverse entailment operator must derive one of the initial clauses, C_2 , given the resolvent C and the other initial clause C_1 . Consider an example in which we are given the resolvent $C = A \vee B$ and the initial clause $C_1 = B \vee D$. How can we derive a clause C_2 such that $C_1 \wedge C_2 \vdash C$? First, note that by the definition of the resolution operator, any literal that occurs in C but not in C_1 must have been present in C_2 . In our example, this indicates that C_2 must contain the literal A . Second, the literal

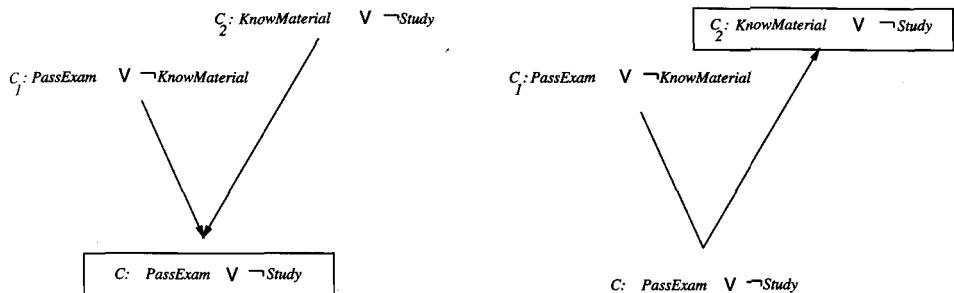
- Given initial clauses C_1 and C_2 , find a literal L from clause C_1 such that $\neg L$ occurs in clause C_2 .
- Form the resolvent C by including all literals from C_1 and C_2 , except for L and $\neg L$. More precisely, the set of literals occurring in the conclusion C is

$$C = (C_1 - \{L\}) \cup (C_2 - \{\neg L\})$$

where \cup denotes set union, and “ $-$ ” denotes set difference.

TABLE 10.5

Resolution operator (propositional form). Given clauses C_1 and C_2 , the resolution operator constructs a clause C such that $C_1 \wedge C_2 \vdash C$.

**FIGURE 10.2**

On the left, an application of the (deductive) resolution rule inferring clause C from the given clauses C_1 and C_2 . On the right, an application of its (inductive) inverse, inferring C_2 from C and C_1 .

that occurs in C_1 but not in C must be the literal removed by the resolution rule, and therefore its negation must occur in C_2 . In our example, this indicates that C_2 must contain the literal $\neg D$. Hence, $C_2 = A \vee \neg D$. The reader can easily verify that applying the resolution rule to C_1 and C_2 does, in fact, produce the desired resolvent C .

Notice there is a second possible solution for C_2 in the above example. In particular, C_2 can also be the more specific clause $A \vee \neg D \vee B$. The difference between this and our first solution is that we have now included in C_2 a literal that occurred in C_1 . The general point here is that inverse resolution is not deterministic—in general there may be multiple clauses C_2 such that C_1 and C_2 produce the resolvent C . One heuristic for choosing among the alternatives is to prefer shorter clauses over longer clauses, or equivalently, to assume C_2 shares no literals in common with C_1 . If we incorporate this bias toward short clauses, the general statement of this inverse resolution procedure is as shown in Table 10.6.

We can develop rule-learning algorithms based on inverse entailment operators such as inverse resolution. In particular, the learning algorithm can use inverse entailment to construct hypotheses that, together with the background information, entail the training data. One strategy is to use a sequential covering algorithm to iteratively learn a set of Horn clauses in this way. On each iteration, the algorithm selects a training example $\langle x_i, f(x_i) \rangle$ that is not yet covered by previously learned clauses. The inverse resolution rule is then applied to

- Given initial clauses C_1 and C , find a literal L that occurs in clause C_1 , but not in clause C .
- Form the second clause C_2 by including the following literals

$$C_2 = (C - (C_1 - \{L\})) \cup \{\neg L\}$$

TABLE 10.6

Inverse resolution operator (propositional form). Given two clauses C and C_1 , this computes a clause C_2 such that $C_1 \wedge C_2 \vdash C$.

generate candidate hypotheses h_i that satisfy $(B \wedge h_i \wedge x_i) \vdash f(x_i)$, where B is the background knowledge plus any clauses learned on previous iterations. Note this is an example-driven search, because each candidate hypothesis is constructed to cover a particular example. Of course if multiple candidate hypotheses exist, then one strategy for selecting among them is to choose the one with highest accuracy over the other examples as well. The CIGOL program uses inverse resolution with this kind of sequential covering algorithm, interacting with the user along the way to obtain training examples and to obtain guidance in its search through the vast space of possible inductive inference steps. However, CIGOL uses first-order rather than propositional representations. Below we describe the extension of the resolution rule required to accommodate first-order representations.

10.7.1 First-Order Resolution

The resolution rule extends easily to first-order expressions. As in the propositional case, it takes two clauses as input and produces a third clause as output. The key difference from the propositional case is that the process is now based on the notion of *unifying* substitutions.

We define a *substitution* to be any mapping of variables to terms. For example, the substitution $\theta = \{x/Bob, y/z\}$ indicates that the variable x is to be replaced by the term *Bob*, and that the variable y is to be replaced by the term *z*. We use the notation $W\theta$ to denote the result of applying the substitution θ to some expression W . For example, if L is the literal $Father(x, Bill)$ and θ is the substitution defined above, then $L\theta = Father(Bob, Bill)$.

We say that θ is a *unifying substitution* for two literals L_1 and L_2 , provided $L_1\theta = L_2\theta$. For example, if $L_1 = Father(x, y)$, $L_2 = Father(Bill, z)$, and $\theta = \{x/Bill, z/y\}$, then θ is a unifying substitution for L_1 and L_2 because $L_1\theta = L_2\theta = Father(Bill, y)$. The significance of a unifying substitution is this: In the propositional form of resolution, the resolvent of two clauses C_1 and C_2 is found by identifying a literal L that appears in C_1 such that $\neg L$ appears in C_2 . In first-order resolution, this generalizes to finding one literal L_1 from clause C_1 and one literal L_2 from C_2 , such that some unifying substitution θ can be found for L_1 and $\neg L_2$ (i.e., such that $L_1\theta = \neg L_2\theta$). The resolution rule then constructs the resolvent C according to the equation

$$C = (C_1 - \{L_1\})\theta \cup (C_2 - \{L_2\})\theta \quad (10.3)$$

The general statement of the resolution rule is shown in Table 10.7. To illustrate, suppose $C_1 = White(x) \leftarrow Swan(x)$ and suppose $C_2 = Swan(Fred)$. To apply the resolution rule, we first re-express C_1 in clause form as the equivalent expression $C_1 = White(x) \vee \neg Swan(x)$. The resolution rule can now be applied. In the first step, it finds the literal $L_1 = \neg Swan(x)$ from C_1 and the literal $L_2 = Swan(Fred)$ from C_2 . If we choose the unifying substitution $\theta = \{x/Fred\}$ then these two literals satisfy $L_1\theta = \neg L_2\theta = \neg Swan(Fred)$. Therefore, the conclusion C is the union of $(C_1 - \{L_1\})\theta = White(Fred)$ and $(C_2 - \{L_2\})\theta = \emptyset$, or $C = White(Fred)$.

1. Find a literal L_1 from clause C_1 , literal L_2 from clause C_2 , and substitution θ such that $L_1\theta = \neg L_2\theta$.
2. Form the resolvent C by including all literals from $C_1\theta$ and $C_2\theta$, except for $L_1\theta$ and $\neg L_2\theta$. More precisely, the set of literals occurring in the conclusion C is

$$C = (C_1 - \{L_1\})\theta \cup (C_2 - \{L_2\})\theta$$

TABLE 10.7

Resolution operator (first-order form).

10.7.2 Inverting Resolution: First-Order Case

We can derive the inverse resolution operator analytically, by algebraic manipulation of Equation (10.3) which defines the resolution rule. First, note the unifying substitution θ in Equation (10.3) can be uniquely factored into θ_1 and θ_2 , where $\theta = \theta_1\theta_2$, where θ_1 contains all substitutions involving variables from clause C_1 , and where θ_2 contains all substitutions involving variables from C_2 . This factorization is possible because C_1 and C_2 will always begin with distinct variable names (because they are distinct universally quantified statements). Using this factorization of θ , we can restate Equation (10.3) as

$$C = (C_1 - \{L_1\})\theta_1 \cup (C_2 - \{L_2\})\theta_2$$

Keep in mind that “ $-$ ” here stands for set difference. Now if we restrict inverse resolution to infer only clauses C_2 that contain no literals in common with C_1 (corresponding to a preference for shortest C_2 clauses), then we can re-express the above as

$$C - (C_1 - \{L_1\})\theta_1 = (C_2 - \{L_2\})\theta_2$$

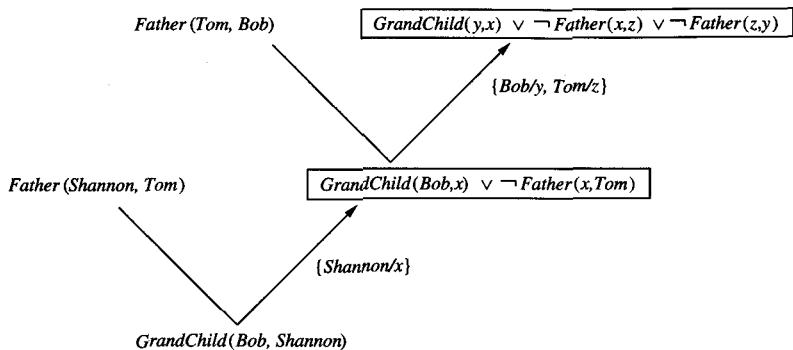
Finally we use the fact that by definition of the resolution rule $L_2 = \neg L_1\theta_1\theta_2^{-1}$, and solve for C_2 to obtain

Inverse resolution:

$$C_2 = (C - (C_1 - \{L_1\})\theta_1)\theta_2^{-1} \cup \{\neg L_1\theta_1\theta_2^{-1}\} \quad (10.4)$$

Equation (10.4) gives the inverse resolution rule for first-order logic. As in the propositional case, this inverse entailment operator is nondeterministic. In particular, in applying it we may in general find multiple choices for the clause C_1 to be resolved and for the unifying substitutions θ_1 and θ_2 . Each set of choices may yield a different solution for C_2 .

Figure 10.3 illustrates a multistep application of this inverse resolution rule for a simple example. In this figure, we wish to learn rules for the target predicate $GrandChild(y, x)$, given the training data $D = GrandChild(Bob, Shannon)$ and the background information $B = \{Father(Shannon, Tom), Father(Tom, Bob)\}$. Consider the bottommost step in the inverse resolution tree of Figure 10.3. Here, we set the conclusion C to the training example $GrandChild(Bob, Shannon)$

**FIGURE 10.3**

A multistep inverse resolution. In each case, the boxed clause is the result of the inference step. For each step, C is the clause at the bottom, C_1 the clause to the left, and C_2 the boxed clause to the right. In both inference steps here, θ_1 is the empty substitution {}, and θ_2^{-1} is the substitution shown below C_2 . Note the final conclusion (the boxed clause at the top right) is the alternative form of the Horn clause $\text{GrandChild}(y, x) \leftarrow \text{Father}(x, z) \wedge \text{Father}(z, y)$.

and select the clause $C_1 = \text{Father}(\text{Shannon}, \text{Tom})$ from the background information. To apply the inverse resolution operator we have only one choice for the literal L_1 , namely $\text{Father}(\text{Shannon}, \text{Tom})$. Suppose we choose the inverse substitutions $\theta_1^{-1} = \{\}$ and $\theta_2^{-1} = \{\text{Shannon}/x\}$. In this case, the resulting clause C_2 is the union of the clause $(C - (C_1 - \{L_1\})\theta_1)\theta_2^{-1} = (C\theta_1)\theta_2^{-1} = \text{GrandChild}(\text{Bob}, x)$, and the clause $\{\neg L_1\theta_1\theta_2^{-1}\} = \neg\text{Father}(x, \text{Tom})$. Hence the result is the clause $\text{GrandChild}(\text{Bob}, x) \vee \neg\text{Father}(x, \text{Tom})$, or equivalently $(\text{GrandChild}(\text{Bob}, x) \leftarrow \text{Father}(x, \text{Tom}))$. Note this general rule, together with C_1 entails the training example $\text{GrandChild}(\text{Bob}, \text{Shannon})$.

In similar fashion, this inferred clause may now be used as the conclusion C for a second inverse resolution step, as illustrated in Figure 10.3. At each such step, note there are several possible outcomes, depending on the choices for the substitutions. (See Exercise 10.7.) In the example of Figure 10.3, the particular set of choices produces the intuitively satisfying final clause $\text{GrandChild}(y, x) \leftarrow \text{Father}(x, z) \wedge \text{Father}(z, y)$.

10.7.3 Summary of Inverse Resolution

To summarize, inverse resolution provides a general approach to automatically generating hypotheses h that satisfy the constraint $(B \wedge h \wedge x_i) \vdash f(x_i)$. This is accomplished by inverting the general resolution rule given by Equation (10.3). Beginning with the resolution rule and solving for the clause C_2 , the inverse resolution rule of Equation (10.4) is easily derived.

Given a set of beginning clauses, multiple hypotheses may be generated by repeated application of this inverse resolution rule. Note the inverse resolution rule has the advantage that it generates *only* hypotheses that satisfy $(B \wedge h \wedge x_i) \vdash f(x_i)$.

In contrast, the generate-and-test search of FOIL generates many hypotheses at each search step, including some that do not satisfy this constraint. FOIL then considers the data D to choose among these hypotheses. Given this difference, we might expect the search based on inverse resolution to be more focused and efficient. However, this will not necessarily be the case. One reason is that the inverse resolution operator can consider only a small fraction of the available data when generating its hypothesis at any given step, whereas FOIL considers all available data to select among its syntactically generated hypotheses. The differences between search strategies that use inverse entailment and those that use generate-and-test search is a subject of ongoing research. Srinivasan et al. (1995) provide one experimental comparison of these two approaches.

10.7.4 Generalization, θ -Subsumption, and Entailment

The previous section pointed out the correspondence between induction and inverse entailment. Given our earlier focus on using the general-to-specific ordering to organize the hypothesis search, it is interesting to consider the relationship between the *more_general_than* relation and inverse entailment. To illuminate this relationship, consider the following definitions.

- *more_general_than*. In Chapter 2, we defined the *more_general_than_or_equal_to* relation (\geq_g) as follows: Given two boolean-valued functions $h_j(x)$ and $h_k(x)$, we say that $h_j \geq_g h_k$ if and only if $(\forall x)h_k(x) \rightarrow h_j(x)$. This \geq_g relation is used by many learning algorithms to guide search through the hypothesis space.
- *θ -subsumption*. Consider two clauses C_j and C_k , both of the form $H \vee L_1 \vee \dots \vee L_n$, where H is a positive literal, and the L_i are arbitrary literals. Clause C_j is said to *θ -subsume* clause C_k if and only if there exists a substitution θ such that $C_j\theta \subseteq C_k$ (where we here describe any clause C by the set of literals in its disjunctive form). This definition is due to Plotkin (1970).
- *Entailment*. Consider two clauses C_j and C_k . Clause C_j is said to *entail* clause C_k (written $C_j \vdash C_k$) if and only if C_k follows deductively from C_j .

What is the relationship among these three definitions? First, let us re-express the definition of \geq_g using the same first-order notation as the other two definitions. If we consider a boolean-valued hypothesis $h(x)$ for some target concept $c(x)$, where $h(x)$ is expressed by a conjunction of literals, then we can re-express the hypothesis as the clause

$$c(x) \leftarrow h(x)$$

Here we follow the usual PROLOG interpretation that x is classified a negative example if it cannot be proven to be a positive example. Hence, we can see that our earlier definition of \geq_g applies to the preconditions, or bodies, of Horn clauses. The implicit postcondition of the Horn clause is the target concept $c(x)$.

What is the relationship between this definition of \geq_g and the definition of θ -subsumption? Note that if $h_1 \geq_g h_2$, then the clause $C_1 : c(x) \leftarrow h_1(x)$ θ -subsumes the clause $C_2 : c(x) \leftarrow h_2(x)$. Furthermore, θ -subsumption can hold even when the clauses have different heads. For example, clause A θ -subsumes clause B in the following case:

$$\begin{aligned} A : \quad & Mother(x, y) \quad \leftarrow Father(x, z) \wedge Spouse(z, y) \\ B : \quad & Mother(x, Louise) \leftarrow Father(x, Bob) \wedge Spouse(Bob, y) \wedge Female(x) \end{aligned}$$

because $A\theta \subseteq B$ if we choose $\theta = \{y/Louise, z/Bob\}$. The key difference here is that \geq_g implicitly assumes two clauses for which the heads are the same, whereas θ -subsumption can hold even for clauses with different heads.

Finally, θ -subsumption is a special case of entailment. That is, if clause A θ -subsumes clause B , then $A \vdash B$. However, we can find clauses A and B such that $A \vdash B$, but where A does not θ -subsume B . One example is the following pair of clauses

$$\begin{aligned} A : \quad & Elephant(father_of(x)) \quad \leftarrow Elephant(x) \\ B : \quad & Elephant(father_of(father_of(y))) \leftarrow Elephant(y) \end{aligned}$$

where $father_of(x)$ is a function that refers to the individual who is the father of x . Note that although B can be proven from A , there is no substitution θ that allows B to be θ -subsumed by A .

As shown by these examples, our earlier notion of *more_general_than* is a special case of θ -subsumption, which is itself a special case of entailment. Therefore, searching the hypothesis space by generalizing or specializing hypotheses is more limited than searching by using general inverse entailment operators. Unfortunately, in its most general form, inverse entailment produces intractable searches. However, the intermediate notion of θ -subsumption provides one convenient notion that lies midway between our earlier definition of *more_general_than* and entailment.

10.7.5 PROGOL

Although inverse resolution is an intriguing method for generating candidate hypotheses, in practice it can easily lead to a combinatorial explosion of candidate hypotheses. An alternative approach is to use inverse entailment to generate just the single most specific hypothesis that, together with the background information, entails the observed data. This most specific hypothesis can then be used to bound a general-to-specific search through the hypothesis space similar to that used by FOIL, but with the additional constraint that the only hypotheses considered are hypotheses more general than this bound. This approach is employed by the PROGOL system, whose algorithm can be summarized as follows:

1. The user specifies a restricted language of first-order expressions to be used as the hypothesis space H . Restrictions are stated using “mode declarations,”

which enable the user to specify the predicate and function symbols to be considered, and the types and formats of arguments for each.

2. PROGOL uses a sequential covering algorithm to learn a set of expressions from H that cover the data. For each example $\langle x_i, f(x_i) \rangle$ that is not yet covered by these learned expressions, it first searches for the most specific hypothesis h_i within H such that $(B \wedge h_i \wedge x_i) \vdash f(x_i)$. More precisely, it approximates this by calculating the most specific hypothesis among those that entail $f(x_i)$ within k applications of the resolution rule (where k is a user-specified parameter).
3. PROGOL then performs a general-to-specific search of the hypothesis space bounded by the most general possible hypothesis and by the specific bound h_i calculated in step 2. Within this set of hypotheses, it seeks the hypothesis having minimum description length (measured by the number of literals). This part of the search is guided by an A^* -like heuristic that allows pruning without running the risk of pruning away the shortest hypothesis.

The details of the PROGOL algorithm are described by Muggleton (1992, 1995).

10.8 SUMMARY AND FURTHER READING

The main points of this chapter include:

- The sequential covering algorithm learns a disjunctive set of rules by first learning a single accurate rule, then removing the positive examples covered by this rule and iterating the process over the remaining training examples. It provides an efficient, greedy algorithm for learning rule sets, and an alternative to top-down decision tree learning algorithms such as ID3, which can be viewed as simultaneous, rather than sequential covering algorithms.
- In the context of sequential covering algorithms, a variety of methods have been explored for learning a single rule. These methods vary in the search strategy they use for examining the space of possible rule preconditions. One popular approach, exemplified by the CN2 program, is to conduct a general-to-specific beam search, generating and testing progressively more specific rules until a sufficiently accurate rule is found. Alternative approaches search from specific to general hypotheses, use an example-driven search rather than generate and test, and employ different statistical measures of rule accuracy to guide the search.
- Sets of first-order rules (i.e., rules containing variables) provide a highly expressive representation. For example, the programming language PROLOG represents general programs using collections of first-order Horn clauses. The problem of learning first-order Horn clauses is therefore often referred to as the problem of inductive logic programming.
- One approach to learning sets of first-order rules is to extend the sequential covering algorithm of CN2 from propositional to first-order representations.

This approach is exemplified by the FOIL program, which can learn sets of first-order rules, including simple recursive rule sets.

- A second approach to learning first-order rules is based on the observation that induction is the inverse of deduction. In other words, the problem of induction is to find a hypothesis h that satisfies the constraint

$$(\forall \langle x_i, f(x_i) \rangle \in D) (B \wedge h \wedge x_i) \vdash f(x_i)$$

where B is general background information, $x_1 \dots x_n$ are descriptions of the instances in the training data D , and $f(x_1) \dots f(x_n)$ are the target values of the training instances.

- Following the view of induction as the inverse of deduction, some programs search for hypotheses by using operators that invert the well-known operators for deductive reasoning. For example, CIGOL uses inverse resolution, an operation that is the inverse of the deductive resolution operator commonly used for mechanical theorem proving. PROGOL combines an inverse entailment strategy with a general-to-specific strategy for searching the hypothesis space.

Early work on learning relational descriptions includes Winston's (1970) well-known program for learning network-style descriptions for concepts such as "arch." Banerji's (1964, 1969) work and Michalski's series of AQ programs (e.g., Michalski 1969; Michalski et al. 1986) were among the earliest to explore the use of logical representations in learning. Plotkin's (1970) definition of θ -subsumption provided an early formalization of the relationship between induction and deduction. Vere (1975) also explored learning logical representations, and Buchanan's (1976) META-DENDRAL program learned relational descriptions representing molecular substructures likely to fragment in a mass spectrometer. This program succeeded in discovering useful rules that were subsequently published in the chemistry literature. Mitchell's (1979) CANDIDATE-ELIMINATION version space algorithm was applied to these same relational descriptions of chemical structures.

With the popularity of the PROLOG language in the mid-1980s, researchers began to look more carefully at learning relational descriptions represented by Horn clauses. Early work on learning Horn clauses includes Shapiro's (1983) MIS and Sammut and Banerji's (1986) MARVIN. Quinlan's (1990) FOIL algorithm, discussed here, was quickly followed by a number of algorithms employing a general-to-specific search for first-order rules including mFOIL (Džeroski 1991), FOCL (Pazzani et al. 1991), CLAUDIEN (De Raedt and Bruynooghe 1993), and MARKUS (Grobelnik 1992). The FOCL algorithm is described in Chapter 12.

An alternative line of research on learning Horn clauses by inverse entailment was spurred by Muggleton and Buntine (1988), who built on related ideas by Sammut and Banerji (1986) and Muggleton (1987). More recent work along this line has focused on alternative search strategies and methods for constraining the hypothesis space to make learning more tractable. For example, Kietz and

Wrobel (1992) use rule schemata in their RDT program to restrict the form of expressions that may be considered during learning, and Muggleton and Feng (1992) discuss the restriction of first-order expressions to ij -determinate literals. Cohen (1994) discusses the GRENDEL program, which accepts as input an explicit description of the language for describing the clause body, thereby allowing the user to explicitly constrain the hypothesis space.

Lavrač and Džeroski (1994) provide a very readable textbook on inductive logic programming. Other useful recent monographs and edited collections include (Bergadano and Gunetti 1995; Morik et al. 1993; Muggleton 1992, 1995b). The overview chapter by Wrobel (1996) also provides a good perspective on the field. Bratko and Muggleton (1995) summarize a number of recent applications of ILP to problems of practical importance. A series of annual workshops on ILP provides a good source of recent research papers (e.g., see De Raedt 1996).

EXERCISES

- 10.1. Consider a sequential covering algorithm such as CN2 and a simultaneous covering algorithm such as ID3. Both algorithms are to be used to learn a target concept defined over instances represented by conjunctions of n boolean attributes. If ID3 learns a balanced decision tree of depth d , it will contain $2^d - 1$ distinct decision nodes, and therefore will have made $2^d - 1$ distinct choices while constructing its output hypothesis. How many rules will be formed if this tree is re-expressed as a disjunctive set of rules? How many preconditions will each rule possess? How many distinct choices would a *sequential* covering algorithm have to make to learn this same set of rules? Which system do you suspect would be more prone to overfitting if both were given the same training data?
- 10.2. Refine the LEARN-ONE-RULE algorithm of Table 10.2 so that it can learn rules whose preconditions include thresholds on real-valued attributes (e.g., *temperature* > 42). Specify your new algorithm as a set of editing changes to the algorithm of Table 10.2. Hint: Consider how this is accomplished for decision tree learning.
- 10.3. Refine the LEARN-ONE-RULE algorithm of Table 10.2 so that it can learn rules whose preconditions include constraints such as *nationality* ∈ {Canadian, Brazilian}, where a discrete-valued attribute is allowed to take on any value in some specified set. Your modified program should explore the hypothesis space containing all such subsets. Specify your new algorithm as a set of editing changes to the algorithm of Table 10.2.
- 10.4. Consider the options for implementing LEARN-ONE-RULE in terms of the possible strategies for searching the hypothesis space. In particular, consider the following attributes of the search
 - (a) generate-and-test versus data-driven
 - (b) general-to-specific versus specific-to-general
 - (c) sequential cover versus simultaneous cover
 Discuss the benefits of the choice made by the algorithm in Tables 10.1 and 10.2. For each of these three attributes of the search strategy, discuss the (positive and negative) impact of choosing the alternative option.
- 10.5. Apply inverse resolution in propositional form to the clauses $C = A \vee B$, $C_1 = A \vee B \vee G$. Give at least two possible results for C_2 .

- 10.6. Apply inverse resolution to the clauses $C = R(B, x) \vee P(x, A)$ and $C_1 = S(B, y) \vee R(z, x)$. Give at least four possible results for C_2 . Here A and B are constants, x and y are variables.
- 10.7. Consider the bottom-most inverse resolution step in Figure 10.3. Derive at least two different outcomes that could result given different choices for the substitutions θ_1 and θ_2 . Derive a result for the inverse resolution step if the clause *Father(Tom, Bob)* is used in place of *Father(Shannon, Tom)*.
- 10.8. Consider the relationship between the definition of the induction problem in this chapter

$$(\forall \langle x_i, f(x_i) \rangle \in D) (B \wedge h \wedge x_i) \vdash f(x_i)$$

and our earlier definition of inductive bias from Chapter 2, Equation 2.1. There we defined the inductive bias, B_{bias} , by the expression

$$(\forall x_i \in X) (B_{bias} \wedge D \wedge x_i) \vdash L(x_i, D)$$

where $L(x_i, D)$ is the classification that the learner assigns to the new instance x_i after learning from the training data D , and where X is the entire instance space. Note the first expression is intended to describe the hypothesis we wish the learner to output, whereas the second expression is intended to describe the learner's policy for generalizing beyond the training data. Invent a learner for which the inductive bias B_{bias} of the learner is identical to the background knowledge B that it is provided.

REFERENCES

- Banerji, R. (1964). A language for the description of concepts. *General Systems*, 9, 135–141.
- Banerji, R. (1969). *Theory of problem solving—an approach to artificial intelligence*. New York: American Elsevier Publishing Company.
- Bergadano, F., & Gunetti, D. (1995). *Inductive logic programming: From machine learning to software engineering*. Cambridge, Ma: MIT Press.
- Bratko, I., & Muggleton, S. (1995). Applications of inductive logic programming. *Communications of the ACM*, 38(11), 65–70.
- Buchanan, B. G., Smith, D. H., White, W. C., Gritter, R., Feigenbaum, E. A., Lederberg, J., & Djerassi, C. (1976). Applications of artificial intelligence for chemical inference, XXII: Automatic rule formation in mass spectrometry by means of the meta-DENDRAL program. *Journal of the American Chemical Society*, 98, 6168.
- Buntine, W. (1986). Generalised subsumption. *Proceedings of the European Conference on Artificial Intelligence*, London.
- Buntine, W. (1988). Generalized subsumption and its applications to induction and redundancy. *Artificial Intelligence*, 36, 149–176.
- Cameron-Jones, R., & Quinlan, J. R. (1993). Avoiding pitfalls when learning recursive theories. *Proceedings of the Eighth International Workshop on Machine Learning* (pp 389–393). San Mateo, CA: Morgan Kaufmann.
- Cestnik, B., & Bratko, I. (1991). On estimating probabilities in tree pruning. *Proceedings of the European Working Session on Machine Learning* (pp. 138–150). Porto, Portugal.
- Clark, P., & Niblett, R. (1989). The CN2 induction algorithm. *Machine Learning*, 3, 261–284.
- Cohen, W. (1994). Grammatically biased learning: Learning logic programs using an explicit antecedent description language. *Artificial Intelligence*, 68(2), 303–366.
- De Raedt, L. (1992). *Interactive theory revision: An inductive logic programming approach*. London: Academic Press.

- De Raedt, L., & Bruynooghe, M. (1993). A theory of clausal discovery. *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*. San Mateo, CA: Morgan Kaufmann.
- De Raedt, L. (Ed.). (1996). *Advances in inductive logic programming: Proceedings of the Fifth International Workshop on Inductive Logic Programming*. Amsterdam: IOS Press.
- Dolsak, B., & Muggleton, S. (1992). The application of inductive logic programming to finite element mesh design. In S. Muggleton (Ed.), *Inductive Logic Programming*. London: Academic Press.
- Džeroski, S. (1991). *Handling noise in inductive logic programming* (Master's thesis). Electrical Engineering and Computer Science, University of Ljubljana, Ljubljana, Slovenia.
- Fleiner, P. (1994). *Logic program synthesis from incomplete information*. The Kluwer international series in engineering and computer science. Boston: Kluwer Academic Publishers.
- Grobelnik, M. (1992). MARKUS: An optimized model inference system. *Proceedings of the Workshop on Logical Approaches to Machine Learning, Tenth European Conference on AI*, Vienna, Austria.
- Jevons, W. S. (1874). *The principles of science: A treatise on logic and scientific method*. London: Macmillan.
- Kietz, J.-U., & Wrobel, S. (1992). Controlling the complexity of learning in logic through syntactic and task-oriented models. In S. Muggleton (Ed.), *Inductive logic programming*. London: Academic Press.
- Lavrač, N., & Džeroski, S. (1994). *Inductive logic programming: Techniques and applications*. Ellis Horwood.
- Lindsay, R. K., Buchanan, B. G., Feigenbaum, E. A., & Lederberg, J. (1980). *Applications of artificial intelligence for organic chemistry*. New York: McGraw-Hill.
- Michalski, R. S., (1969). On the quasi-minimal solution of the general covering problem. *Proceedings of the First International Symposium on Information Processing* (pp. 125–128). Bled, Yugoslavia.
- Michalski, R. S., Mozetic, I., Hong, J., and Lavrac, H. (1986). The multi-purpose incremental learning system AQ15 and its testing application to three medical domains. *Proceedings of the Fifth National Conference on AI* (pp. 1041–1045). Philadelphia: Morgan-Kaufmann.
- Mitchell, T. M. (1979). *Version spaces: An approach to concept learning* (Ph.D. dissertation). Electrical Engineering Dept., Stanford University, Stanford, CA.
- Morik, K., Wrobel, S., Kietz, J.-U., & Emde, W. (1993). *Knowledge acquisition and machine learning: Theory, methods, and applications*. London: Academic Press.
- Muggleton, S. (1987). DUCE: An oracle based approach to constructive induction. *Proceedings of the International Joint Conference on AI* (pp. 287–292). San Mateo, CA: Morgan Kaufmann.
- Muggleton, S. (1995a). Inverse entailment and PROGOL. *New Generation Computing*, 13, 245–286.
- Muggleton, S. (1995b). *Foundations of inductive logic programming*. Englewood Cliffs, NJ: Prentice Hall.
- Muggleton, S., & Buntine, W. (1988). Machine invention of first-order predicates by inverting resolution. *Proceedings of the Fifth International Machine Learning Conference* (pp. 339–352). Ann Arbor, Michigan: Morgan Kaufmann.
- Muggleton, S., & Feng, C. (1990). Efficient induction of logic programs. *Proceedings of the First Conference on Algorithmic Learning Theory*. Ohmsha, Tokyo.
- Muggleton, S., & Feng, C. (1992). Efficient induction of logic programs. In Muggleton (Ed.), *Inductive logic programming*. London: Academic Press.
- Muggleton, S. (Ed.). (1992). *Inductive logic programming*. London: Academic Press.
- Pazzani, M., Brunk, C., & Silverstein, G. (1991). A knowledge-intensive approach to learning relational concepts. *Proceedings of the Eighth International Workshop on Machine Learning* (pp. 432–436). San Francisco: Morgan Kaufmann.
- Plotkin, G. D. (1970). A note on inductive generalization. In B. Meltzer & D. Michie (Eds.), *Machine Intelligence 5* (pp. 153–163). Edinburgh University Press.
- Plotkin, G. D. (1971). A further note on inductive generalization. In B. Meltzer & D. Michie (Eds.), *Machine Intelligence 6*. New York: Elsevier.
- Quinlan, J. R. (1990). Learning logical definitions from relations. *Machine Learning*, 5, 239–266.

- Quinlan, J. R. (1991). *Improved estimates for the accuracy of small disjuncts* (Technical Note). *Machine Learning*, 6(1), 93–98. Boston: Kluwer Academic Publishers.
- Rivest R. L. (1987). Learning decision lists. *Machine Learning*, 2(3), 229–246.
- Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1), 23–41.
- Sammut, C. A. (1981). Concept learning by experiment. *Seventh International Joint Conference on Artificial Intelligence*, Vancouver.
- Sammut, C. A., & Banerji, R. B. (1986). Learning concepts by asking questions. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach* (Vol 2, pp. 167–192). Los Altos, California: Morgan Kaufmann.
- Shapiro, E. (1983). *Algorithmic program debugging*. Cambridge MA: MIT Press.
- Srinivasan, A., Muggleton, S., & King, R. D. (1995). *Comparing the use of background knowledge by inductive logic programming systems* (PRG Technical report PRG-TR-9-95). Oxford University Computing Laboratory.
- Srinivasan, A., Muggleton, S., King, R. D., & Sternberg, M. J. E. (1994). Mutagenesis: ILP experiments in a non-determinate biological domain. *Proceedings of the Fourth Inductive Logic Programming Workshop*.
- Vere, S. (1975). Induction of concepts in the predicate calculus. *Proceedings of the Fourth International Joint Conference on Artificial Intelligence* (pp. 351–356).
- Winston, P. (1970). *Learning structural descriptions from examples* (Ph.D. dissertation) (MIT Technical Report AI-TR-231).
- Wrobel, S. (1994). *Concept formation and knowledge revision*. Boston: Kluwer Academic Publishers.
- Wrobel, S. (1996). Inductive logic programming. In G. Brewka (Ed.), *Principles of knowledge representation*. Stanford, CA: CSLI Publications.

CHAPTER

11

ANALYTICAL LEARNING

Inductive learning methods such as neural network and decision tree learning require a certain number of training examples to achieve a given level of generalization accuracy, as reflected in the theoretical bounds and experimental results discussed in earlier chapters. Analytical learning uses prior knowledge and deductive reasoning to augment the information provided by the training examples, so that it is not subject to these same bounds. This chapter considers an analytical learning method called explanation-based learning (EBL). In explanation-based learning, prior knowledge is used to analyze, or explain, how each observed training example satisfies the target concept. This explanation is then used to distinguish the relevant features of the training example from the irrelevant, so that examples can be generalized based on logical rather than statistical reasoning. Explanation-based learning has been successfully applied to learning search control rules for a variety of planning and scheduling tasks. This chapter considers explanation-based learning when the learner's prior knowledge is correct and complete. The next chapter considers combining inductive and analytical learning in situations where prior knowledge is only approximately correct.

11.1 INTRODUCTION

Previous chapters have considered a variety of *inductive* learning methods: that is, methods that generalize from observed training examples by identifying features that empirically distinguish positive from negative training examples. Decision tree learning, neural network learning, inductive logic programming, and genetic

algorithms are all examples of inductive methods that operate in this fashion. The key practical limit on these inductive learners is that they perform poorly when insufficient data is available. In fact, as discussed in Chapter 7, theoretical analysis shows that there are fundamental bounds on the accuracy that can be achieved when learning inductively from a given number of training examples.

Can we develop learning methods that are not subject to these fundamental bounds on learning accuracy imposed by the amount of training data available? Yes, if we are willing to reconsider the formulation of the learning problem itself. One way is to develop learning algorithms that accept explicit prior knowledge as an input, in addition to the input training data. Explanation-based learning is one such approach. It uses prior knowledge to analyze, or explain, each training example in order to infer which example features are relevant to the target function and which are irrelevant. These explanations enable it to generalize more accurately than inductive systems that rely on the data alone. As we saw in the previous chapter, inductive logic programming systems such as CIGOL also use prior background knowledge to guide learning. However, they use their background knowledge to infer features that augment the input descriptions of instances, thereby increasing the complexity of the hypothesis space to be searched. In contrast, explanation-based learning uses prior knowledge to *reduce* the complexity of the hypothesis space to be searched, thereby reducing sample complexity and improving generalization accuracy of the learner.

To capture the intuition underlying explanation-based learning, consider the task of learning to play chess. In particular, suppose we would like our chess program to learn to recognize important classes of game positions, such as the target concept “chessboard positions in which black will lose its queen within two moves.” Figure 11.1 shows a positive training example of this target concept. Inductive learning methods could, of course, be employed to learn this target concept. However, because the chessboard is fairly complex (there are 32 pieces that may be on any of 64 squares), and because the particular patterns that capture this concept are fairly subtle (involving the relative positions of various pieces on the board), we would have to provide thousands of training examples similar to the one in Figure 11.1 to expect an inductively learned hypothesis to generalize correctly to new situations.

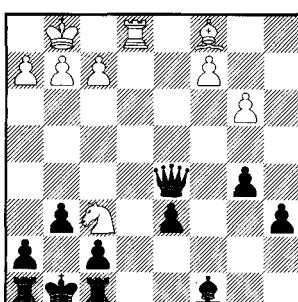


FIGURE 11.1

A positive example of the target concept “chess positions in which black will lose its queen within two moves.” Note the white knight is simultaneously attacking both the black king and queen. Black must therefore move its king, enabling white to capture its queen.

What is interesting about this chess-learning task is that humans appear to learn such target concepts from just a handful of training examples! In fact, after considering only the single example shown in Figure 11.1, most people would be willing to suggest a general hypothesis for the target concept, such as “board positions in which the black king and queen are simultaneously attacked,” and would not even consider the (equally consistent) hypothesis “board positions in which four white pawns are still in their original locations.” How is it that humans can generalize so successfully from just this one example?

The answer appears to be that people rely heavily on explaining, or analyzing, the training example in terms of their prior knowledge about the legal moves of chess. If asked to explain why the training example of Figure 11.1 is a positive example of “positions in which the queen will be lost in two moves,” most people would give an explanation similar to the following: “Because white’s knight is attacking both the king and queen, black must move out of check, thereby allowing the knight to capture the queen.” The importance of such explanations is that they provide the information needed to *rationally* generalize from the details of the training example to a correct general hypothesis. Features of the training example that are mentioned by the explanation (e.g., the position of the white knight, black king, and black queen) are relevant to the target concept and should be included in the general hypothesis. In contrast, features of the example that are not mentioned by the explanation (e.g., the fact that there are six black pawns on the board) can be assumed to be irrelevant details.

What exactly is the prior knowledge needed by a learner to construct the explanation in this chess example? It is simply knowledge about the legal rules of chess: knowledge of which moves are legal for the knight and other pieces, the fact that players must alternate moves in the game, and the fact that to win the game one player must capture his opponent’s king. Note that given just this prior knowledge it is possible *in principle* to calculate the optimal chess move for any board position. However, in practice this calculation can be frustratingly complex and despite the fact that we humans ourselves possess this complete, perfect knowledge of chess, we remain unable to play the game optimally. As a result, much of human learning in chess (and in other search-intensive problems such as scheduling and planning) involves a long process of uncovering the consequences of our prior knowledge, guided by specific training examples encountered as we play the game.

This chapter describes learning algorithms that automatically construct and learn from such explanations. In the remainder of this section we define more precisely the analytical learning problem. The next section presents a particular explanation-based learning algorithm called PROLOG-EBG. Subsequent sections then examine the general properties of this algorithm and its relationship to inductive learning algorithms discussed in other chapters. The final section describes the application of explanation-based learning to improving performance at large state-space search problems. In this chapter we consider the special case in which explanations are generated from prior knowledge that is perfectly correct, as it is for us humans in the above chess example. In Chapter 12 we consider the more general case of learning when prior knowledge is only approximately correct.

11.1.1 Inductive and Analytical Learning Problems

The essential difference between analytical and inductive learning methods is that they assume two different formulations of the learning problem:

- In inductive learning, the learner is given a hypothesis space H from which it must select an output hypothesis, and a set of training examples $D = \{(x_1, f(x_1)), \dots, (x_n, f(x_n))\}$ where $f(x_i)$ is the target value for the instance x_i . The desired output of the learner is a hypothesis h from H that is consistent with these training examples.
- In analytical learning, the input to the learner includes the same hypothesis space H and training examples D as for inductive learning. In addition, the learner is provided an additional input: A *domain theory* B consisting of background knowledge that can be used to explain observed training examples. The desired output of the learner is a hypothesis h from H that is consistent with both the training examples D and the domain theory B .

To illustrate, in our chess example each instance x_i would describe a particular chess position, and $f(x_i)$ would be *True* when x_i is a position for which black will lose its queen within two moves, and *False* otherwise. We might define the hypothesis space H to consist of sets of Horn clauses (if-then rules) as in Chapter 10, where the predicates used by the rules refer to the positions or relative positions of specific pieces on the board. The domain theory B would consist of a formalization of the rules of chess, describing the legal moves, the fact that players must take turns, and the fact that the game is won when one player captures her opponent's king.

Note in analytical learning, the learner must output a hypothesis that is consistent with *both* the training data and the domain theory. We say that hypothesis h is *consistent* with domain theory B provided B does not entail the negation of h (i.e., $B \not\vdash \neg h$). This additional constraint that the output hypothesis must be consistent with B reduces the ambiguity faced by the learner when the data alone cannot resolve among all hypotheses in H . The net effect, provided the domain theory is correct, is to increase the accuracy of the output hypothesis.

Let us introduce in detail a second example of an analytical learning problem—one that we will use for illustration throughout this chapter. Consider an instance space X in which each instance is a pair of physical objects. Each of the two physical objects in the instance is described by the predicates *Color*, *Volume*, *Owner*, *Material*, *Type*, and *Density*, and the relationship between the two objects is described by the predicate *On*. Given this instance space, the task is to learn the target concept “pairs of physical objects, such that one can be stacked safely on the other,” denoted by the predicate *SafeToStack*(x,y). Learning this target concept might be useful, for example, to a robot system that has the task of storing various physical objects within a limited workspace. The full definition of this analytical learning task is given in Table 11.1.

Given:

- Instance space X : Each instance describes a pair of objects represented by the predicates *Type*, *Color*, *Volume*, *Owner*, *Material*, *Density*, and *On*.
- Hypothesis space H : Each hypothesis is a set of Horn clause rules. The head of each Horn clause is a literal containing the target predicate *SafeToStack*. The body of each Horn clause is a conjunction of literals based on the same predicates used to describe the instances, as well as the predicates *LessThan*, *Equal*, *GreaterThan*, and the functions *plus*, *minus*, and *times*. For example, the following Horn clause is in the hypothesis space:

$$\text{SafeToStack}(x, y) \leftarrow \text{Volume}(x, vx) \wedge \text{Volume}(y, vy) \wedge \text{LessThan}(vx, vy)$$

- Target concept: $\text{SafeToStack}(x, y)$
- Training Examples: A typical positive example, $\text{SafeToStack}(\text{Obj1}, \text{Obj2})$, is shown below:

$\text{On}(\text{Obj1}, \text{Obj2})$	$\text{Owner}(\text{Obj1}, \text{Fred})$
$\text{Type}(\text{Obj1}, \text{Box})$	$\text{Owner}(\text{Obj2}, \text{Louise})$
$\text{Type}(\text{Obj2}, \text{Endtable})$	$\text{Density}(\text{Obj1}, 0.3)$
$\text{Color}(\text{Obj1}, \text{Red})$	$\text{Material}(\text{Obj1}, \text{Cardboard})$
$\text{Color}(\text{Obj2}, \text{Blue})$	$\text{Material}(\text{Obj2}, \text{Wood})$
$\text{Volume}(\text{Obj1}, 2)$	

- Domain Theory B :

$\text{SafeToStack}(x, y) \leftarrow \neg \text{Fragile}(y)$
$\text{SafeToStack}(x, y) \leftarrow \text{Lighter}(x, y)$
$\text{Lighter}(x, y) \leftarrow \text{Weight}(x, wx) \wedge \text{Weight}(y, wy) \wedge \text{LessThan}(wx, wy)$
$\text{Weight}(x, w) \leftarrow \text{Volume}(x, v) \wedge \text{Density}(x, d) \wedge \text{Equal}(w, \text{times}(v, d))$
$\text{Weight}(x, 5) \leftarrow \text{Type}(x, \text{Endtable})$
$\text{Fragile}(x) \leftarrow \text{Material}(x, \text{Glass})$
...

Determine:

- A hypothesis from H consistent with the training examples and domain theory.

TABLE 11.1

An analytical learning problem: $\text{SafeToStack}(x, y)$.

As shown in Table 11.1, we have chosen a hypothesis space H in which each hypothesis is a set of first-order if-then rules, or Horn clauses (throughout this chapter we follow the notation and terminology for first-order Horn clauses summarized in Table 10.3). For instance, the example Horn clause hypothesis shown in the table asserts that it is *SafeToStack* any object x on any object y , if the *Volume* of x is *LessThan* the *Volume* of y (in this Horn clause the variables vx and vy represent the volumes of x and y , respectively). Note the Horn clause hypothesis can refer to any of the predicates used to describe the instances, as well as several additional predicates and functions. A typical positive training example, $\text{SafeToStack}(\text{Obj1}, \text{Obj2})$, is also shown in the table.

To formulate this task as an analytical learning problem we must also provide a domain theory sufficient to explain why observed positive examples satisfy the target concept. In our earlier chess example, the domain theory corresponded to knowledge of the legal moves in chess, from which we constructed explanations

describing why black would lose its queen. In the current example, the domain theory must similarly explain why certain pairs of objects can be safely stacked on one another. The domain theory shown in the table includes assertions such as “it is safe to stack x on y if y is not *Fragile*,” and “an object x is *Fragile* if the *Material* from which x is made is *Glass*. ” Like the learned hypothesis, the domain theory is described by a collection of Horn clauses, enabling the system in principle to incorporate any learned hypotheses into subsequent domain theories. Notice that the domain theory refers to additional predicates such as *Lighter* and *Fragile*, which are not present in the descriptions of the training examples, but which can be inferred from more primitive instance attributes such as *Material*, *Density*, and *Volume*, using other rules in the domain theory. Finally, notice that the domain theory shown in the table is sufficient to prove that the positive example shown there satisfies the target concept *SafeToStack*.

11.2 LEARNING WITH PERFECT DOMAIN THEORIES: PROLOG-EBG

As stated earlier, in this chapter we consider explanation-based learning from domain theories that are perfect, that is, domain theories that are correct and complete. A domain theory is said to be *correct* if each of its assertions is a truthful statement about the world. A domain theory is said to be *complete* with respect to a given target concept and instance space, if the domain theory covers every positive example in the instance space. Put another way, it is complete if every instance that satisfies the target concept can be proven by the domain theory to satisfy it. Notice our definition of completeness does not require that the domain theory be able to prove that negative examples do not satisfy the target concept. However, if we follow the usual PROLOG convention that unprovable assertions are assumed to be false, then this definition of completeness includes full coverage of both positive and negative examples by the domain theory.

The reader may well ask at this point whether it is reasonable to assume that such perfect domain theories are available to the learner. After all, if the learner had a perfect domain theory, why would it need to learn? There are two responses to this question.

- First, there are cases in which it is feasible to provide a perfect domain theory. Our earlier chess problem provides one such case, in which the legal moves of chess form a perfect domain theory from which the optimal chess playing strategy can (in principle) be inferred. Furthermore, although it is quite easy to write down the legal moves of chess that constitute this domain theory, it is extremely difficult to write down the optimal chess-playing strategy. In such cases, we prefer to provide the domain theory to the learner and rely on the learner to formulate a useful description of the target concept (e.g., “board states in which I am about to lose my queen”) by examining and generalizing from specific training examples. Section 11.4 describes the successful application of explanation-based learning with perfect domain

theories to automatically improve performance at several search-intensive planning and optimization problems.

- Second, in many other cases it is unreasonable to assume that a perfect domain theory is available. It is difficult to write a perfectly correct and complete theory even for our relatively simple *SafeToStack* problem. A more realistic assumption is that plausible explanations based on imperfect domain theories must be used, rather than exact proofs based on perfect knowledge. Nevertheless, we can begin to understand the role of explanations in learning by considering the ideal case of perfect domain theories. In Chapter 12 we will consider learning from imperfect domain theories.

This section presents an algorithm called PROLOG-EBG (Kedar-Cabelli and McCarty 1987) that is representative of several explanation-based learning algorithms. PROLOG-EBG is a sequential covering algorithm (see Chapter 10). In other words, it operates by learning a single Horn clause rule, removing the positive training examples covered by this rule, then iterating this process on the remaining positive examples until no further positive examples remain uncovered. When given a complete and correct domain theory, PROLOG-EBG is guaranteed to output a hypothesis (set of rules) that is itself correct and that covers the observed positive training examples. For any set of training examples, the hypothesis output by PROLOG-EBG constitutes a set of logically sufficient conditions for the target concept, according to the domain theory. PROLOG-EBG is a refinement of the EBG algorithm introduced by Mitchell et al. (1986) and is similar to the EGGS algorithm described by DeJong and Mooney (1986). The PROLOG-EBG algorithm is summarized in Table 11.2.

11.2.1 An Illustrative Trace

To illustrate, consider again the training example and domain theory shown in Table 11.1. As summarized in Table 11.2, the PROLOG-EBG algorithm is a sequential covering algorithm that considers the training data incrementally. For each new positive training example that is not yet covered by a learned Horn clause, it forms a new Horn clause by: (1) explaining the new positive training example, (2) analyzing this explanation to determine an appropriate generalization, and (3) refining the current hypothesis by adding a new Horn clause rule to cover this positive example, as well as other similar instances. Below we examine each of these three steps in turn.

11.2.1.1 EXPLAIN THE TRAINING EXAMPLE

The first step in processing each novel training example is to construct an explanation in terms of the domain theory, showing how this positive example satisfies the target concept. When the domain theory is correct and complete this explanation constitutes a *proof* that the training example satisfies the target concept. When dealing with imperfect prior knowledge, the notion of explanation must be extended to allow for plausible, approximate arguments rather than perfect proofs.

PROLOG-EBG(*TargetConcept*, *TrainingExamples*, *DomainTheory*)

- $\text{LearnedRules} \leftarrow \{\}$
- $\text{Pos} \leftarrow$ the positive examples from *TrainingExamples*
- for each *PositiveExample* in *Pos* that is not covered by *LearnedRules*, do
 1. *Explain*:
 - $\text{Explanation} \leftarrow$ an explanation (proof) in terms of the *DomainTheory* that *PositiveExample* satisfies the *TargetConcept*
 2. *Analyze*:
 - $\text{SufficientConditions} \leftarrow$ the most general set of features of *PositiveExample* sufficient to satisfy the *TargetConcept* according to the *Explanation*.
 3. *Refine*:
 - $\text{LearnedRules} \leftarrow \text{LearnedRules} + \text{NewHornClause}$, where *NewHornClause* is of the form

$$\text{TargetConcept} \leftarrow \text{SufficientConditions}$$
- Return *LearnedRules*

TABLE 11.2

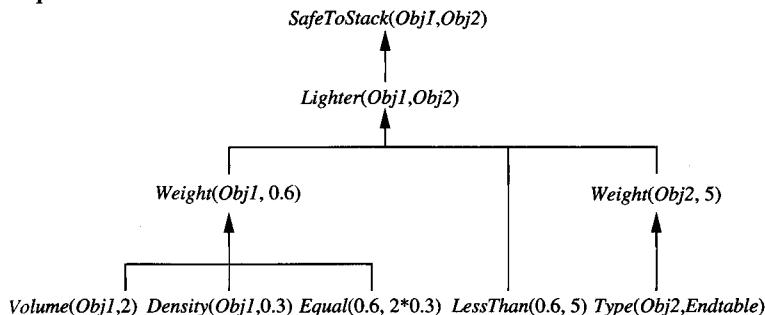
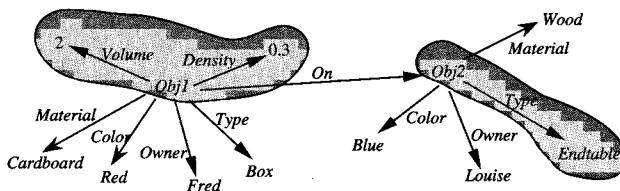
The explanation-based learning algorithm PROLOG-EBG. For each positive example that is not yet covered by the set of learned Horn clauses (*LearnedRules*), a new Horn clause is created. This new Horn clause is created by (1) explaining the training example in terms of the domain theory, (2) analyzing this explanation to determine the relevant features of the example, then (3) constructing a new Horn clause that concludes the target concept when this set of features is satisfied.

The explanation for the current training example is shown in Figure 11.2. Note the bottom of this figure depicts in graphical form the positive training example *SafeToStack(Obj1, Obj2)* from Table 11.1. The top of the figure depicts the explanation constructed for this training example. Notice the explanation, or proof, states that it is *SafeToStack Obj1* on *Obj2* because *Obj1* is *Lighter* than *Obj2*. Furthermore, *Obj1* is known to be *Lighter*, because its *Weight* can be inferred from its *Density* and *Volume*, and because the *Weight* of *Obj2* can be inferred from the default weight of an *Endtable*. The specific Horn clauses that underlie this explanation are shown in the domain theory of Table 11.1. Notice that the explanation mentions only a small fraction of the known attributes of *Obj1* and *Obj2* (i.e., those attributes corresponding to the shaded region in the figure).

While only a single explanation is possible for the training example and domain theory shown here, in general there may be multiple possible explanations. In such cases, any or all of the explanations may be used. While each may give rise to a somewhat different generalization of the training example, all will be justified by the given domain theory. In the case of PROLOG-EBG, the explanation is generated using a backward chaining search as performed by PROLOG. PROLOG-EBG, like PROLOG, halts once it finds the first valid proof.

11.2.1.2 ANALYZE THE EXPLANATION

The key question faced in generalizing the training example is “of the many features that happen to be true of the current training example, which ones are gen-

Explanation:**Training Example:****FIGURE 11.2**

Explanation of a training example. The network at the bottom depicts graphically the training example $\text{SafeToStack}(\text{Obj1}, \text{Obj2})$ described in Table 11.1. The top portion of the figure depicts the explanation of how this example satisfies the target concept, SafeToStack . The shaded region of the training example indicates the example attributes used in the explanation. The other, irrelevant, example attributes will be dropped from the generalized hypothesis formed from this analysis.

erally relevant to the target concept?" The explanation constructed by the learner provides a direct answer to this question: precisely those features mentioned in the explanation. For example, the explanation of Figure 11.2 refers to the *Density* of *Obj1*, but not to its *Owner*. Therefore, the hypothesis for $\text{SafeToStack}(x, y)$ should include $\text{Density}(x, 0.3)$, but not $\text{Owner}(x, \text{Fred})$. By collecting just the features mentioned in the leaf nodes of the explanation in Figure 11.2 and substituting variables *x* and *y* for *Obj1* and *Obj2*, we can form a general rule that is justified by the domain theory:

$$\text{SafeToStack}(x, y) \leftarrow \text{Volume}(x, 2) \wedge \text{Density}(x, 0.3) \wedge \text{Type}(y, \text{Endtable})$$

The body of the above rule includes each leaf node in the proof tree, except for the leaf nodes "*Equal*(0.6, *times*(2, 0.3))" and "*LessThan*(0.6, 5)." We omit these two because they are by definition always satisfied, independent of *x* and *y*.

Along with this learned rule, the program can also provide its justification: The explanation of the training example forms a proof for the correctness of this rule. Although this explanation was formed to cover the observed training example, the same explanation will apply to any instance that matches this general rule.

The above rule constitutes a significant generalization of the training example, because it omits many properties of the example (e.g., the *Color* of the two objects) that are irrelevant to the target concept. However, an even more general rule can be obtained by more careful analysis of the explanation. PROLOG-EBG computes the most general rule that can be justified by the explanation, by computing the *weakest preimage* of the explanation, defined as follows:

Definition: The **weakest preimage** of a conclusion C with respect to a proof P is the most general set of initial assertions A , such that A entails C according to P .

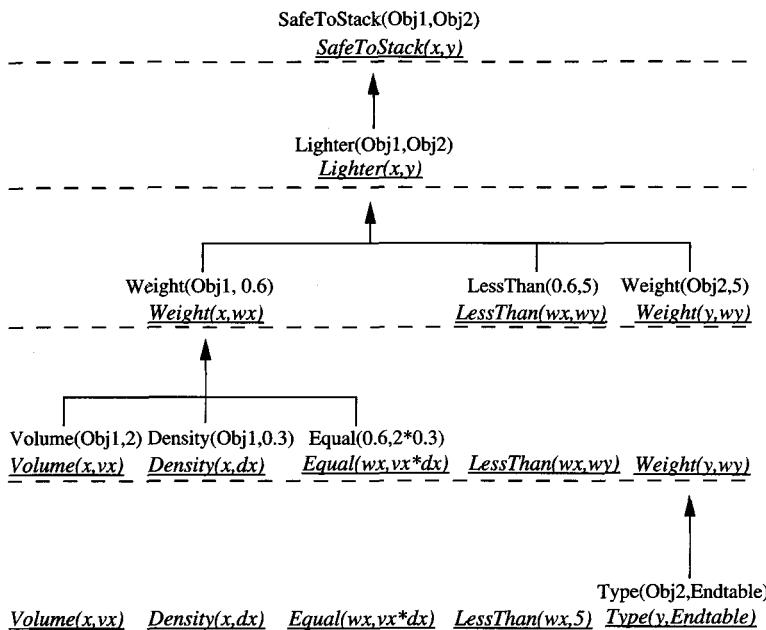
For example, the weakest preimage of the target concept $\text{SafeToStack}(x, y)$, with respect to the explanation from Table 11.1, is given by the body of the following rule. This is the most general rule that can be justified by the explanation of Figure 11.2:

$$\begin{aligned} \text{SafeToStack}(x, y) \leftarrow & \text{Volume}(x, vx) \wedge \text{Density}(x, dx) \wedge \\ & \text{Equal}(wx, \text{times}(vx, dx)) \wedge \text{LessThan}(wx, 5) \wedge \\ & \text{Type}(y, \text{Endtable}) \end{aligned}$$

Notice this more general rule does not require the specific values for *Volume* and *Density* that were required by the first rule. Instead, it states a more general constraint on the values of these attributes.

PROLOG-EBG computes the weakest preimage of the target concept with respect to the explanation, using a general procedure called *regression* (Waldinger 1977). The regression procedure operates on a domain theory represented by an arbitrary set of Horn clauses. It works iteratively backward through the explanation, first computing the weakest preimage of the target concept with respect to the final proof step in the explanation, then computing the weakest preimage of the resulting expressions with respect to the preceding step, and so on. The procedure terminates when it has iterated over all steps in the explanation, yielding the weakest precondition of the target concept with respect to the literals at the leaf nodes of the explanation.

A trace of this regression process is illustrated in Figure 11.3. In this figure, the explanation from Figure 11.2 is redrawn in standard (nonitalic) font. The frontier of regressed expressions created at each step by the regression procedure is shown underlined in italicics. The process begins at the root of the tree, with the frontier initialized to the general target concept $\text{SafeToStack}(x, y)$. The first step is to compute the weakest preimage of this frontier expression with respect to the final (top-most) inference rule in the explanation. The rule in this case is $\text{SafeToStack}(x, y) \leftarrow \text{Lighter}(x, y)$, so the resulting weakest preimage is $\text{Lighter}(x, y)$. The process now continues by regressing the new frontier, $\{\text{Lighter}(x, y)\}$, through the next Horn clause in the explanation, resulting in the regressed expressions $\{\text{Weight}(x, wx), \text{LessThan}(wx, wy), \text{Weight}(y, wy)\}$. This indicates that the explanation will hold for any x and y such that the weight wx of x is less than the weight wy of y . The regression of this frontier back to the leaf nodes of the explanation continues in this step-by-step fashion, finally

**FIGURE 11.3**

Computing the weakest preimage of $\text{SafeToStack}(\text{Obj1}, \text{Obj2})$ with respect to the explanation. The target concept is regressed from the root (conclusion) of the explanation, down to the leaves. At each step (indicated by the dashed lines) the current frontier set of literals (underlined in italics) is regressed backward over one rule in the explanation. When this process is completed, the conjunction of resulting literals constitutes the weakest preimage of the target concept with respect to the explanation. This weakest preimage is shown by the italicized literals at the bottom of the figure.

resulting in a set of generalized literals for the leaf nodes of the tree. This final set of literals, shown at the bottom of Figure 11.3, forms the body of the final rule.

The heart of the regression procedure is the algorithm that at each step regresses the current frontier of expressions through a single Horn clause from the domain theory. This algorithm is described and illustrated in Table 11.3. The illustrated example in this table corresponds to the bottommost single regression step of Figure 11.3. As shown in the table, the REGRESS algorithm operates by finding a substitution that unifies the head of the Horn clause rule with the corresponding literal in the frontier, replacing this expression in the frontier by the rule body, then applying a unifying substitution to the entire frontier.

The final Horn clause rule output by PROLOG-EBG is formulated as follows: The clause body is defined to be the weakest preconditions calculated by the above procedure. The clause head is the target concept itself, with each substitution from each regression step (i.e., the substitution θ_{hl} in Table 11.3) applied to it. This substitution is necessary in order to keep consistent variable names between the head and body of the created clause, and to specialize the clause head when the

REGRESS(*Frontier*, *Rule*, *Literal*, θ_{hi})

Frontier: Set of literals to be regressed through *Rule*

Rule: A Horn clause

Literal: A literal in *Frontier* that is inferred by *Rule* in the explanation

θ_{hi} : The substitution that unifies the head of *Rule* to the corresponding literal in the explanation

Returns the set of literals forming the weakest preimage of *Frontier* with respect to *Rule*

- $head \leftarrow head$ of *Rule*
- $body \leftarrow body$ of *Rule*
- $\theta_{hl} \leftarrow$ the most general unifier of *head* with *Literal* such that there exists a substitution θ_{li} for which

$$\theta_{li}(\theta_{hi}(head)) = \theta_{hi}(head)$$

- Return $\theta_{hl}(Frontier - head + body)$
-

Example (the bottommost regression step in Figure 11.3):

REGRESS(*Frontier*, *Rule*, *Literal*, θ_{hi}) where

Frontier = {*Volume*(*x*, *vs*), *Density*(*x*, *dx*), *Equal*(*wx*, *times*(*vx*, *dx*)), *LessThan*(*wx*, *wy*), *Weight*(*y*, *wy*)}
Rule = *Weight*(*z*, 5) \leftarrow *Type*(*z*, *Endtable*)

Literal = *Weight*(*y*, *wy*)

θ_{hi} = {*z*/*Obj* 2}

- $head \leftarrow Weight(z, 5)$
 - $body \leftarrow Type(z, Endtable)$
 - $\theta_{hl} \leftarrow \{z/y, wy/5\}$, where $\theta_{li} = \{y/Obj\ 2\}$
 - Return {*Volume*(*x*, *vs*), *Density*(*x*, *dx*), *Equal*(*wx*, *times*(*vx*, *dx*)), *LessThan*(*wx*, 5), *Type*(*y*, *Endtable*)}
-

TABLE 11.3

Algorithm for regressing a set of literals through a single Horn clause. The set of literals given by *Frontier* is regressed through *Rule*. *Literal* is the member of *Frontier* inferred by *Rule* in the explanation. The substitution θ_{hi} gives the binding of variables from the head of *Rule* to the corresponding literal in the explanation. The algorithm first computes a substitution θ_{hl} that unifies the *Rule* head to *Literal*, in a way that is consistent with the substitution θ_{hi} . It then applies this substitution θ_{hl} to construct the preimage of *Frontier* with respect to *Rule*. The symbols “+” and “-” in the algorithm denote set union and set difference. The notation {*z*/*y*} denotes the substitution of *y* in place of *z*. An example trace is given.

explanation applies to only a special case of the target concept. As noted earlier, for the current example the final rule is

$$\begin{aligned} SafeToStack(x, y) \leftarrow & Volume(x, vx) \wedge Density(x, dx) \wedge \\ & Equal(wx, times(vx, dx)) \wedge LessThan(wx, 5) \wedge \\ & Type(y, Endtable) \end{aligned}$$

11.2.1.3 REFINING THE CURRENT HYPOTHESIS

The current hypothesis at each stage consists of the set of Horn clauses learned thus far. At each stage, the sequential covering algorithm picks a new positive

example that is not yet covered by the current Horn clauses, explains this new example, and formulates a new rule according to the procedure described above. Notice only positive examples are covered in the algorithm as we have defined it, and the learned set of Horn clause rules predicts only positive examples. A new instance is classified as negative if the current rules fail to predict that it is positive. This is in keeping with the standard negation-as-failure approach used in Horn clause inference systems such as PROLOG.

11.3 REMARKS ON EXPLANATION-BASED LEARNING

As we saw in the above example, PROLOG-EBG conducts a detailed analysis of individual training examples to determine how best to generalize from the specific example to a general Horn clause hypothesis. The following are the key properties of this algorithm.

- Unlike inductive methods, PROLOG-EBG produces *justified* general hypotheses by using prior knowledge to analyze individual examples.
- The explanation of how the example satisfies the target concept determines which example attributes are relevant: those mentioned by the explanation.
- The further analysis of the explanation, regressing the target concept to determine its weakest preimage with respect to the explanation, allows deriving more general constraints on the values of the relevant features.
- Each learned Horn clause corresponds to a sufficient condition for satisfying the target concept. The set of learned Horn clauses covers the positive training examples encountered by the learner, as well as other instances that share the same explanations.
- The generality of the learned Horn clauses will depend on the formulation of the domain theory and on the sequence in which training examples are considered.
- PROLOG-EBG implicitly assumes that the domain theory is correct and complete. If the domain theory is incorrect or incomplete, the resulting learned concept may also be incorrect.

There are several related perspectives on explanation-based learning that help to understand its capabilities and limitations.

- *EBL as theory-guided generalization of examples.* EBL uses its given domain theory to generalize *rationally* from examples, distinguishing the relevant example attributes from the irrelevant, thereby allowing it to avoid the bounds on sample complexity that apply to purely inductive learning. This is the perspective implicit in the above description of the PROLOG-EBG algorithm.
- *EBL as example-guided reformulation of theories.* The PROLOG-EBG algorithm can be viewed as a method for reformulating the domain theory into a more operational form. In particular, the original domain theory is reformulated by creating rules that (a) follow deductively from the domain theory,

and (b) classify the observed training examples in a single inference step. Thus, the learned rules can be seen as a reformulation of the domain theory into a set of special-case rules capable of classifying instances of the target concept in a single inference step.

- *EBL as “just” restating what the learner already “knows.”* In one sense, the learner in our *SafeToStack* example begins with full knowledge of the *SafeToStack* concept. That is, if its initial domain theory is sufficient to explain any observed training examples, then it is also sufficient to predict their classification in advance. In what sense, then, does this qualify as learning? One answer is that in many tasks the difference between what one knows *in principle* and what one can efficiently compute *in practice* may be great, and in such cases this kind of “knowledge reformulation” can be an important form of learning. In playing chess, for example, the rules of the game constitute a perfect domain theory, sufficient in principle to play perfect chess. Despite this fact, people still require considerable experience to learn how to play chess well. This is precisely a situation in which a complete, perfect domain theory is already known to the (human) learner, and further learning is “simply” a matter of reformulating this knowledge into a form in which it can be used more effectively to select appropriate moves. A beginning course in Newtonian physics exhibits the same property—the basic laws of physics are easily stated, but students nevertheless spend a large part of a semester working out the consequences so they have this knowledge in more operational form and need not derive every problem solution from first principles come the final exam. PROLOG-EBG performs this type of reformulation of knowledge—its learned rules map directly from observable instance features to the classification relative to the target concept, in a way that is consistent with the underlying domain theory. Whereas it may require many inference steps and considerable search to classify an arbitrary instance using the original domain theory, the learned rules classify the observed instances in a single inference step.

Thus, in its pure form EBL involves reformulating the domain theory to produce general rules that classify examples in a single inference step. This kind of knowledge reformulation is sometimes referred to as *knowledge compilation*, indicating that the transformation is an efficiency improving one that does not alter the correctness of the system’s knowledge.

11.3.1 Discovering New Features

One interesting capability of PROLOG-EBG is its ability to formulate new features that are not explicit in the description of the training examples, but that are needed to describe the general rule underlying the training example. This capability is illustrated by the algorithm trace and the learned rule in the previous section. In particular, the learned rule asserts that the essential constraint on the *Volume* and *Density* of x is that their product is less than 5. In fact, the training examples

contain no description of such a product, or of the value it should take on. Instead, this constraint is formulated automatically by the learner.

Notice this learned “feature” is similar in kind to the types of features represented by the hidden units of neural networks; that is, this feature is one of a very large set of potential features that can be computed from the available instance attributes. Like the BACKPROPAGATION algorithm, PROLOG-EBG automatically formulates such features in its attempt to fit the training data. However, unlike the statistical process that derives hidden unit features in neural networks from many training examples, PROLOG-EBG employs an analytical process to derive new features based on analysis of single training examples. Above, PROLOG-EBG derives the feature $Volume \cdot Density > 5$ analytically from the particular instantiation of the domain theory used to explain a single training example. For example, the notion that the product of $Volume$ and $Density$ is important arises from the domain theory rule that defines $Weight$. The notion that this product should be less than 5 arises from two other domain theory rules that assert that $Obj1$ should be *Lighter* than the *Endtable*, and that the *Weight* of the *Endtable* is 5. Thus, it is the particular composition and instantiation of these primitive terms from the domain theory that gives rise to defining this new feature.

The issue of automatically learning useful features to augment the instance representation is an important issue for machine learning. The analytical derivation of new features in explanation-based learning and the inductive derivation of new features in the hidden layer of neural networks provide two distinct approaches. Because they rely on different sources of information (statistical regularities over many examples versus analysis of single examples using the domain theory), it may be useful to explore new methods that combine both sources.

11.3.2 Deductive Learning

In its pure form, PROLOG-EBG is a deductive, rather than inductive, learning process. That is, by calculating the weakest preimage of the explanation it produces a hypothesis h that follows deductively from the domain theory B , while covering the training data D . To be more precise, PROLOG-EBG outputs a hypothesis h that satisfies the following two constraints:

$$(\forall \langle x_i, f(x_i) \rangle \in D) \quad (h \wedge x_i) \vdash f(x_i) \quad (11.1)$$

$$D \wedge B \vdash h \quad (11.2)$$

where the training data D consists of a set of training examples in which x_i is the i th training instance and $f(x_i)$ is its target value (f is the target function). Notice the first of these constraints is simply a formalization of the usual requirement in machine learning, that the hypothesis h correctly predict the target value $f(x_i)$ for each instance x_i in the training data.[†] Of course there will, in general, be many

[†]Here we include PROLOG-style negation-by-failure in our definition of entailment (\vdash), so that examples are entailed to be negative examples if they cannot be proven to be positive.

alternative hypotheses that satisfy this first constraint. The second constraint describes the impact of the domain theory in PROLOG-EBL: The output hypothesis is further constrained so that it must follow from the domain theory and the data. This second constraint reduces the ambiguity faced by the learner when it must choose a hypothesis. Thus, the impact of the domain theory is to reduce the effective size of the hypothesis space and hence reduce the sample complexity of learning.

Using similar notation, we can state the type of knowledge that is required by PROLOG-EBG for its domain theory. In particular, PROLOG-EBG assumes the domain theory B entails the classifications of the instances in the training data:

$$(\forall \langle x_i, f(x_i) \rangle \in D) \quad (B \wedge x_i) \vdash f(x_i) \quad (11.3)$$

This constraint on the domain theory B assures that an explanation can be constructed for each positive example.

It is interesting to compare the PROLOG-EBG learning setting to the setting for inductive logic programming (ILP) discussed in Chapter 10. In that chapter, we discussed a generalization of the usual inductive learning task, in which background knowledge B' is provided to the learner. We will use B' rather than B to denote the background knowledge used by ILP, because it does not typically satisfy the constraint given by Equation (11.3). ILP is an inductive learning system, whereas PROLOG-EBG is deductive. ILP uses its background knowledge B' to enlarge the set of hypotheses to be considered, whereas PROLOG-EBG uses its domain theory B to reduce the set of acceptable hypotheses. As stated in Equation (10.2), ILP systems output a hypothesis h that satisfies the following constraint:

$$(\forall \langle x_i, f(x_i) \rangle \in D) \quad (B' \wedge h \wedge x_i) \vdash f(x_i)$$

Note the relationship between this expression and the constraints on h imposed by PROLOG-EBG (given by Equations (11.1) and (11.2)). This ILP constraint on h is a weakened form of the constraint given by Equation (11.1)—the ILP constraint requires only that $(B' \wedge h \wedge x_i) \vdash f(x_i)$, whereas the PROLOG-EBG constraint requires the more strict $(h \wedge x_i) \vdash f(x_i)$. Note also that ILP imposes no constraint corresponding to the PROLOG-EBG constraint of Equation (11.2).

11.3.3 Inductive Bias in Explanation-Based Learning

Recall from Chapter 2 that the inductive bias of a learning algorithm is a set of assertions that, together with the training examples, deductively entail subsequent predictions made by the learner. The importance of inductive bias is that it characterizes how the learner generalizes beyond the observed training examples.

What is the inductive bias of PROLOG-EBG? In PROLOG-EBG the output hypothesis h follows deductively from $D \wedge B$, as described by Equation (11.2). Therefore, the domain theory B is a set of assertions which, together with the training examples, entail the output hypothesis. Given that predictions of the learner follow from this hypothesis h , it appears that the inductive bias of PROLOG-EBG is simply the domain theory B input to the learner. In fact, this is the case except for one

additional detail that must be considered: There are many alternative sets of Horn clauses entailed by the domain theory. The remaining component of the inductive bias is therefore the basis by which PROLOG-EBG chooses among these alternative sets of Horn clauses. As we saw above, PROLOG-EBG employs a sequential covering algorithm that continues to formulate additional Horn clauses until all positive training examples have been covered. Furthermore, each individual Horn clause is the most general clause (weakest preimage) licensed by the explanation of the current training example. Therefore, among the sets of Horn clauses entailed by the domain theory, we can characterize the bias of PROLOG-EBG as a preference for small sets of maximally general Horn clauses. In fact, the greedy algorithm of PROLOG-EBG is only a heuristic approximation to the exhaustive search algorithm that would be required to find the truly shortest set of maximally general Horn clauses. Nevertheless, the inductive bias of PROLOG-EBG can be approximately characterized in this fashion.

Approximate inductive bias of PROLOG-EBG: The domain theory B , plus a preference for small sets of maximally general Horn clauses.

The most important point here is that the inductive bias of PROLOG-EBG—the policy by which it generalizes beyond the training data—is largely determined by the input domain theory. This lies in stark contrast to most of the other learning algorithms we have discussed (e.g., neural networks, decision tree learning), in which the inductive bias is a fixed property of the learning algorithm, typically determined by the syntax of its hypothesis representation. Why is it important that the inductive bias be an input parameter rather than a fixed property of the learner? Because, as we have discussed in Chapter 2 and elsewhere, there is no universally effective inductive bias and because bias-free learning is futile. Therefore, any attempt to develop a general-purpose learning method must at minimum allow the inductive bias to vary with the learning problem at hand. On a more practical level, in many tasks it is quite natural to input domain-specific knowledge (e.g., the knowledge about *Weight* in the *SafeToStack* example) to influence how the learner will generalize beyond the training data. In contrast, it is less natural to “implement” an appropriate bias by restricting the syntactic form of the hypotheses (e.g., prefer short decision trees). Finally, if we consider the larger issue of how an autonomous agent may improve its learning capabilities over time, then it is attractive to have a learning algorithm whose generalization capabilities improve as it acquires more knowledge of its domain.

11.3.4 Knowledge Level Learning

As pointed out in Equation (11.2), the hypothesis h output by PROLOG-EBG follows deductively from the domain theory B and training data D . In fact, by examining the PROLOG-EBG algorithm it is easy to see that h follows directly from B alone, independent of D . One way to see this is to imagine an algorithm that we might

call LEMMA-ENUMERATOR. The LEMMA-ENUMERATOR algorithm simply enumerates all proof trees that conclude the target concept based on assertions in the domain theory B . For each such proof tree, LEMMA-ENUMERATOR calculates the weakest preimage and constructs a Horn clause, in the same fashion as PROLOG-EBG. The only difference between LEMMA-ENUMERATOR and PROLOG-EBG is that LEMMA-ENUMERATOR ignores the training data and enumerates all proof trees.

Notice LEMMA-ENUMERATOR will output a superset of the Horn clauses output by PROLOG-EBG. Given this fact, several questions arise. First, if its hypotheses follow from the domain theory alone, then what is the role of training data in PROLOG-EBG? The answer is that training examples focus the PROLOG-EBG algorithm on generating rules that cover the distribution of instances that occur in practice. In our original chess example, for instance, the set of all possible lemmas is huge, whereas the set of chess positions that occur in normal play is only a small fraction of those that are syntactically possible. Therefore, by focusing only on training examples encountered in practice, the program is likely to develop a smaller, more relevant set of rules than if it attempted to enumerate all possible lemmas about chess.

The second question that arises is whether PROLOG-EBG can ever learn a hypothesis that goes beyond the knowledge that is already implicit in the domain theory. Put another way, will it ever learn to classify an instance that could not be classified by the original domain theory (assuming a theorem prover with unbounded computational resources)? Unfortunately, it will not. If $B \vdash h$, then any classification entailed by h will also be entailed by B . Is this an inherent limitation of analytical or deductive learning methods? No, it is not, as illustrated by the following example.

To produce an instance of deductive learning in which the learned hypothesis h entails conclusions that are not entailed by B , we must create an example where $B \not\vdash h$ but where $D \wedge B \vdash h$ (recall the constraint given by Equation (11.2)). One interesting case is when B contains assertions such as “If x satisfies the target concept, then so will $g(x)$.” Taken alone, this assertion does not entail the classification of any instances. However, once we observe a positive example, it allows generalizing deductively to other unseen instances. For example, consider learning the *PlayTennis* target concept, describing the days on which our friend Ross would like to play tennis. Imagine that each day is described only by the single attribute *Humidity*, and the domain theory B includes the single assertion “If Ross likes to play tennis when the humidity is x , then he will also like to play tennis when the humidity is lower than x ,” which can be stated more formally as

$$\begin{aligned} (\forall x) \quad & \text{IF } ((\text{PlayTennis} = \text{Yes}) \leftarrow (\text{Humidity} = x)) \\ & \text{THEN } ((\text{PlayTennis} = \text{Yes}) \leftarrow (\text{Humidity} \leq x)) \end{aligned}$$

Note that this domain theory does not entail any conclusions regarding which instances are positive or negative instances of *PlayTennis*. However, once the learner observes a positive example day for which *Humidity* = .30, the domain theory together with this positive example entails the following general hypothe-

sis h :

$$(PlayTennis = Yes) \leftarrow (Humidity \leq .30)$$

To summarize, this example illustrates a situation where $B \not\vdash h$, but where $B \wedge D \vdash h$. The learned hypothesis in this case entails predictions that are not entailed by the domain theory alone. The phrase *knowledge-level learning* is sometimes used to refer to this type of learning, in which the learned hypothesis entails predictions that go beyond those entailed by the domain theory. The set of all predictions entailed by a set of assertions Y is often called the *deductive closure* of Y . The key distinction here is that in knowledge-level learning the deductive closure of B is a proper subset of the deductive closure of $B + h$.

A second example of knowledge-level analytical learning is provided by considering a type of assertions known as *determinations*, which have been explored in detail by Russell (1989) and others. Determinations assert that some attribute of the instance is fully determined by certain other attributes, without specifying the exact nature of the dependence. For example, consider learning the target concept “people who speak Portuguese,” and imagine we are given as a domain theory the single determination assertion “the language spoken by a person is determined by their nationality.” Taken alone, this domain theory does not enable us to classify any instances as positive or negative. However, if we observe that “Joe, a 23-year-old left-handed Brazilian, speaks Portuguese,” then we can conclude from this positive example and the domain theory that “all Brazilians speak Portuguese.”

Both of these examples illustrate how deductive learning can produce output hypotheses that are not entailed by the domain theory alone. In both of these cases, the output hypothesis h satisfies $B \wedge D \vdash h$, but does not satisfy $B \vdash h$. In both cases, the learner *deduces* a justified hypothesis that does not follow from either the domain theory alone or the training data alone.

11.4 EXPLANATION-BASED LEARNING OF SEARCH CONTROL KNOWLEDGE

As noted above, the practical applicability of the PROLOG-EBG algorithm is restricted by its requirement that the domain theory be correct and complete. One important class of learning problems where this requirement is easily satisfied is learning to speed up complex search programs. In fact, the largest scale attempts to apply explanation-based learning have addressed the problem of learning to control search, or what is sometimes called “speedup” learning. For example, playing games such as chess involves searching through a vast space of possible moves and board positions to find the best move. Many practical scheduling and optimization problems are easily formulated as large search problems, in which the task is to find some move toward the goal state. In such problems the definitions of the legal search operators, together with the definition of the search objective, provide a complete and correct domain theory for learning search control knowledge.

Exactly how should we formulate the problem of learning search control so that we can apply explanation-based learning? Consider a general search problem where S is the set of possible search states, O is a set of legal search operators that transform one search state into another, and G is a predicate defined over S that indicates which states are goal states. The problem in general is to find a sequence of operators that will transform an arbitrary initial state s_i to some final state s_f that satisfies the goal predicate G . One way to formulate the learning problem is to have our system learn a separate target concept for each of the operators in O . In particular, for each operator o in O it might attempt to learn the target concept “the set of states for which o leads toward a goal state.” Of course the exact choice of which target concepts to learn depends on the internal structure of problem solver that must use this learned knowledge. For example, if the problem solver is a means-ends planning system that works by establishing and solving subgoals, then we might instead wish to learn target concepts such as “the set of planning states in which subgoals of type A should be solved before subgoals of type B .”

One system that employs explanation-based learning to improve its search is PRODIGY (Carbonell et al. 1990). PRODIGY is a domain-independent planning system that accepts the definition of a problem domain in terms of the state space S and operators O . It then solves problems of the form “find a sequence of operators that leads from initial state s_i to a state that satisfies goal predicate G .” PRODIGY uses a means-ends planner that decomposes problems into subgoals, solves them, then combines their solutions into a solution for the full problem. Thus, during its search for problem solutions PRODIGY repeatedly faces questions such as “Which subgoal should be solved next?” and “Which operator should be considered for solving this subgoal?” Minton (1988) describes the integration of explanation-based learning into PRODIGY by defining a set of target concepts appropriate for these kinds of control decisions that it repeatedly confronts. For example, one target concept is “the set of states in which subgoal A should be solved before subgoal B .” An example of a rule learned by PRODIGY for this target concept in a simple block-stacking problem domain is

IF	One subgoal to be solved is $On(x, y)$, and
	One subgoal to be solved is $On(y, z)$
THEN	Solve the subgoal $On(y, z)$ before $On(x, y)$

To understand this rule, consider again the simple block stacking problem illustrated in Figure 9.3. In the problem illustrated by that figure, the goal is to stack the blocks so that they spell the word “universal.” PRODIGY would decompose this problem into several subgoals to be achieved, including $On(U, N)$, $On(N, I)$, etc. Notice the above rule matches the subgoals $On(U, N)$ and $On(N, I)$, and recommends solving the subproblem $On(N, I)$ before solving $On(U, N)$. The justification for this rule (and the explanation used by PRODIGY to learn the rule) is that if we solve the subgoals in the reverse sequence, we will encounter a conflict in which we must undo the solution to the $On(U, N)$ subgoal in order to achieve the other subgoal $On(N, I)$. PRODIGY learns by first encountering such a conflict, then

explaining to itself the reason for this conflict and creating a rule such as the one above. The net effect is that PRODIGY uses domain-independent knowledge about possible subgoal conflicts, together with domain-specific knowledge of specific operators (e.g., the fact that the robot can pick up only one block at a time), to learn useful domain-specific planning rules such as the one illustrated above.

The use of explanation-based learning to acquire control knowledge for PRODIGY has been demonstrated in a variety of problem domains including the simple block-stacking problem above, as well as more complex scheduling and planning problems. Minton (1988) reports experiments in three problem domains, in which the learned control rules improve problem-solving efficiency by a factor of two to four. Furthermore, the performance of these learned rules is comparable to that of handwritten rules across these three problem domains. Minton also describes a number of extensions to the basic explanation-based learning procedure that improve its effectiveness for learning control knowledge. These include methods for simplifying learned rules and for removing learned rules whose benefits are smaller than their cost.

A second example of a general problem-solving architecture that incorporates a form of explanation-based learning is the SOAR system (Laird et al. 1986; Newell 1990). SOAR supports a broad variety of problem-solving strategies that subsumes PRODIGY's means-ends planning strategy. Like PRODIGY, however, SOAR learns by explaining situations in which its current search strategy leads to inefficiencies. When it encounters a search choice for which it does not have a definite answer (e.g., which operator to apply next) SOAR reflects on this search impasse, using weak methods such as generate-and-test to determine the correct course of action. The reasoning used to resolve this impasse can be interpreted as an explanation for how to resolve similar impasses in the future. SOAR uses a variant of explanation-based learning called *chunking* to extract the general conditions under which the same explanation applies. SOAR has been applied in a great number of problem domains and has also been proposed as a psychologically plausible model of human learning processes (see Newell 1990).

PRODIGY and SOAR demonstrate that explanation-based learning methods can be successfully applied to acquire search control knowledge in a variety of problem domains. Nevertheless, many or most heuristic search programs still use numerical evaluation functions similar to the one described in Chapter 1, rather than rules acquired by explanation-based learning. What is the reason for this? In fact, there are significant practical problems with applying EBL to learning search control. First, in many cases the number of control rules that must be learned is very large (e.g., many thousands of rules). As the system learns more and more control rules to improve its search, it must pay a larger and larger cost at each step to match this set of rules against the current search state. Note this problem is not specific to explanation-based learning; it will occur for any system that represents its learned knowledge by a growing set of rules. Efficient algorithms for matching rules can alleviate this problem, but not eliminate it completely. Minton (1988) discusses strategies for empirically estimating the computational cost and benefit of each rule, learning rules only when the estimated benefits outweigh the estimated costs

and deleting rules later found to have negative utility. He describes how using this kind of *utility analysis* to determine what should be learned and what should be forgotten significantly enhances the effectiveness of explanation-based learning in PRODIGY. For example, in a series of robot block-stacking problems, PRODIGY encountered 328 opportunities for learning a new rule, but chose to exploit only 69 of these, and eventually reduced the learned rules to a set of 19, once low-utility rules were eliminated. Tambe et al. (1990) and Doorenbos (1993) discuss how to identify types of rules that will be particularly costly to match, as well as methods for re-expressing such rules in more efficient forms and methods for optimizing rule-matching algorithms. Doorenbos (1993) describes how these methods enabled SOAR to efficiently match a set of 100,000 learned rules in one problem domain, without a significant increase in the cost of matching rules per state.

A second practical problem with applying explanation-based learning to learning search control is that in many cases it is intractable even to construct the explanations for the desired target concept. For example, in chess we might wish to learn a target concept such as “states for which operator A leads toward the optimal solution.” Unfortunately, to prove or explain why A leads toward the optimal solution requires explaining that every alternative operator leads to a less optimal outcome. This typically requires effort exponential in the search depth. Chien (1993) and Tadepalli (1990) explore methods for “lazy” or “incremental” explanation, in which heuristics are used to produce partial and approximate, but tractable, explanations. Rules are extracted from these imperfect explanations as though the explanations were perfect. Of course these learned rules may be incorrect due to the incomplete explanations. The system accommodates this by monitoring the performance of the rule on subsequent cases. If the rule subsequently makes an error, then the original explanation is incrementally elaborated to cover the new case, and a more refined rule is extracted from this incrementally improved explanation.

Many additional research efforts have explored the use of explanation-based learning for improving the efficiency of search-based problem solvers (for example, Mitchell 1981; Silver 1983; Shavlik 1990; Mahadevan et al. 1993; Gervasio and DeJong 1994; DeJong 1994). Bennett and DeJong (1996) explore explanation-based learning for robot planning problems where the system has an imperfect domain theory that describes its world and actions. Dietterich and Flann (1995) explore the integration of explanation-based learning with reinforcement learning methods discussed in Chapter 13. Mitchell and Thrun (1993) describe the application of an explanation-based neural network learning method (see the EBNN algorithm discussed in Chapter 12) to reinforcement learning problems.

11.5 SUMMARY AND FURTHER READING

The main points of this chapter include:

- In contrast to purely inductive learning methods that seek a hypothesis to fit the training data, purely analytical learning methods seek a hypothesis

that fits the learner's prior knowledge and covers the training examples. Humans often make use of prior knowledge to guide the formation of new hypotheses. This chapter examines purely analytical learning methods. The next chapter examines combined inductive-analytical learning.

- Explanation-based learning is a form of analytical learning in which the learner processes each novel training example by (1) explaining the observed target value for this example in terms of the domain theory, (2) analyzing this explanation to determine the general conditions under which the explanation holds, and (3) refining its hypothesis to incorporate these general conditions.
- PROLOG-EBG is an explanation-based learning algorithm that uses first-order Horn clauses to represent both its domain theory and its learned hypotheses. In PROLOG-EBG an explanation is a PROLOG proof, and the hypothesis extracted from the explanation is the weakest preimage of this proof. As a result, the hypotheses output by PROLOG-EBG follow deductively from its domain theory.
- Analytical learning methods such as PROLOG-EBG construct useful intermediate features as a side effect of analyzing individual training examples. This analytical approach to feature generation complements the statistically based generation of intermediate features (e.g., hidden unit features) in inductive methods such as BACKPROPAGATION.
- Although PROLOG-EBG does not produce hypotheses that extend the deductive closure of its domain theory, other deductive learning procedures can. For example, a domain theory containing determination assertions (e.g., "nationality determines language") can be used together with observed data to deductively infer hypotheses that go beyond the deductive closure of the domain theory.
- One important class of problems for which a correct and complete domain theory can be found is the class of large state-space search problems. Systems such as PRODIGY and SOAR have demonstrated the utility of explanation-based learning methods for automatically acquiring effective search control knowledge that speeds up problem solving in subsequent cases.
- Despite the apparent usefulness of explanation-based learning methods in humans, purely deductive implementations such as PROLOG-EBG suffer the disadvantage that the output hypothesis is only as correct as the domain theory. In the next chapter we examine approaches that combine inductive and analytical learning methods in order to learn effectively from imperfect domain theories and limited training data.

The roots of analytical learning methods can be traced to early work by Fikes et al. (1972) on learning macro-operators through analysis of operators in ABSTRIPS and to somewhat later work by Soloway (1977) on the use of explicit prior knowledge in learning. Explanation-based learning methods similar to those discussed in this chapter first appeared in a number of systems developed during the early 1980s, including DeJong (1981); Mitchell (1981); Winston et al.

(1983); and Silver (1983). DeJong and Mooney (1986) and Mitchell et al. (1986) provided general descriptions of the explanation-based learning paradigm, which helped spur a burst of research on this topic during the late 1980s. A collection of research on explanation-based learning performed at the University of Illinois is described by DeJong (1993), including algorithms that modify the structure of the explanation in order to correctly generalize iterative and temporal explanations. More recent research has focused on extending explanation-based methods to accommodate imperfect domain theories and to incorporate inductive together with analytical learning (see Chapter 12). An edited collection exploring the role of goals and prior knowledge in human and machine learning is provided by Ram and Leake (1995), and a recent overview of explanation-based learning is given by DeJong (1997).

The most serious attempts to employ explanation-based learning with perfect domain theories have been in the area of learning search control, or “speedup” learning. The SOAR system described by Laird et al. (1986) and the PRODIGY system described by Carbonell et al. (1990) are among the most developed systems that use explanation-based learning methods for learning in problem solving. Rosenbloom and Laird (1986) discuss the close relationship between SOAR’s learning method (called “chunking”) and other explanation-based learning methods. More recently, Dietterich and Flann (1995) have explored the combination of explanation-based learning with reinforcement learning methods for learning search control.

While our primary purpose here is to study machine learning algorithms, it is interesting to note that experimental studies of human learning provide support for the conjecture that human learning is based on explanations. For example, Ahn et al. (1987) and Qin et al. (1992) summarize evidence supporting the conjecture that humans employ explanation-based learning processes. Wisniewski and Medin (1995) describe experimental studies of human learning that suggest a rich interplay between prior knowledge and observed data to influence the learning process. Kotovsky and Baillargeon (1994) describe experiments that suggest even 11-month-old infants build on prior knowledge as they learn.

The analysis performed in explanation-based learning is similar to certain kinds of program optimization methods used for PROLOG programs, such as partial evaluation; van Harmelen and Bundy (1988) provide one discussion of the relationship.

EXERCISES

- 11.1.** Consider the problem of learning the target concept “pairs of people who live in the same house,” denoted by the predicate *HouseMates*(*x*, *y*). Below is a positive example of the concept.

HouseMates(*Joe*, *Sue*)

Person(*Joe*)

Sex(*Joe*, *Male*)

HairColor(*Joe*, *Black*)

Person(*Sue*)

Sex(*Sue*, *Female*)

HairColor(*Sue*, *Brown*)

<i>Height(Joe, Short)</i>	<i>Height(Sue, Short)</i>
<i>Nationality(Joe, US)</i>	<i>Nationality(Sue, US)</i>
<i>Mother(Joe, Mary)</i>	<i>Mother(Sue, Mary)</i>
<i>Age(Joe, 8)</i>	<i>Age(Sue, 6)</i>

The following domain theory is helpful for acquiring the *HouseMates* concept:

HouseMates(x, y) ← InSameFamily(x, y)
HouseMates(x, y) ← FraternityBrothers(x, y)
InSameFamily(x, y) ← Married(x, y)
InSameFamily(x, y) ← Youngster(x) ∧ Youngster(y) ∧ SameMother(x, y)
SameMother(x, y) ← Mother(x, z) ∧ Mother(y, z)
Youngster(x) ← Age(x, a) ∧ LessThan(a, 10)

Apply the PROLOG-EBG algorithm to the task of generalizing from the above instance, using the above domain theory. In particular,

- (a) Show a hand-trace of the PROLOG-EBG algorithm applied to this problem; that is, show the explanation generated for the training instance, show the result of regressing the target concept through this explanation, and show the resulting Horn clause rule.
 - (b) Suppose that the target concept is “people who live with Joe” instead of “pairs of people who live together.” Write down this target concept in terms of the above formalism. Assuming the same training instance and domain theory as before, what Horn clause rule will PROLOG-EBG produce for this new target concept?
- 11.2. As noted in Section 11.3.1, PROLOG-EBG can construct useful new features that are not explicit features of the instances, but that are defined in terms of the explicit features and that are useful for describing the appropriate generalization. These features are derived as a side effect of analyzing the training example explanation. A second method for deriving useful features is the BACKPROPAGATION algorithm for multilayer neural networks, in which new features are learned by the hidden units based on the statistical properties of a large number of examples. Can you suggest a way in which one might combine these analytical and inductive approaches to generating new features? (Warning: This is an open research problem.)

REFERENCES

- Ahn, W., Mooney, R. J., Brewer, W. F., & DeJong, G. F. (1987). Schema acquisition from one example: Psychological evidence for explanation-based learning. *Ninth Annual Conference of the Cognitive Science Society* (pp. 50–57). Hillsdale, NJ: Lawrence Erlbaum Associates.
- Bennett, S. W., & DeJong, G. F. (1996). Real-world robotics: Learning to plan for robust execution. *Machine Learning*, 23, 121.
- Carbonell, J., Knoblock, C., & Minton, S. (1990). PRODIGY: An integrated architecture for planning and learning. In K. VanLehn (Ed.), *Architectures for Intelligence*. Hillsdale, NJ: Lawrence Erlbaum Associates.
- Chien, S. (1993). NONMON: Learning with recoverable simplifications. In G. DeJong (Ed.), *Investigating explanation-based learning* (pp. 410–434). Boston, MA: Kluwer Academic Publishers.
- Davies, T. R., and Russell, S. J. (1987). A logical approach to reasoning by analogy. *Proceedings of the 10th International Joint Conference on Artificial Intelligence* (pp. 264–270). San Mateo, CA: Morgan Kaufmann.

- DeJong, G. (1981). Generalizations based on explanations. *Proceedings of the Seventh International Joint Conference on Artificial Intelligence* (pp. 67–70).
- DeJong, G., & Mooney, R. (1986). Explanation-based learning: An alternative view. *Machine Learning*, 1(2), 145–176.
- DeJong, G. (Ed.). (1993). *Investigating explanation-based learning*. Boston, MA: Kluwer Academic Publishers.
- DeJong, G. (1994). Learning to plan in continuous domains. *Artificial Intelligence*, 64(1), 71–141.
- DeJong, G. (1997). Explanation-based learning. In A. Tucker (Ed.), *The Computer Science and Engineering Handbook* (pp. 499–520). Boca Raton, FL: CRC Press.
- Dietterich, T. G., Flann, N. S. (1995). Explanation-based learning and reinforcement learning: A unified view. *Proceedings of the 12th International Conference on Machine Learning* (pp. 176–184). San Mateo, CA: Morgan Kaufmann.
- Doorenbos, R. E. (1993). Matching 100,000 learned rules. *Proceedings of the Eleventh National Conference on Artificial Intelligence* (pp. 290–296). AAAI Press/MIT Press.
- Fikes, R., Hart, P., & Nilsson, N. (1972). Learning and executing generalized robot plans. *Artificial Intelligence*, 3(4), 251–288.
- Fisher, D., Subramanian, D., & Tadepalli, P. (1992). An overview of current research on knowledge compilation and speedup learning. *Proceedings of the Second International Workshop on Knowledge Compilation and Speedup Learning*.
- Flann, N. S., & Dietterich, T. G. (1989). A study of explanation-based methods for inductive learning. *Machine Learning*, 4, 187–226.
- Gervasio, M. T., & DeJong, G. F. (1994). An incremental learning approach to completable planning. *Proceedings of the Eleventh International Conference on Machine Learning*, New Brunswick, NJ. San Mateo, CA: Morgan Kaufmann.
- van Harmelen, F., & Bundy, A. (1988). Explanation-based generalisation = partial evaluation. *Artificial Intelligence*, 36(3), 401–412.
- Kedar-Cabelli, S., & McCarty, T. (1987). Explanation-based generalization as resolution theorem proving. *Proceedings of the Fourth International Workshop on Machine Learning* (pp. 383–389). San Francisco: Morgan Kaufmann.
- Kotovsky, L., & Baillargeon, R. (1994). Calibration-based reasoning about collision events in 11-month-old infants. *Cognition*, 51, 107–129.
- Laird, J. E., Rosenbloom, P. S., & Newell, A. (1986). Chunking in SOAR: The anatomy of a general learning mechanism. *Machine Learning*, 1; 11.
- Mahadevan, S., Mitchell, T., Mostow, D. J., Steinberg, L., & Tadepalli, P. (1993). An apprentice-based approach to knowledge acquisition. In S. Mahadevan, T. Mitchell, D. J. Mostow, L. Steinberg, & P. Tadepalli (Eds.), *Artificial Intelligence*, 64(1), 1–52.
- Minton, S. (1988). *Learning search control knowledge: An explanation-based approach*. Boston, MA: Kluwer Academic Publishers.
- Minton, S., Carbonell, J., Knoblock, C., Kuokka, D., Etzioni, O., & Gil, Y. (1989). Explanation-based learning: A problem solving perspective. *Artificial Intelligence*, 40, 63–118.
- Minton, S. (1990). Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence*, 42, 363–391.
- Mitchell, T. M. (1981). *Toward combining empirical and analytical methods for inferring heuristics* (Technical Report LCSR-TR-27), Rutgers Computer Science Department. (Also reprinted in A. Elithorn & R. Banerji (Eds), *Artificial and Human Intelligence*. North-Holland, 1984.)
- Mitchell, T. M. (1983). Learning and problem-solving. *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*. San Francisco: Morgan Kaufmann.
- Mitchell, T. M., Keller, R., & Kedar-Cabelli, S. (1986). Explanation-based generalization: A unifying view. *Machine Learning*, 1(1), 47–80.
- Mitchell, T. M. (1990). Becoming increasingly reactive. *Proceedings of the Eighth National Conference on Artificial Intelligence*. Menlo Park, CA: AAAI Press.
- Mitchell, T. M., & Thrun, S. B. (1993). Explanation-based neural network learning for robot control. In S. Hanson et al. (Eds.), *Advances in neural information processing systems 5* (pp. 287–294). San Mateo, CA: Morgan-Kaufmann Press.

- Newell, A. (1990). *Unified theories of cognition*. Cambridge, MA: Harvard University Press.
- Qin, Y., Mitchell, T., & Simon, H. (1992). Using explanation-based generalization to simulate human learning from examples and learning by doing. *Proceedings of the Florida AI Research Symposium* (pp. 235–239).
- Ram, A., & Leake, D. B. (Eds.). (1995). *Goal-driven learning*. Cambridge, MA: MIT Press.
- Rosenbloom, P., & Laird, J. (1986). Mapping explanation-based generalization onto SOAR. *Fifth National Conference on Artificial Intelligence* (pp. 561–567). AAAI Press.
- Russell, S. (1989). *The use of knowledge in analogy and induction*. San Francisco: Morgan Kaufmann.
- Shavlik, J. W. (1990). Acquiring recursive and iterative concepts with explanation-based learning. *Machine Learning*, 5, 39.
- Silver, B. (1983). Learning equation solving methods from worked examples. *Proceedings of the 1983 International Workshop on Machine Learning* (pp. 99–104). CS Department, University of Illinois at Urbana-Champaign.
- Silver, B. (1986). Precondition analysis: Learning control information. In R. Michalski et al. (Eds.), *Machine Learning: An AI approach* (pp. 647–670). San Mateo, CA: Morgan Kaufmann.
- Soloway, E. (1977). *Knowledge directed learning using multiple levels of description* (Ph.D. thesis). University of Massachusetts, Amherst.
- Tadepalli, P. (1990). *Tractable learning and planning in games* (Technical report ML-TR-31) (Ph.D. dissertation). Rutgers University Computer Science Department.
- Tambe, M., Newell, A., & Rosenbloom, P. S. (1990). The problem of expensive chunks and its solution by restricting expressiveness. *Machine Learning*, 5(4), 299–348.
- Waldinger, R. (1977). Achieving several goals simultaneously. In E. Elcock & D. Michie (Eds.), *Machine Intelligence 8*. London: Ellis Horwood Ltd.
- Winston, P., Binford, T., Katz, B., & Lowry, M. (1983). Learning physical descriptions from functional definitions, examples, and precedents. *Proceedings of the National Conference on Artificial Intelligence* (pp. 433–439). San Mateo, CA: Morgan Kaufmann.
- Wisniewski, E. J., & Medin, D. L. (1995). Harpoons and long sticks: The interaction of theory and similarity in rule induction. In A. Ram & D. B. Leake (Eds.), *Goal-driven learning* (pp. 177–210). Cambridge, MA: MIT Press.

CHAPTER

12

COMBINING INDUCTIVE AND ANALYTICAL LEARNING

Purely inductive learning methods formulate general hypotheses by finding empirical regularities over the training examples. Purely analytical methods use prior knowledge to derive general hypotheses deductively. This chapter considers methods that combine inductive and analytical mechanisms to obtain the benefits of both approaches: better generalization accuracy when prior knowledge is available and reliance on observed training data to overcome shortcomings in prior knowledge. The resulting combined methods outperform both purely inductive and purely analytical learning methods. This chapter considers inductive-analytical learning methods based on both symbolic and artificial neural network representations.

12.1 MOTIVATION

In previous chapters we have seen two paradigms for machine learning: inductive learning and analytical learning. Inductive methods, such as decision tree induction and neural network BACKPROPAGATION, seek general hypotheses that fit the observed training data. Analytical methods, such as PROLOG-EBG, seek general hypotheses that fit prior knowledge while covering the observed data. These two learning paradigms are based on fundamentally different justifications for learned hypotheses and offer complementary advantages and disadvantages. Combining them offers the possibility of more powerful learning methods.

Purely analytical learning methods offer the advantage of generalizing more accurately from less data by using prior knowledge to guide learning. However, they can be misled when given incorrect or insufficient prior knowledge. Purely inductive methods offer the advantage that they require no explicit prior knowledge and learn regularities based solely on the training data. However, they can fail when given insufficient training data, and can be misled by the implicit inductive bias they must adopt in order to generalize beyond the observed data. Table 12.1 summarizes these complementary advantages and pitfalls of inductive and analytical learning methods. This chapter considers the question of how to combine the two into a single algorithm that captures the best aspects of both.

The difference between inductive and analytical learning methods can be seen in the nature of the *justifications* that can be given for their learned hypotheses. Hypotheses output by purely analytical learning methods such as PROLOG-EBG carry a *logical* justification; the output hypothesis follows deductively from the domain theory and training examples. Hypotheses output by purely inductive learning methods such as BACKPROPAGATION carry a *statistical* justification; the output hypothesis follows from statistical arguments that the training sample is sufficiently large that it is probably representative of the underlying distribution of examples. This statistical justification for induction is clearly articulated in the PAC-learning results discussed in Chapter 7.

Given that analytical methods provide logically justified hypotheses and inductive methods provide statistically justified hypotheses, it is easy to see why combining them would be useful: Logical justifications are only as compelling as the assumptions, or prior knowledge, on which they are built. They are suspect or powerless if prior knowledge is incorrect or unavailable. Statistical justifications are only as compelling as the data and statistical assumptions on which they rest. They are suspect or powerless when assumptions about the underlying distributions cannot be trusted or when data is scarce. In short, the two approaches work well for different types of problems. By combining them we can hope to devise a more general learning approach that covers a more broad range of learning tasks.

Figure 12.1 summarizes a spectrum of learning problems that varies by the availability of prior knowledge and training data. At one extreme, a large volume

	Inductive learning	Analytical learning
Goal:	Hypothesis fits data	Hypothesis fits domain theory
Justification:	Statistical inference	Deductive inference
Advantages:	Requires little prior knowledge	Learns from scarce data
Pitfalls:	Scarce data, incorrect bias	Imperfect domain theory

TABLE 12.1

Comparison of purely analytical and purely inductive learning.

**FIGURE 12.1**

A spectrum of learning tasks. At the left extreme, no prior knowledge is available, and purely inductive learning methods with high sample complexity are therefore necessary. At the rightmost extreme, a perfect domain theory is available, enabling the use of purely analytical methods such as PROLOG-EBG. Most practical problems lie somewhere between these two extremes.

of training data is available, but no prior knowledge. At the other extreme, strong prior knowledge is available, but little training data. Most practical learning problems lie somewhere between these two extremes of the spectrum. For example, in analyzing a database of medical records to learn “symptoms for which treatment x is more effective than treatment y ,” one often begins with approximate prior knowledge (e.g., a qualitative model of the cause-effect mechanisms underlying the disease) that suggests the patient’s temperature is more likely to be relevant than the patient’s middle initial. Similarly, in analyzing a stock market database to learn the target concept “companies whose stock value will double over the next 10 months,” one might have approximate knowledge of economic causes and effects, suggesting that the gross revenue of the company is more likely to be relevant than the color of the company logo. In both of these settings, our own prior knowledge is incomplete, but is clearly useful in helping discriminate relevant features from irrelevant.

The question considered in this chapter is “What kinds of learning algorithms can we devise that make use of approximate prior knowledge, together with available data, to form general hypotheses?” Notice that even when using a purely inductive learning algorithm, one has the opportunity to make design choices based on prior knowledge of the particular learning task. For example, when applying BACKPROPAGATION to a problem such as speech recognition, one must choose the encoding of input and output data, the error function to be minimized during gradient descent, the number of hidden units, the topology of the network, the learning rate and momentum, etc. In making these choices, human designers have the opportunity to embed task-specific knowledge into the learning algorithm. The result, however, is a purely inductive instantiation of BACKPROPAGATION, *specialized* by the designer’s choices to the task of speech recognition. Our interest here lies in something different. We are interested in systems that take prior knowledge as an *explicit input* to the learner, in the same sense that the training data is an explicit input, so that they remain general purpose algorithms, even while taking advantage of domain-specific knowledge. In brief, our interest here lies in *domain-independent algorithms that employ explicitly input domain-dependent knowledge*.

What criteria should we use to compare alternative approaches to combining inductive and analytical learning? Given that the learner will generally not know the quality of the domain theory or the training data in advance, we are interested

in general methods that can operate robustly over the entire spectrum of problems of Figure 12.1. Some specific properties we would like from such a learning method include:

- Given no domain theory, it should learn at least as effectively as purely inductive methods.
- Given a perfect domain theory, it should learn at least as effectively as purely analytical methods.
- Given an imperfect domain theory and imperfect training data, it should combine the two to outperform either purely inductive or purely analytical methods.
- It should accommodate an unknown level of error in the training data.
- It should accommodate an unknown level of error in the domain theory.

Notice this list of desirable properties is quite ambitious. For example, accommodating errors in the training data is problematic even for statistically based induction without at least some prior knowledge or assumption regarding the distribution of errors. Combining inductive and analytical learning is an area of active current research. While the above list is a fair summary of what we would like our algorithms to accomplish, we do not yet have algorithms that satisfy all these constraints in a fully general fashion.

The next section provides a more detailed discussion of the combined inductive-analytical learning problem. Subsequent sections describe three different approaches to combining approximate prior knowledge with available training data to guide the learner's search for an appropriate hypothesis. Each of these three approaches has been demonstrated to outperform purely inductive methods in multiple task domains. For ease of comparison, we use a single example problem to illustrate all three approaches.

12.2 INDUCTIVE-ANALYTICAL APPROACHES TO LEARNING

12.2.1 The Learning Problem

To summarize, the learning problem considered in this chapter is

Given:

- A set of training examples D , possibly containing errors
- A domain theory B , possibly containing errors
- A space of candidate hypotheses H

Determine:

- A hypothesis that best fits the training examples and domain theory

What precisely shall we mean by “the hypothesis that best fits the training examples and domain theory?” In particular, shall we prefer hypotheses that fit

the data a little better at the expense of fitting the theory less well, or vice versa? We can be more precise by defining measures of hypothesis error with respect to the data and with respect to the domain theory, then phrasing the question in terms of these errors. Recall from Chapter 5 that $error_D(h)$ is defined to be the proportion of examples from D that are misclassified by h . Let us define the error $error_B(h)$ of h with respect to a domain theory B to be the probability that h will disagree with B on the classification of a randomly drawn instance. We can attempt to characterize the desired output hypothesis in terms of these errors. For example, we could require the hypothesis that minimizes some combined measure of these errors, such as

$$\operatorname{argmin}_{h \in H} k_D error_D(h) + k_B error_B(h)$$

While this appears reasonable at first glance, it is not clear what values to assign to k_D and k_B to specify the relative importance of fitting the data versus fitting the theory. If we have a very poor theory and a great deal of reliable data, it will be best to weight $error_D(h)$ more heavily. Given a strong theory and a small sample of very noisy data, the best results would be obtained by weighting $error_B(h)$ more heavily. Of course if the learner does not know in advance the quality of the domain theory or training data, it will be unclear how it should weight these two error components.

An alternative perspective on the question of how to weight prior knowledge and data is the Bayesian perspective. Recall from Chapter 6 that Bayes theorem describes how to compute the posterior probability $P(h|D)$ of hypothesis h given observed training data D . In particular, Bayes theorem computes this posterior probability based on the observed data D , together with prior knowledge in the form of $P(h)$, $P(D)$, and $P(D|h)$. Thus we can think of $P(h)$, $P(D)$, and $P(D|h)$ as a form of background knowledge or domain theory, and we can think of Bayes theorem as a method for weighting this domain theory, together with the observed data D , to assign a posterior probability $P(h|D)$ to h . The Bayesian view is that one should simply choose the hypothesis whose posterior probability is greatest, and that Bayes theorem provides the proper method for weighting the contribution of this prior knowledge and observed data. Unfortunately, Bayes theorem implicitly assumes *perfect* knowledge about the probability distributions $P(h)$, $P(D)$, and $P(D|h)$. When these quantities are only imperfectly known, Bayes theorem alone does not prescribe how to combine them with the observed data. (One possible approach in such cases is to assume prior probability distributions over $P(h)$, $P(D)$, and $P(D|h)$ themselves, then calculate the expected value of the posterior $P(h|D)$. However, this requires additional knowledge about the priors over $P(h)$, $P(D)$, and $P(D|h)$, so it does not really solve the general problem.)

We will revisit the question of what we mean by “best” fit to the hypothesis and data as we examine specific algorithms. For now, we will simply say that the learning problem is to minimize some combined measure of the error of the hypothesis over the data and the domain theory.

12.2.2 Hypothesis Space Search

How can the domain theory and training data best be combined to constrain the search for an acceptable hypothesis? This remains an open question in machine learning. This chapter surveys a variety of approaches that have been proposed, many of which consist of extensions to inductive methods we have already studied (e.g., BACKPROPAGATION, FOIL).

One way to understand the range of possible approaches is to return to our view of learning as a task of searching through the space of alternative hypotheses. We can characterize most learning methods as search algorithms by describing the hypothesis space H they search, the initial hypothesis h_0 at which they begin their search, the set of search operators O that define individual search steps, and the goal criterion G that specifies the search objective. In this chapter we explore three different methods for using prior knowledge to alter the search performed by purely inductive methods.

- *Use prior knowledge to derive an initial hypothesis from which to begin the search.* In this approach the domain theory B is used to construct an initial hypothesis h_0 that is consistent with B . A standard inductive method is then applied, starting with the initial hypothesis h_0 . For example, the KBANN system described below learns artificial neural networks in this way. It uses prior knowledge to design the interconnections and weights for an initial network, so that this initial network is perfectly consistent with the given domain theory. This initial network hypothesis is then refined inductively using the BACKPROPAGATION algorithm and available data. Beginning the search at a hypothesis consistent with the domain theory makes it more likely that the final output hypothesis will better fit this theory.
- *Use prior knowledge to alter the objective of the hypothesis space search.* In this approach, the goal criterion G is modified to require that the output hypothesis fits the domain theory as well as the training examples. For example, the EBNN system described below learns neural networks in this way. Whereas inductive learning of neural networks performs gradient descent search to minimize the squared error of the network over the training data, EBNN performs gradient descent to optimize a different criterion. This modified criterion includes an additional term that measures the error of the learned network relative to the domain theory.
- *Use prior knowledge to alter the available search steps.* In this approach, the set of search operators O is altered by the domain theory. For example, the FOCL system described below learns sets of Horn clauses in this way. It is based on the inductive system FOIL, which conducts a greedy search through the space of possible Horn clauses, at each step revising its current hypothesis by adding a single new literal. FOCL uses the domain theory to expand the set of alternatives available when revising the hypothesis, allowing the

addition of multiple literals in a single search step when warranted by the domain theory. In this way, FOCL allows single-step moves through the hypothesis space that would correspond to many steps using the original inductive algorithm. These “macro-moves” can dramatically alter the course of the search, so that the final hypothesis found consistent with the data is different from the one that would be found using only the inductive search steps.

The following sections describe each of these approaches in turn.

12.3 USING PRIOR KNOWLEDGE TO INITIALIZE THE HYPOTHESIS

One approach to using prior knowledge is to initialize the hypothesis to perfectly fit the domain theory, then inductively refine this initial hypothesis as needed to fit the training data. This approach is used by the KBANN (Knowledge-Based Artificial Neural Network) algorithm to learn artificial neural networks. In KBANN an initial network is first constructed so that for every possible instance, the classification assigned by the network is identical to that assigned by the domain theory. The BACKPROPAGATION algorithm is then employed to adjust the weights of this initial network as needed to fit the training examples.

It is easy to see the motivation for this technique: if the domain theory is correct, the initial hypothesis will correctly classify all the training examples and there will be no need to revise it. However, if the initial hypothesis is found to imperfectly classify the training examples, then it will be refined inductively to improve its fit to the training examples. Recall that in the purely inductive BACKPROPAGATION algorithm, weights are typically initialized to small random values. The intuition behind KBANN is that even if the domain theory is only approximately correct, initializing the network to fit this domain theory will give a better starting approximation to the target function than initializing the network to random initial weights. This should lead, in turn, to better generalization accuracy for the final hypothesis.

This *initialize-the-hypothesis* approach to using the domain theory has been explored by several researchers, including Shavlik and Towell (1989), Towell and Shavlik (1994), Fu (1989, 1993), and Pratt (1993a, 1993b). We will use the KBANN algorithm described in Shavlik and Towell (1989) to illustrate this approach.

12.3.1 The KBANN Algorithm

The KBANN algorithm exemplifies the initialize-the-hypothesis approach to using domain theories. It assumes a domain theory represented by a set of propositional, nonrecursive Horn clauses. A Horn clause is propositional if it contains no variables. The input and output of KBANN are as follows:

KBANN(*Domain_Theory*, *Training_Examples*)*Domain_Theory*: Set of propositional, nonrecursive Horn clauses.*Training_Examples*: Set of $\langle \text{input output} \rangle$ pairs of the target function.*Analytical step*: Create an initial network equivalent to the domain theory.

1. For each instance attribute create a network input.
 2. For each Horn clause in the *Domain_Theory*, create a network unit as follows:
 - Connect the inputs of this unit to the attributes tested by the clause antecedents.
 - For each non-negated antecedent of the clause, assign a weight of W to the corresponding sigmoid unit input.
 - For each negated antecedent of the clause, assign a weight of $-W$ to the corresponding sigmoid unit input.
 - Set the threshold weight w_0 for this unit to $-(n - .5)W$, where n is the number of non-negated antecedents of the clause.
 3. Add additional connections among the network units, connecting each network unit at depth i from the input layer to all network units at depth $i + 1$. Assign random near-zero weights to these additional connections.
- Inductive step*: Refine the initial network.
4. Apply the BACKPROPAGATION algorithm to adjust the initial network weights to fit the *Training_Examples*.

TABLE 12.2

The KBANN algorithm. The domain theory is translated into an equivalent neural network (steps 1–3), which is inductively refined using the BACKPROPAGATION algorithm (step 4). A typical value for the constant W is 4.0.

Given:

- A set of training examples
- A domain theory consisting of nonrecursive, propositional Horn clauses

Determine:

- An artificial neural network that fits the training examples, biased by the domain theory

The two stages of the KBANN algorithm are first to create an artificial neural network that perfectly fits the domain theory and second to use the BACKPROPAGATION algorithm to refine this initial network to fit the training examples. The details of this algorithm, including the algorithm for creating the initial network, are given in Table 12.2 and illustrated in Section 12.3.2.

12.3.2 An Illustrative Example

To illustrate the operation of KBANN, consider the simple learning problem summarized in Table 12.3, adapted from Towell and Shavlik (1989). Here each instance describes a physical object in terms of the material from which it is made, whether it is light, etc. The task is to learn the target concept *Cup* defined over such physical objects. Table 12.3 describes a set of training examples and a domain theory for the *Cup* target concept. Notice the domain theory defines a *Cup*

Domain theory:

$$\begin{aligned}
 Cup &\leftarrow Stable, Liftable, OpenVessel \\
 Stable &\leftarrow BottomIsFlat \\
 Liftable &\leftarrow Graspable, Light \\
 Graspable &\leftarrow HasHandle \\
 OpenVessel &\leftarrow HasConcavity, ConcavityPointsUp
 \end{aligned}$$

Training examples:

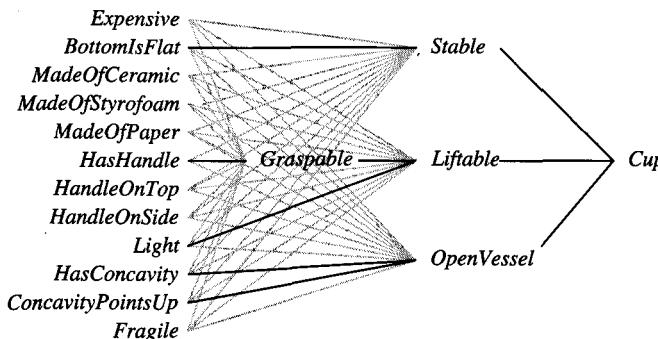
	Cups				Non-Cups				
	✓	✓	✓	✓	✓	✓	✓	✓	✓
BottomIsFlat	✓	✓	✓	✓	✓	✓	✓	✓	✓
ConcavityPointsUp	✓	✓	✓	✓	✓	✓	✓	✓	
Expensive	✓		✓			✓		✓	
Fragile	✓	✓			✓	✓	✓		✓
HandleOnTop					✓		✓		
HandleOnSide	✓			✓					✓
HasConcavity	✓	✓	✓	✓	✓	✓	✓	✓	✓
HasHandle	✓			✓	✓	✓			✓
Light	✓	✓	✓	✓	✓	✓	✓		✓
MadeOfCeramic	✓				✓	✓	✓		
MadeOfPaper				✓					✓
MadeOfStyrofoam	✓	✓			✓				✓

TABLE 12.3

The *Cup* learning task. An approximate domain theory and a set of training examples for the target concept *Cup*.

as an object that is *Stable*, *Litable*, and an *OpenVessel*. The domain theory also defines each of these three attributes in terms of more primitive attributes, terminating in the primitive, operational attributes that describe the instances. Note the domain theory is not perfectly consistent with the training examples. For example, the domain theory fails to classify the second and third training examples as positive examples. Nevertheless, the domain theory forms a useful approximation to the target concept. KBANN uses the domain theory and training examples together to learn the target concept more accurately than it could from either alone.

In the first stage of the KBANN algorithm (steps 1–3 in the algorithm), an initial network is constructed that is consistent with the domain theory. For example, the network constructed from the *Cup* domain theory is shown in Figure 12.2. In general the network is constructed by creating a sigmoid threshold unit for each Horn clause in the domain theory. KBANN follows the convention that a sigmoid output value greater than 0.5 is interpreted as *True* and a value below 0.5 as *False*. Each unit is therefore constructed so that its output will be greater than 0.5 just in those cases where the corresponding Horn clause applies. For each antecedent to the Horn clause, an input is created to the corresponding sigmoid unit. The weights of the sigmoid unit are then set so that it computes the logical AND of its inputs. In particular, for each input corresponding to a non-negated antecedent,

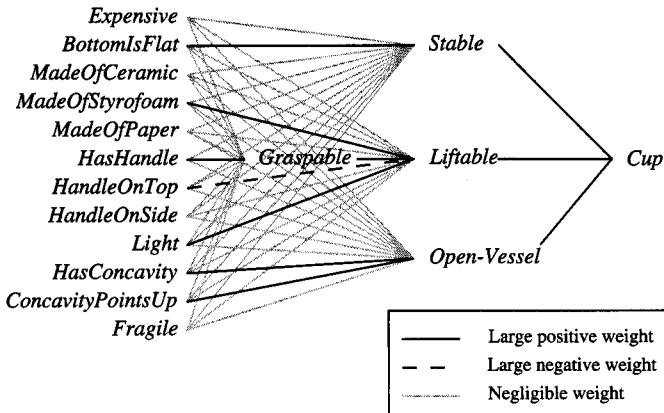
**FIGURE 12.2**

A neural network equivalent to the domain theory. This network, created in the first stage of the KBANN algorithm, produces output classifications identical to those of the given domain theory clauses. Dark lines indicate connections with weight W and correspond to antecedents of clauses from the domain theory. Light lines indicate connections with weights of approximately zero.

the weight is set to some positive constant W . For each input corresponding to a negated antecedent, the weight is set to $-W$. The threshold weight of the unit, w_0 is then set to $-(n-.5)W$, where n is the number of non-negated antecedents. When unit input values are 1 or 0, this assures that their weighted sum plus w_0 will be positive (and the sigmoid output will therefore be greater than 0.5) if and only if all clause antecedents are satisfied. Note for sigmoid units at the second and subsequent layers, unit inputs will not necessarily be 1 and 0 and the above argument may not apply. However, if a sufficiently large value is chosen for W , this KBANN algorithm can correctly encode the domain theory for arbitrarily deep networks. Towell and Shavlik (1994) report using $W = 4.0$ in many of their experiments.

Each sigmoid unit input is connected to the appropriate network input or to the output of another sigmoid unit, to mirror the graph of dependencies among the corresponding attributes in the domain theory. As a final step many additional inputs are added to each threshold unit, with their weights set approximately to zero. The role of these additional connections is to enable the network to inductively learn additional dependencies beyond those suggested by the given domain theory. The solid lines in the network of Figure 12.2 indicate unit inputs with weights of W , whereas the lightly shaded lines indicate connections with initial weights near zero. It is easy to verify that for sufficiently large values of W this network will output values identical to the predictions of the domain theory.

The second stage of KBANN (step 4 in the algorithm of Table 12.2) uses the training examples and the BACKPROPAGATION algorithm to refine the initial network weights. Of course if the domain theory and training examples contain no errors, the initial network will already fit the training data. In the *Cup* example, however, the domain theory and training data are inconsistent, and this step therefore alters the initial network weights. The resulting trained network is summarized in Figure 12.3, with dark solid lines indicating the largest positive weights, dashed lines indicating the largest negative weights, and light lines

**FIGURE 12.3**

Result of inductively refining the initial network. KBANN uses the training examples to modify the network weights derived from the domain theory. Notice the new dependency of *Liftable* on *MadeOfStyrofoam* and *HandleOnTop*.

indicating negligible weights. Although the initial network misclassifies several training examples from Table 12.3, the refined network of Figure 12.3 perfectly classifies all of these training examples.

It is interesting to compare the final, inductively refined network weights to the initial weights derived from the domain theory. As can be seen in Figure 12.3, significant new dependencies were discovered during the inductive step, including the dependency of the *Liftable* unit on the feature *MadeOfStyrofoam*. It is important to keep in mind that while the unit labeled *Liftable* was initially defined by the given Horn clause for *Liftable*, the subsequent weight changes performed by BACKPROPAGATION may have dramatically changed the meaning of this hidden unit. After training of the network, this unit may take on a very different meaning unrelated to the initial notion of *Liftable*.

12.3.3 Remarks

To summarize, KBANN analytically creates a network equivalent to the given domain theory, then inductively refines this initial hypothesis to better fit the training data. In doing so, it modifies the network weights as needed to overcome inconsistencies between the domain theory and observed data.

The chief benefit of KBANN over purely inductive BACKPROPAGATION (beginning with random initial weights) is that it typically generalizes more accurately than BACKPROPAGATION when given an approximately correct domain theory, especially when training data is scarce. KBANN and other initialize-the-hypothesis approaches have been demonstrated to outperform purely inductive systems in several practical problems. For example, Towell et al. (1990) describe the application of KBANN to a molecular genetics problem. Here the task was to learn to

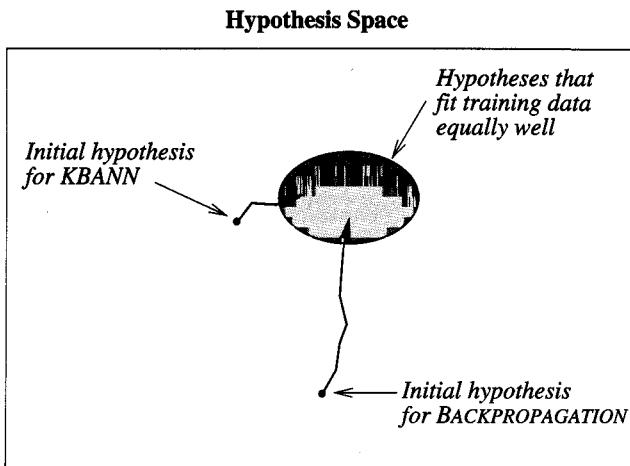
recognize DNA segments called promoter regions, which influence gene activity. In this experiment KBANN was given an initial domain theory obtained from a molecular geneticist, and a set of 53 positive and 53 negative training examples of promoter regions. Performance was evaluated using a leave-one-out strategy in which the system was run 106 different times. On each iteration KBANN was trained using 105 of the 106 examples and tested on the remaining example. The results of these 106 experiments were accumulated to provide an estimate of the true error rate. KBANN obtained an error rate of 4/106, compared to an error rate of 8/106 using standard BACKPROPAGATION. A variant of the KBANN approach was applied by Fu (1993), who reports an error rate of 2/106 on the same data. Thus, the impact of prior knowledge in these experiments was to reduce significantly the error rate. The training data for this experiment is available at World Wide Web site <http://www.ics.uci.edu/~mlearn/MLRepository.html>.

Both Fu (1993) and Towell et al. (1990) report that Horn clauses extracted from the final trained network provided a refined domain theory that better fit the observed data. Although it is sometimes possible to map from the learned network weights back to a refined set of Horn clauses, in the general case this is problematic because some weight settings have no direct Horn clause analog. Craven and Shavlik (1994) and Craven (1996) describe alternative methods for extracting symbolic rules from learned networks.

To understand the significance of KBANN it is useful to consider how its hypothesis search differs from that of the purely inductive BACKPROPAGATION algorithm. The hypothesis space search conducted by both algorithms is depicted schematically in Figure 12.4. As shown there, the key difference is the initial hypothesis from which weight tuning is performed. In the case that multiple hypotheses (weight vectors) can be found that fit the data—a condition that will be especially likely when training data is scarce—KBANN is likely to converge to a hypothesis that generalizes beyond the data in a way that is more similar to the domain theory predictions. On the other hand, the particular hypothesis to which BACKPROPAGATION converges will more likely be a hypothesis with small weights, corresponding roughly to a generalization bias of smoothly interpolating between training examples. In brief, KBANN uses a domain-specific theory to bias generalization, whereas BACKPROPAGATION uses a domain-independent syntactic bias toward small weight values. Note in this summary we have ignored the effect of local minima on the search.

Limitations of KBANN include the fact that it can accommodate only propositional domain theories; that is, collections of variable-free Horn clauses. It is also possible for KBANN to be misled when given highly inaccurate domain theories, so that its generalization accuracy can deteriorate below the level of BACKPROPAGATION. Nevertheless, it and related algorithms have been shown to be useful for several practical problems.

KBANN illustrates the initialize-the-hypothesis approach to combining analytical and inductive learning. Other examples of this approach include Fu (1993); Gallant (1988); Bradshaw et al. (1989); Yang and Bhargava (1990); Lacher et al. (1991). These approaches vary in the exact technique for constructing the initial

**FIGURE 12.4**

Hypothesis space search in KBANN. KBANN initializes the network to fit the domain theory, whereas BACKPROPAGATION initializes the network to small random weights. Both then refine the weights iteratively using the same gradient descent rule. When multiple hypotheses can be found that fit the training data (shaded region), KBANN and BACKPROPAGATION are likely to find different hypotheses due to their different starting points.

network, the application of BACKPROPAGATION to weight tuning, and in methods for extracting symbolic descriptions from the refined network. Pratt (1993a, 1993b) describes an initialize-the-hypothesis approach in which the prior knowledge is provided by a previously learned neural network for a related task, rather than a manually provided symbolic domain theory. Methods for training the values of Bayesian belief networks, as discussed in Section 6.11, can also be viewed as using prior knowledge to initialize the hypothesis. Here the prior knowledge corresponds to a set of conditional independence assumptions that determine the graph structure of the Bayes net, whose conditional probability tables are then induced from the training data.

12.4 USING PRIOR KNOWLEDGE TO ALTER THE SEARCH OBJECTIVE

The above approach begins the gradient descent search with a hypothesis that perfectly fits the domain theory, then perturbs this hypothesis as needed to maximize the fit to the training data. An alternative way of using prior knowledge is to incorporate it into the error criterion minimized by gradient descent, so that the network must fit a combined function of the training data and domain theory. In this section, we consider using prior knowledge in this fashion. In particular, we consider prior knowledge in the form of known derivatives of the target function. Certain types of prior knowledge can be expressed quite naturally in this form. For example, in training a neural network to recognize handwritten characters we

can specify certain derivatives of the target function in order to express our prior knowledge that “the identity of the character is independent of small translations and rotations of the image.”

Below we describe the TANGENTPROP algorithm, which trains a neural network to fit both training values and training derivatives. Section 12.4.4 then describes how these training derivatives can be obtained from a domain theory similar to the one used in the *Cup* example of Section 12.3. In particular, it discusses how the EBNN algorithm constructs explanations of individual training examples in order to extract training derivatives for use by TANGENTPROP. TANGENTPROP and EBNN have been demonstrated to outperform purely inductive methods in a variety of domains, including character and object recognition, and robot perception and control tasks.

12.4.1 The TANGENTPROP Algorithm

TANGENTPROP (Simard et al. 1992) accommodates domain knowledge expressed as derivatives of the target function with respect to transformations of its inputs. Consider a learning task involving an instance space X and target function f . Up to now we have assumed that each training example consists of a pair $\langle x_i, f(x_i) \rangle$ that describes some instance x_i and its training value $f(x_i)$. The TANGENTPROP algorithm assumes various training derivatives of the target function are also provided. For example, if each instance x_i is described by a single real value, then each training example may be of the form $\langle x_i, f(x_i), \frac{\partial f(x)}{\partial x}|_{x_i} \rangle$. Here $\frac{\partial f(x)}{\partial x}|_{x_i}$ denotes the derivative of the target function f with respect to x , evaluated at the point $x = x_i$.

To develop an intuition for the benefits of providing training derivatives as well as training values during learning, consider the simple learning task depicted in Figure 12.5. The task is to learn the target function f shown in the leftmost plot of the figure, based on the three training examples shown: $\langle x_1, f(x_1) \rangle$, $\langle x_2, f(x_2) \rangle$, and $\langle x_3, f(x_3) \rangle$. Given these three training examples, the BACKPROPAGATION algorithm can be expected to hypothesize a smooth function, such as the function g depicted in the middle plot of the figure. The rightmost plot shows the effect of

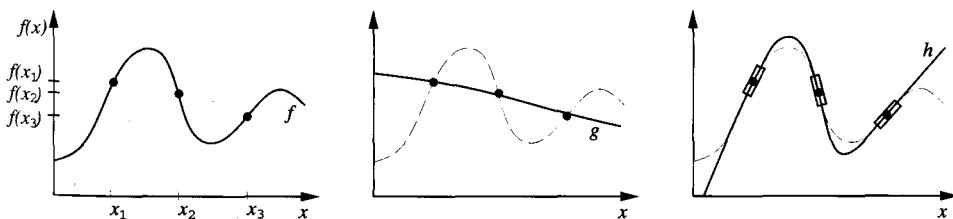


FIGURE 12.5

Fitting values and derivatives with TANGENTPROP. Let f be the target function for which three examples $\langle x_1, f(x_1) \rangle$, $\langle x_2, f(x_2) \rangle$, and $\langle x_3, f(x_3) \rangle$ are known. Based on these points the learner might generate the hypothesis g . If the derivatives are also known, the learner can generalize more accurately h .

providing training derivatives, or slopes, as additional information for each training example (e.g., $\langle x_1, f(x_1), \frac{\partial f(x)}{\partial x}|_{x_1} \rangle$). By fitting both the training values $f(x_i)$ and these training derivatives $\frac{\partial f(x)}{\partial x}|_{x_i}$, the learner has a better chance to correctly generalize from the sparse training data. To summarize, the impact of including the training derivatives is to override the usual syntactic inductive bias of BACKPROPAGATION that favors a smooth interpolation between points, replacing it by explicit input information about required derivatives. The resulting hypothesis h shown in the rightmost plot of the figure provides a much more accurate estimate of the true target function f .

In the above example, we considered only simple kinds of derivatives of the target function. In fact, TANGENTPROP can accept training derivatives with respect to various transformations of the input x . Consider, for example, the task of learning to recognize handwritten characters. In particular, assume the input x corresponds to an image containing a single handwritten character, and the task is to correctly classify the character. In this task, we might be interested in informing the learner that “the target function is invariant to small rotations of the character within the image.” In order to express this prior knowledge to the learner, we first define a transformation $s(\alpha, x)$, which rotates the image x by α degrees. Now we can express our assertion about rotational invariance by stating that for each training instance x_i , the derivative of the target function with respect to this transformation is zero (i.e., that rotating the input image does not alter the value of the target function). In other words, we can assert the following training derivative for every training instance x_i

$$\frac{\partial f(s(\alpha, x_i))}{\partial \alpha} = 0$$

where f is the target function and $s(\alpha, x_i)$ is the image resulting from applying the transformation s to the image x_i .

How are such training derivatives used by TANGENTPROP to constrain the weights of the neural network? In TANGENTPROP these training derivatives are incorporated into the error function that is minimized by gradient descent. Recall from Chapter 4 that the BACKPROPAGATION algorithm performs gradient descent to attempt to minimize the sum of squared errors

$$E = \sum_i (f(x_i) - \hat{f}(x_i))^2$$

where x_i denotes the i th training instance, f denotes the true target function, and \hat{f} denotes the function represented by the learned neural network.

In TANGENTPROP an additional term is added to the error function to penalize discrepancies between the training derivatives and the actual derivatives of the learned neural network function \hat{f} . In general, TANGENTPROP accepts multiple transformations (e.g., we might wish to assert both rotational invariance and translational invariance of the character identity). Each transformation must be of the form $s_j(\alpha, x)$ where α is a continuous parameter, where s_j is differentiable, and where $s_j(0, x) = x$ (e.g., for rotation of zero degrees the transformation is the identity function). For each such transformation, $s_j(\alpha, x)$, TANGENT-

PROP considers the squared error between the specified training derivative and the actual derivative of the learned neural network. The modified error function is

$$E = \sum_i \left[(f(x_i) - \hat{f}(x_i))^2 + \mu \sum_j \left(\frac{\partial f(s_j(\alpha, x_i))}{\partial \alpha} - \frac{\partial \hat{f}(s_j(\alpha, x_i))}{\partial \alpha} \right)_{\alpha=0}^2 \right] \quad (12.1)$$

where μ is a constant provided by the user to determine the relative importance of fitting training values versus fitting training derivatives. Notice the first term in this definition of E is the original squared error of the network versus training values, and the second term is the squared error in the network versus training derivatives.

Simard et al. (1992) give the gradient descent rule for minimizing this extended error function E . It can be derived in a fashion analogous to the derivation given in Chapter 4 for the simpler BACKPROPAGATION rule.

12.4.2 An Illustrative Example

Simard et al. (1992) present results comparing the generalization accuracy of TANGENTPROP and purely inductive BACKPROPAGATION for the problem of recognizing handwritten characters. More specifically, the task in this case is to label images containing a single digit between 0 and 9. In one experiment, both TANGENTPROP and BACKPROPAGATION were trained using training sets of varying size, then evaluated based on their performance over a separate test set of 160 examples. The prior knowledge given to TANGENTPROP was the fact that the classification of the digit is invariant of vertical and horizontal translation of the image (i.e., that the derivative of the target function was 0 with respect to these transformations). The results, shown in Table 12.4, demonstrate the ability of TANGENTPROP using this prior knowledge to generalize more accurately than purely inductive BACKPROPAGATION.

Training set size	Percent error on test set	
	TANGENTPROP	BACKPROPAGATION
10	34	48
20	17	33
40	7	18
80	4	10
160	0	3
320	0	0

TABLE 12.4

Generalization accuracy for TANGENTPROP and BACKPROPAGATION, for handwritten digit recognition. TANGENTPROP generalizes more accurately due to its prior knowledge that the identity of the digit is invariant of translation. These results are from Simard et al. (1992).

12.4.3 Remarks

To summarize, TANGENTPROP uses prior knowledge in the form of desired derivatives of the target function with respect to transformations of its inputs. It combines this prior knowledge with observed training data, by minimizing an objective function that measures both the network's error with respect to the training example values (fitting the data) and its error with respect to the desired derivatives (fitting the prior knowledge). The value of μ determines the degree to which the network will fit one or the other of these two components in the total error. The behavior of the algorithm is sensitive to μ , which must be chosen by the designer.

Although TANGENTPROP succeeds in combining prior knowledge with training data to guide learning of neural networks, it is not robust to errors in the prior knowledge. Consider what will happen when prior knowledge is incorrect, that is, when the training derivatives input to the learner do not correctly reflect the derivatives of the true target function. In this case the algorithm will attempt to fit incorrect derivatives. It may therefore generalize less accurately than if it ignored this prior knowledge altogether and used the purely inductive BACKPROPAGATION algorithm. If we knew in advance the degree of error in the training derivatives, we might use this information to select the constant μ that determines the relative importance of fitting training values and fitting training derivatives. However, this information is unlikely to be known in advance. In the next section we discuss the EBNN algorithm, which automatically selects values for μ on an example-by-example basis in order to address the possibility of incorrect prior knowledge.

It is interesting to compare the search through hypothesis space (weight space) performed by TANGENTPROP, KBANN, and BACKPROPAGATION. TANGENTPROP incorporates prior knowledge to influence the hypothesis search by altering the objective function to be minimized by gradient descent. This corresponds to altering the goal of the hypothesis space search, as illustrated in Figure 12.6. Like BACKPROPAGATION (but unlike KBANN), TANGENTPROP begins the search with an initial network of small random weights. However, the gradient descent training rule produces different weight updates than BACKPROPAGATION, resulting in a different final hypothesis. As shown in the figure, the set of hypotheses that minimizes the TANGENTPROP objective may differ from the set that minimizes the BACKPROPAGATION objective. Importantly, if the training examples and prior knowledge are both correct, and the target function can be accurately represented by the ANN, then the set of weight vectors that satisfy the TANGENTPROP objective will be a subset of those satisfying the weaker BACKPROPAGATION objective. The difference between these two sets of final hypotheses is the set of incorrect hypotheses that will be considered by BACKPROPAGATION, but ruled out by TANGENTPROP due to its prior knowledge.

Note one alternative to fitting the training derivatives of the target function is to simply synthesize additional training examples near the observed training examples, using the known training derivatives to estimate training values for these nearby instances. For example, one could take a training image in the above character recognition task, translate it a small amount, and assert that the trans-

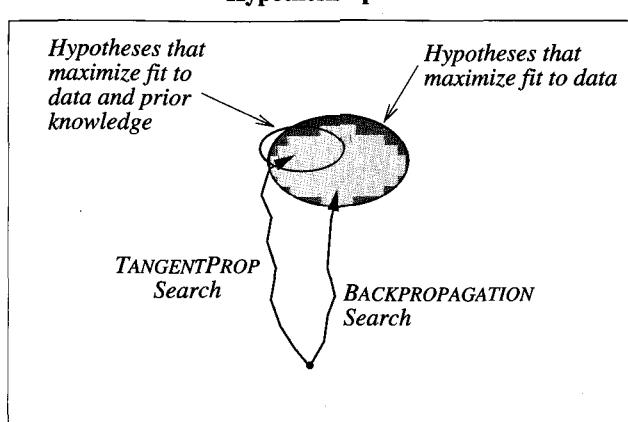


FIGURE 12.6

Hypothesis space search in TANGENTPROP. TANGENTPROP initializes the network to small random weights, just as in BACKPROPAGATION. However, it uses a different error function to drive the gradient descent search. The error used by TANGENTPROP includes both the error in predicting training *values* and in predicting the training *derivatives* provided as prior knowledge.

lated image belonged to the same class as the original example. We might expect that fitting these synthesized examples using BACKPROPAGATION would produce results similar to fitting the original training examples and derivatives using TANGENTPROP. Simard et al. (1992) report experiments showing similar generalization error in the two cases, but report that TANGENTPROP converges considerably more efficiently. It is interesting to note that the ALVINN system, which learns to steer an autonomous vehicle (see Chapter 4), uses a very similar approach to synthesize additional training examples. It uses prior knowledge of how the desired steering direction changes with horizontal translation of the camera image to create multiple synthetic training examples to augment each observed training example.

12.4.4 The EBNN Algorithm

The EBNN (Explanation-Based Neural Network learning) algorithm (Mitchell and Thrun 1993a; Thrun 1996) builds on the TANGENTPROP algorithm in two significant ways. First, instead of relying on the user to provide training derivatives, EBNN computes training derivatives itself for each observed training example. These training derivatives are calculated by explaining each training example in terms of a given domain theory, then extracting training derivatives from this explanation. Second, EBNN addresses the issue of how to weight the relative importance of the inductive and analytical components of learning (i.e., how to select the parameter μ in Equation [12.1]). The value of μ is chosen independently for each training example, based on a heuristic that considers how accurately the domain theory predicts the training value for this particular example. Thus, the analytical component of learning is emphasized for those training examples that are correctly

explained by the domain theory and de-emphasized for training examples that are poorly explained.

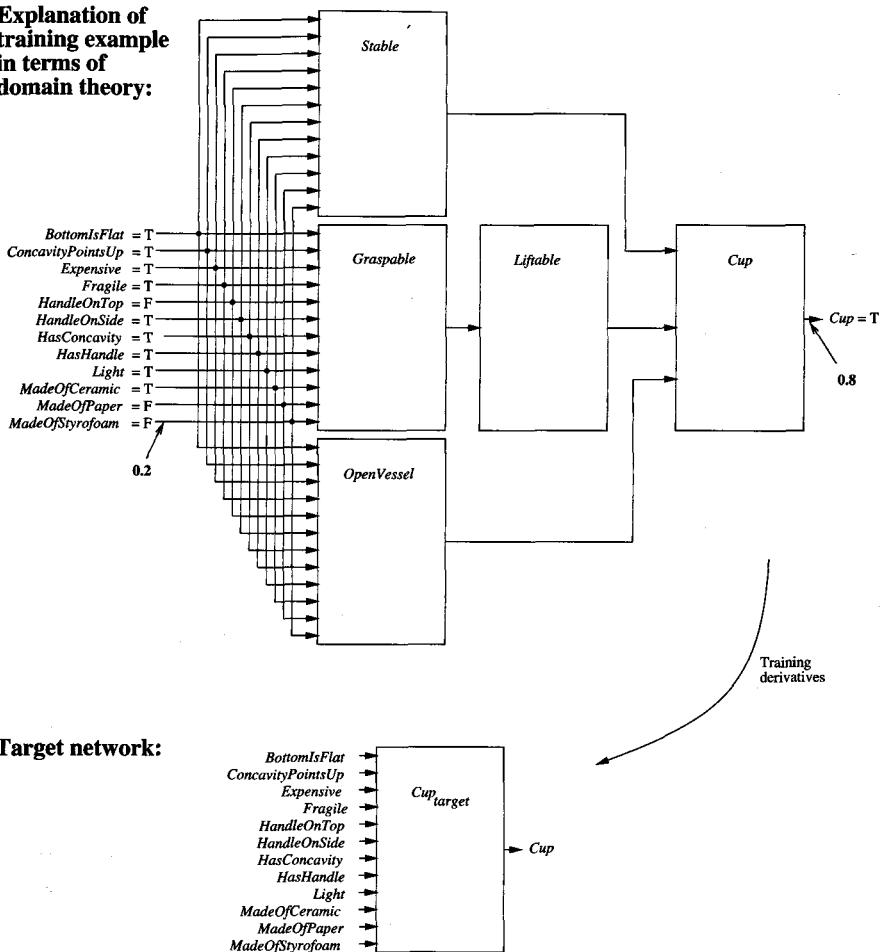
The inputs to EBNN include (1) a set of training examples of the form $\langle x_i, f(x_i) \rangle$ with no training derivatives provided, and (2) a domain theory analogous to that used in explanation-based learning (Chapter 11) and in KBANN, but represented by a set of previously trained neural networks rather than a set of Horn clauses. The output of EBNN is a new neural network that approximates the target function f . This learned network is trained to fit both the training examples $\langle x_i, f(x_i) \rangle$ and training derivatives of f extracted from the domain theory. Fitting the training examples $\langle x_i, f(x_i) \rangle$ constitutes the inductive component of learning, whereas fitting the training derivatives extracted from the domain theory provides the analytical component.

To illustrate the type of domain theory used by EBNN, consider Figure 12.7. The top portion of this figure depicts an EBNN domain theory for the target function *Cup*, with each rectangular block representing a distinct neural network in the domain theory. Notice in this example there is one network for each of the Horn clauses in the symbolic domain theory of Table 12.3. For example, the network labeled *Graspable* takes as input the description of an instance and produces as output a value indicating whether the object is graspable (EBNN typically represents true propositions by the value 0.8 and false propositions by the value 0.2). This network is analogous to the Horn clause for *Graspable* given in Table 12.3. Some networks take the outputs of other networks as their inputs (e.g., the right-most network labeled *Cup* takes its inputs from the outputs of the *Stable*, *Liftable*, and *OpenVessel* networks). Thus, the networks that make up the domain theory can be chained together to infer the target function value for the input instance, just as Horn clauses might be chained together for this purpose. In general, these domain theory networks may be provided to the learner by some external source, or they may be the result of previous learning by the same system. EBNN makes use of these domain theory networks to learn the new target function. It does not alter the domain theory networks during this process.

The goal of EBNN is to learn a new neural network to describe the target function. We will refer to this new network as the *target network*. In the example of Figure 12.7, the target network Cup_{target} shown at the bottom of the figure takes as input the description of an arbitrary instance and outputs a value indicating whether the object is a *Cup*.

EBNN learns the target network by invoking the TANGENTPROP algorithm described in the previous section. Recall that TANGENTPROP trains a network to fit both training values and training derivatives. EBNN passes along to TANGENTPROP the training values $\langle x_i, f(x_i) \rangle$ that it receives as input. In addition, EBNN provides TANGENTPROP with derivatives that it calculates from the domain theory. To see how EBNN calculates these training derivatives, consider again Figure 12.7. The top portion of this figure shows the domain theory prediction of the target function value for a particular training instance, x_i . EBNN calculates the derivative of this prediction with respect to each feature of the input instance. For the example in the figure, the instance x_i is described by features such as *MadeOfStyrofoam* = 0.2

Explanation of training example in terms of domain theory:



Target network:

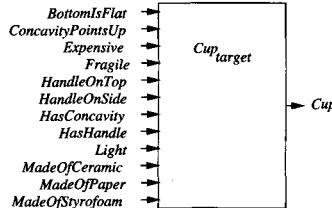


FIGURE 12.7

Explanation of a training example in EBNN. The explanation consists of a prediction of the target function value by the domain theory networks (top). Training derivatives are extracted from this explanation in order to train the separate target network (bottom). Each rectangular block represents a distinct multilayer neural network.

(i.e., *False*), and the domain theory prediction is that *Cup* = 0.8 (i.e., *True*). EBNN calculates the partial derivative of this prediction with respect to each instance feature, yielding the set of derivatives

$$\left[\frac{\partial \text{Cup}}{\partial \text{BottomIsFlat}}, \frac{\partial \text{Cup}}{\partial \text{ConcavityPointsUp}}, \dots, \frac{\partial \text{Cup}}{\partial \text{MadeOfStyrofoam}} \right]_{x=x_i}$$

This set of derivatives is the gradient of the domain theory prediction function with respect to the input instance. The subscript refers to the fact that these derivatives

hold when $x = x_i$. In the more general case where the target function has multiple output units, the gradient is computed for each of these outputs. This matrix of gradients is called the Jacobian of the target function.

To see the importance of these training derivatives in helping to learn the target network, consider the derivative $\frac{\partial \text{Cup}}{\partial \text{Expensive}}$. If the domain theory encodes the knowledge that the feature *Expensive* is irrelevant to the target function *Cup*, then the derivative $\frac{\partial \text{Cup}}{\partial \text{Expensive}}$ extracted from the explanation will have the value zero. A derivative of zero corresponds to the assertion that a change in the feature *Expensive* will have no impact on the predicted value of *Cup*. On the other hand, a large positive or negative derivative corresponds to the assertion that the feature is highly relevant to determining the target value. Thus, the derivatives extracted from the domain theory explanation provide important information for distinguishing relevant from irrelevant features. When these extracted derivatives are provided as training derivatives to TANGENTPROP for learning the target network $\text{Cup}_{\text{target}}$, they provide a useful bias for guiding generalization. The usual syntactic inductive bias of neural network learning is replaced in this case by the bias exerted by the derivatives obtained from the domain theory.

Above we described how the domain theory prediction can be used to generate a set of training derivatives. To be more precise, the full EBNN algorithm is as follows. Given the training examples and domain theory, EBNN first creates a new, fully connected feedforward network to represent the target function. This target network is initialized with small random weights, just as in BACK-PROPAGATION. Next, for each training example $\langle x_i, f(x_i) \rangle$ EBNN determines the corresponding training derivatives in a two-step process. First, it uses the domain theory to predict the value of the target function for instance x_i . Let $A(x_i)$ denote this domain theory prediction for instance x_i . In other words, $A(x_i)$ is the function defined by the composition of the domain theory networks forming the explanation for x_i . Second, the weights and activations of the domain theory networks are analyzed to extract the derivatives of $A(x_i)$ with respect to each of the components of x_i (i.e., the Jacobian of $A(x)$ evaluated at $x = x_i$). Extracting these derivatives follows a process very similar to calculating the δ terms in the BACK-PROPAGATION algorithm (see Exercise 12.5). Finally, EBNN uses a minor variant of the TANGENTPROP algorithm to train the target network to fit the following error function

$$E = \sum_i \left[(f(x_i) - \hat{f}(x_i))^2 + \mu_i \sum_j \left(\frac{\partial A(x)}{\partial x^j} - \frac{\partial \hat{f}(x)}{\partial x^j} \right)^2_{(x=x_i)} \right] \quad (12.2)$$

where

$$\mu_i \equiv 1 - \frac{|A(x_i) - f(x_i)|}{c} \quad (12.3)$$

Here x_i denotes the i th training instance and $A(x)$ denotes the domain theory prediction for input x . The superscript notation x^j denotes the j th component of the vector x (i.e., the j th input node of the neural network). The coefficient c is a normalizing constant whose value is chosen to assure that for all i , $0 \leq \mu_i \leq 1$.

Although the notation here appears a bit tedious, the idea is simple. The error given by Equation (12.2) has the same general form as the error function in Equation (12.1) minimized by TANGENTPROP. The leftmost term measures the usual sum of squared errors between the training value $f(x_i)$ and the value predicted by the target network $\hat{f}(x_i)$. The rightmost term measures the squared error between the training derivatives $\frac{\partial A(x)}{\partial x^j}$ extracted from the domain theory and the actual derivatives of the target network $\frac{\partial \hat{f}(x)}{\partial x^j}$. Thus, the leftmost term contributes the inductive constraint that the hypothesis must fit the observed training data, whereas the rightmost term contributes the analytical constraint that it must fit the training derivatives extracted from the domain theory. Notice the derivative $\frac{\partial \hat{f}(x)}{\partial x^j}$ in Equation (12.2) is just a special case of the expression $\frac{\partial \hat{f}(s_j(\alpha, x_i))}{\partial \alpha}$ of Equation (12.1), for which $s_j(\alpha, x_i)$ is the transformation that replaces x_i^j by $x_i^j + \alpha$. The precise weight-training rule used by EBNN is described by Thrun (1996).

The relative importance of the inductive and analytical learning components is determined in EBNN by the constant μ_i , defined in Equation (12.3). The value of μ_i is determined by the discrepancy between the domain theory prediction $A(x_i)$ and the training value $f(x_i)$. The analytical component of learning is thus weighted more heavily for training examples that are correctly predicted by the domain theory and is suppressed for examples that are not correctly predicted. This weighting heuristic assumes that the training *derivatives* extracted from the domain theory are more likely to be correct in cases where the training *value* is correctly predicted by the domain theory. Although one can construct situations in which this heuristic fails, in practice it has been found effective in several domains (e.g., see Mitchell and Thrun [1993a]; Thrun [1996]).

12.4.5 Remarks

To summarize, the EBNN algorithm uses a domain theory expressed as a set of previously learned neural networks, together with a set of training examples, to train its output hypothesis (the target network). For each training example EBNN uses its domain theory to explain the example, then extracts training derivatives from this explanation. For each attribute of the instance, a training derivative is computed that describes how the target function value is influenced by a small change to this attribute value, according to the domain theory. These training derivatives are provided to a variant of TANGENTPROP, which fits the target network to these derivatives and to the training example values. Fitting the derivatives constrains the learned network to fit dependencies given by the domain theory, while fitting the training values constrains it to fit the observed data itself. The weight μ_i placed on fitting the derivatives is determined independently for each training example, based on how accurately the domain theory predicts the training value for this example.

EBNN has been shown to be an effective method for learning from approximate domain theories in several domains. Thrun (1996) describes its application to a variant of the *Cup* learning task discussed above and reports that

EBNN generalizes more accurately than standard BACKPROPAGATION, especially when training data is scarce. For example, after 30 training examples, EBNN achieved a root-mean-squared error of 5.5 on a separate set of test data, compared to an error of 12.0 for BACKPROPAGATION. Mitchell and Thrun (1993a) describe applying EBNN to learning to control a simulated mobile robot, in which the domain theory consists of neural networks that predict the effects of various robot actions on the world state. Again, EBNN using an approximate, previously learned domain theory, outperformed BACKPROPAGATION. Here BACKPROPAGATION required approximately 90 training episodes to reach the level of performance achieved by EBNN after 25 training episodes. O'Sullivan et al. (1997) and Thrun (1996) describe several other applications of EBNN to real-world robot perception and control tasks, in which the domain theory consists of networks that predict the effect of actions for an indoor mobile robot using sonar, vision, and laser range sensors.

EBNN bears an interesting relation to other explanation-based learning methods, such as PROLOG-EBG described in Chapter 11. Recall from that chapter that PROLOG-EBG also constructs explanations (predictions of example target values) based on a domain theory. In PROLOG-EBG the explanation is constructed from a domain theory consisting of Horn clauses, and the target hypothesis is refined by calculating the weakest conditions under which this explanation holds. Relevant dependencies in the explanation are thus captured in the learned Horn clause hypothesis. EBNN constructs an analogous explanation, but it is based on a domain theory consisting of neural networks rather than Horn clauses. As in PROLOG-EBG, relevant dependencies are then extracted from the explanation and used to refine the target hypothesis. In the case of EBNN, these dependencies take the form of derivatives because derivatives are the natural way to represent dependencies in continuous functions such as neural networks. In contrast, the natural way to represent dependencies in symbolic explanations or logical proofs is to describe the set of examples to which the proof applies.

There are several differences in capabilities between EBNN and the symbolic explanation-based methods of Chapter 11. The main difference is that EBNN accommodates imperfect domain theories, whereas PROLOG-EBG does not. This difference follows from the fact that EBNN is built on the inductive mechanism of fitting the observed training values and uses the domain theory only as an additional constraint on the learned hypothesis. A second important difference follows from the fact that PROLOG-EBG learns a growing set of Horn clauses, whereas EBNN learns a fixed-size neural network. As discussed in Chapter 11, one difficulty in learning sets of Horn clauses is that the cost of classifying a new instance grows as learning proceeds and new Horn clauses are added. This problem is avoided in EBNN because the fixed-size target network requires constant time to classify new instances. However, the fixed-size neural network suffers the corresponding disadvantage that it may be unable to represent sufficiently complex functions, whereas a growing set of Horn clauses can represent increasingly complex functions. Mitchell and Thrun (1993b) provide a more detailed discussion of the relationship between EBNN and symbolic explanation-based learning methods.

12.5 USING PRIOR KNOWLEDGE TO AUGMENT SEARCH OPERATORS

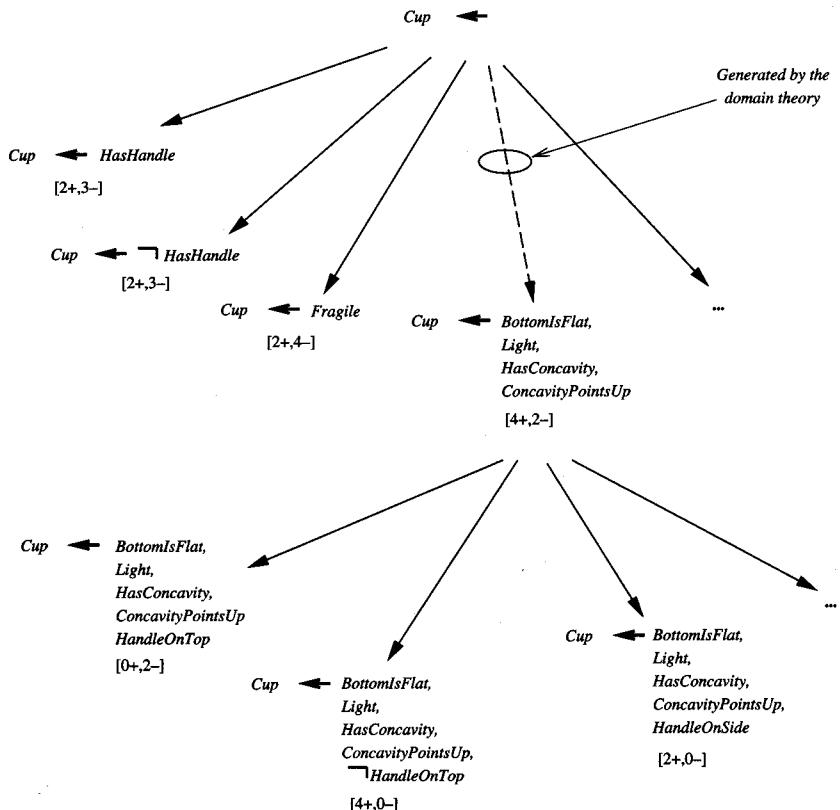
The two previous sections examined two different roles for prior knowledge in learning: initializing the learner's hypothesis and altering the objective function that guides search through the hypothesis space. In this section we consider a third way of using prior knowledge to alter the hypothesis space search: using it to alter the set of operators that define legal steps in the search through the hypothesis space. This approach is followed by systems such as FOCL (Pazzani et al. 1991; Pazzani and Kibler 1992) and ML-SMART (Bergadano and Giordana 1990). Here we use FOCL to illustrate the approach.

12.5.1 The FOCL Algorithm

FOCL is an extension of the purely inductive FOIL system described in Chapter 10. Both FOIL and FOCL learn a set of first-order Horn clauses to cover the observed training examples. Both systems employ a sequential covering algorithm that learns a single Horn clause, removes the positive examples covered by this new Horn clause, and then iterates this procedure over the remaining training examples. In both systems, each new Horn clause is created by performing a general-to-specific search, beginning with the most general possible Horn clause (i.e., a clause containing no preconditions). Several candidate specializations of the current clause are then generated, and the specialization with greatest information gain relative to the training examples is chosen. This process is iterated, generating further candidate specializations and selecting the best, until a Horn clause with satisfactory performance is obtained.

The difference between FOIL and FOCL lies in the way in which candidate specializations are generated during the general-to-specific search for a single Horn clause. As described in Chapter 10, FOIL generates each candidate specialization by adding a single new literal to the clause preconditions. FOCL uses this same method for producing candidate specializations, but also generates additional specializations based on the domain theory. The *solid* edges in the search tree of Figure 12.8 show the general-to-specific search steps considered in a typical search by FOIL. The *dashed* edge in the search tree of Figure 12.8 denotes an additional candidate specialization that is considered by FOCL and based on the domain theory.

Although FOCL and FOIL both learn first-order Horn clauses, we illustrate their operation here using the simpler domain of propositional (variable-free) Horn clauses. In particular, consider again the *Cup* target concept, training examples, and domain theory from Figure 12.3. To describe the operation of FOCL, we must first draw a distinction between two kinds of literals that appear in the domain theory and hypothesis representation. We will say a literal is *operational* if it is allowed to be used in describing an output hypothesis. For example, in the *Cup* example of Figure 12.3 we allow output hypotheses to refer only to the 12 attributes that describe the training examples (e.g., *HasHandle*, *HandleOnTop*). Literals based on these 12 attributes are thus considered operational. In contrast, literals that occur only as intermediate features in the domain theory, but not as

**FIGURE 12.8**

Hypothesis space search in FOCL. To learn a single rule, FOCL searches from general to increasingly specific hypotheses. Two kinds of operators generate specializations of the current hypothesis. One kind adds a single new literal (solid lines in the figure). A second kind of operator specializes the rule by adding a set of literals that constitute logically sufficient conditions for the target concept, according to the domain theory (dashed lines in the figure). FOCL selects among all these candidate specializations, based on their performance over the data. Therefore, imperfect domain theories will impact the hypothesis only if the evidence supports the theory. This example is based on the same training data and domain theory as the earlier KBANN example.

primitive attributes of the instances, are considered nonoperational. An example of a nonoperational attribute in this case is the attribute *Stable*.

At each point in its general-to-specific search, FOCL expands its current hypothesis h using the following two operators:

1. For each *operational* literal that is not part of h , create a specialization of h by adding this single literal to the preconditions. This is also the method used by FOIL to generate candidate successors. The solid arrows in Figure 12.8 denote this type of specialization.

2. Create an operational, logically sufficient condition for the target concept according to the domain theory. Add this set of literals to the current preconditions of h . Finally, prune the preconditions of h by removing any literals that are unnecessary according to the training data. The dashed arrow in Figure 12.8 denotes this type of specialization.

The detailed procedure for the second operator above is as follows. FOCL first selects one of the domain theory clauses whose head (postcondition) matches the target concept. If there are several such clauses, it selects the clause whose body (preconditions) have the highest information gain relative to the training examples of the target concept. For example, in the domain theory and training data of Figure 12.3, there is only one such clause:

$$\text{Cup} \leftarrow \text{Stable}, \text{Liftable}, \text{OpenVessel}$$

The preconditions of the selected clause form a logically sufficient condition for the target concept. Each nonoperational literal in these sufficient conditions is now replaced, again using the domain theory and substituting clause preconditions for clause postconditions. For example, the domain theory clause $\text{Stable} \leftarrow \text{BottomIsFlat}$ is used to substitute the operational BottomIsFlat for the unoperational Stable . This process of “unfolding” the domain theory continues until the sufficient conditions have been restated in terms of operational literals. If there are several alternative domain theory clauses that produce different results, then the one with the greatest information gain is greedily selected at each step of the unfolding process. The reader can verify that the final operational sufficient condition given the data and domain theory in the current example is

$$\text{BottomIsFlat}, \text{HasHandle}, \text{Light}, \text{HasConcavity}, \text{ConcavityPointsUp}$$

As a final step in generating the candidate specialization, this sufficient condition is pruned. For each literal in the expression, the literal is removed unless its removal reduces classification accuracy over the training examples. This step is included to recover from overspecialization in case the imperfect domain theory includes irrelevant literals. In our current example, the above set of literals matches two positive and two negative examples. Pruning (removing) the literal HasHandle results in improved performance. The final pruned, operational, sufficient conditions are, therefore,

$$\text{BottomIsFlat}, \text{Light}, \text{HasConcavity}, \text{ConcavityPointsUp}$$

This set of literals is now added to the preconditions of the current hypothesis. Note this hypothesis is the result of the search step shown by the dashed arrow in Figure 12.8.

Once candidate specializations of the current hypothesis have been generated, using both of the two operations above, the candidate with highest information gain is selected. In the example shown in Figure 12.8 the candidate chosen at the first level in the search tree is the one generated by the domain theory. The search then proceeds by considering further specializations of the theory-suggested

preconditions, thereby allowing the inductive component of learning to refine the preconditions derived from the domain theory. In this example, the domain theory affects the search only at the first search level. However, this will not always be the case. Should the empirical support be stronger for some other candidate at the first level, theory-suggested literals may still be added at subsequent steps in the search. To summarize, FOCL learns Horn clauses of the form

$$c \leftarrow o_i \wedge o_b \wedge o_f$$

where c is the target concept, o_i is an initial conjunction of operational literals added one at a time by the first syntactic operator, o_b is a conjunction of operational literals added in a single step based on the domain theory, and o_f is a final conjunction of operational literals added one at a time by the first syntactic operator. Any of these three sets of literals may be empty.

The above discussion illustrates the use of a propositional domain theory to create candidate specializations of the hypothesis during the general-to-specific search for a single Horn clause. The algorithm is easily extended to first-order representations (i.e., representations including variables). Chapter 10 discusses in detail the algorithm used by FOIL to generate first-order Horn clauses, including the extension of the first of the two search operators described above to first-order representations. To extend the second operator to accommodate first-order domain theories, variable substitutions must be considered when unfolding the domain theory. This can be accomplished using a procedure related to the regression procedure described in Table 11.3.

12.5.2 Remarks

FOCL uses the domain theory to increase the number of candidate specializations considered at each step of the search for a single Horn clause. Figure 12.9 compares the hypothesis space search performed by FOCL to that performed by the purely inductive FOIL algorithm on which it is based. FOCL's theory-suggested specializations correspond to "macro" steps in FOIL's search, in which several literals are added in a single step. This process can be viewed as promoting a hypothesis that might be considered later in the search to one that will be considered immediately. If the domain theory is correct, the training data will bear out the superiority of this candidate over the others and it will be selected. If the domain theory is incorrect, the empirical evaluation of all the candidates should direct the search down an alternative path.

To summarize, FOCL uses both a syntactic generation of candidate specializations and a domain theory driven generation of candidate specializations at each step in the search. The algorithm chooses among these candidates based solely on their empirical support over the training data. Thus, the domain theory is used in a fashion that biases the learner, but leaves final search choices to be made based on performance over the training data. The bias introduced by the domain theory is a preference in favor of Horn clauses most similar to operational, logically sufficient conditions entailed by the domain theory. This bias is combined with

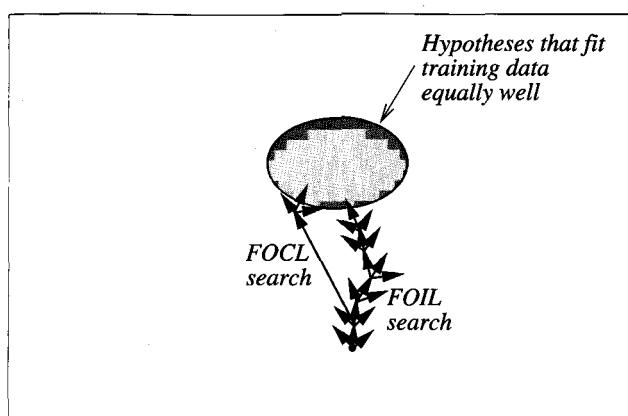


FIGURE 12.9

Hypothesis space search in FOCL. FOCL augments the set of search operators used by FOIL. Whereas FOIL considers adding a single new literal at each step, FOCL also considers adding multiple literals derived from the domain theory.

the bias of the purely inductive FOIL program, which is a preference for shorter hypotheses.

FOCL has been shown to generalize more accurately than the purely inductive FOIL algorithm in a number of application domains in which an imperfect domain theory is available. For example, Pazzani and Kibler (1992) explore learning the concept “legal chessboard positions.” Given 60 training examples describing 30 legal and 30 illegal endgame board positions, FOIL achieved an accuracy of 86% over an independent set of test examples. FOCL was given the same 60 training examples, along with an approximate domain theory with an accuracy of 76%. FOCL produced a hypothesis with generalization accuracy of 94%—less than half the error rate of FOIL. Similar results have been obtained in other domains. For example, given 500 training examples of telephone network problems and their diagnoses from the telephone company NYNEX, FOIL achieved an accuracy of 90%, whereas FOCL reached an accuracy of 98% when given the same training data along with a 95% accurate domain theory.

12.6 STATE OF THE ART

The methods presented in this chapter are only a sample of the possible approaches to combining analytical and inductive learning. While each of these methods has been demonstrated to outperform purely inductive learning methods in selected domains, none of these has been thoroughly tested or proven across a large variety of problem domains. The topic of combining inductive and analytical learning remains a very active research area.

12.7 SUMMARY AND FURTHER READING

The main points of this chapter include:

- Approximate prior knowledge, or domain theories, are available in many practical learning problems. Purely inductive methods such as decision tree induction and neural network BACKPROPAGATION fail to utilize such domain theories, and therefore perform poorly when data is scarce. Purely analytical learning methods such as PROLOG-EBG utilize such domain theories, but produce incorrect hypotheses when given imperfect prior knowledge. Methods that blend inductive and analytical learning can gain the benefits of both approaches: reduced sample complexity and the ability to overrule incorrect prior knowledge.
- One way to view algorithms for combining inductive and analytical learning is to consider how the domain theory affects the hypothesis space search. In this chapter we examined methods that use imperfect domain theories to (1) create the initial hypothesis in the search, (2) expand the set of search operators that generate revisions to the current hypothesis, and (3) alter the objective of the search.
- A system that uses the domain theory to initialize the hypothesis is KBANN. This algorithm uses a domain theory encoded as propositional rules to analytically construct an artificial neural network that is equivalent to the domain theory. This network is then inductively refined using the BACKPROPAGATION algorithm, to improve its performance over the training data. The result is a network biased by the original domain theory, whose weights are refined inductively based on the training data.
- TANGENTPROP uses prior knowledge represented by desired derivatives of the target function. In some domains, such as image processing, this is a natural way to express prior knowledge. TANGENTPROP incorporates this knowledge by altering the objective function minimized by gradient descent search through the space of possible hypotheses.
- EBNN uses the domain theory to alter the objective in searching the hypothesis space of possible weights for an artificial neural network. It uses a domain theory consisting of previously learned neural networks to perform a neural network analog to symbolic explanation-based learning. As in symbolic explanation-based learning, the domain theory is used to explain individual examples, yielding information about the relevance of different example features. With this neural network representation, however, information about relevance is expressed in the form of derivatives of the target function value with respect to instance features. The network hypothesis is trained using a variant of the TANGENTPROP algorithm, in which the error to be minimized includes both the error in network output values and the error in network derivatives obtained from explanations.
- FOCL uses the domain theory to expand the set of candidates considered at each step in the search. It uses an approximate domain theory represented

by first order Horn clauses to learn a set of Horn clauses that approximate the target function. FOCL employs a sequential covering algorithm, learning each Horn clause by a general-to-specific search. The domain theory is used to augment the set of next more specific candidate hypotheses considered at each step of this search. Candidate hypotheses are then evaluated based on their performance over the training data. In this way, FOCL combines the greedy, general-to-specific inductive search strategy of FOIL with the rule-chaining, analytical reasoning of analytical methods.

- The question of how to best blend prior knowledge with new observations remains one of the key open questions in machine learning.

There are many more examples of algorithms that attempt to combine inductive and analytical learning. For example, methods for learning Bayesian belief networks discussed in Chapter 6 provide one alternative to the approaches discussed here. The references at the end of this chapter provide additional examples and sources for further reading.

EXERCISES

- 12.1.** Consider learning the target concept *GoodCreditRisk* defined over instances described by the four attributes *HasStudentLoan*, *HasSavingsAccount*, *IsStudent*, *OwnsCar*. Give the initial network created by KBANN for the following domain theory, including all network connections and weights.

$$\begin{aligned} \textit{GoodCreditRisk} &\leftarrow \textit{Employed}, \textit{LowDebt} \\ \textit{Employed} &\leftarrow \neg \textit{IsStudent} \\ \textit{LowDebt} &\leftarrow \neg \textit{HasStudentLoan}, \textit{HasSavingsAccount} \end{aligned}$$

- 12.2.** KBANN converts a set of propositional Horn clauses into an initial neural network. Consider the class of *n-of-m clauses*, which are Horn clauses containing *m* literals in the preconditions (antecedents), and an associated parameter *n* where $n \leq m$. The preconditions of an *n-of-m* Horn clause are considered to be satisfied if at least *n* of its *m* preconditions are satisfied. For example, the clause

$$\textit{Student} \leftarrow \textit{LivesInDorm}, \textit{Young}, \textit{Studies}; \quad n = 2$$

asserts that one is a *Student* if at least two of these three preconditions are satisfied.

Give an algorithm similar to that used by KBANN, that accepts a set of propositional *n-of-m* clauses and constructs a neural network consistent with the domain theory.

- 12.3.** Consider extending KBANN to accept a domain theory consisting of first-order rather than propositional Horn clauses (i.e., Horn clauses containing variables, as in Chapter 10). Either give an algorithm for constructing a neural network equivalent to a set of Horn clauses, or discuss the difficulties that prevent this.
- 12.4.** This exercise asks you to derive a gradient descent rule analogous to that used by TANGENTPROP. Consider the instance space *X* consisting of the real numbers, and consider the hypothesis space *H* consisting of quadratic functions of *x*. That is,

each hypothesis $h(x)$ is of the form

$$h(x) = w_0 + w_1x + w_2x^2$$

(a) Derive a gradient descent rule that minimizes the same criterion as BACKPROPAGATION; that is, the sum of squared errors between the hypothesis and target values of the training data.

(b) Derive a second gradient descent rule that minimizes the same criterion as TANGENTPROP. Consider only the single transformation $s(\alpha, x) = x + \alpha$.

12.5. EBNN extracts training derivatives from explanations by examining the weights and activations of the neural networks that make up the explanation. Consider the simple example in which the explanation is formed by a single sigmoid unit with n inputs. Derive a procedure for extracting the derivative $\frac{\partial \hat{f}(x)}{\partial x^j}|_{x=x_i}$ where x_i is a particular training instance input to the unit, $\hat{f}(x)$ is the sigmoid unit output, and x^j denotes the j th input to the sigmoid unit. You may wish to use the notation x_i^j to refer to the j th component of x_i . Hint: The derivation is similar to the derivation of the BACKPROPAGATION training rule.

12.6. Consider again the search trace of FOCL shown in Figure 12.8. Suppose that the hypothesis selected at the first level in the search is changed to

$$Cup \leftarrow \neg HasHandle$$

Describe the second-level candidate hypotheses that will be generated by FOCL as successors to this hypothesis. You need only include those hypotheses generated by FOCL's second search operator, which uses its domain theory. Don't forget to post-prune the sufficient conditions. Use the training data from Table 12.3.

12.7. This chapter discussed three approaches to using prior knowledge to impact the search through the space of possible hypotheses. Discuss your ideas for how these three approaches could be integrated. Can you propose a specific algorithm that integrates at least two of these three for some specific hypothesis representation? What advantages and disadvantages would you anticipate from this integration?

12.8. Consider again the question from Section 12.2.1, regarding what criterion to use for choosing among hypotheses when both data and prior knowledge are available. Give your own viewpoint on this issue.

REFERENCES

- Abu-Mostafa, Y. S. (1989). Learning from hints in neural networks. *Journal of Complexity*, 6(2), 192–198.
- Bergadano, F., & Giordana, A. (1990). Guiding induction with domain theories. In R. Michalski et al. (Eds.), *Machine learning: An artificial intelligence approach 3* (pp. 474–492). San Mateo, CA: Morgan Kaufmann.
- Bradshaw, G., Fozzard, R., & Cice, L. (1989). A connectionist expert system that really works. In *Advances in neural information processing*. San Mateo, CA: Morgan Kaufmann.
- Caruana, R. (1996). Algorithms and applications for multitask learning. *Proceedings of the 13th International Conference on Machine Learning*. San Francisco: Morgan Kaufmann.
- Cooper, G. C., & Herskovits, E. (1992). A Bayesian method for the induction of probabilistic networks from data. *Machine Learning*, 9, 309–347.
- Craven, M. W. (1996). *Extracting comprehensible models from trained neural networks* (PhD thesis) (UW Technical Report CS-TR-96-1326). Department of Computer Sciences, University of Wisconsin-Madison.

- Craven, M. W., & Shavlik, J. W. (1994). Using sampling and queries to extract rules from trained neural networks. *Proceedings of the 11th International Conference on Machine Learning* (pp. 37–45). San Mateo, CA: Morgan Kaufmann.
- Fu, L. M. (1989). Integration of neural heuristics into knowledge-based inference. *Connection Science*, 1(3), 325–339.
- Fu, L. M. (1993). Knowledge-based connectionism for revising domain theories. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(1), 173–182.
- Gallant, S. I. (1988). Connectionist expert systems. *CACM*, 31(2), 152–169.
- Koppel, M., Feldman, R., & Segre, A. (1994). Bias-driven revision of logical domain theories. *Journal of Artificial Intelligence*, 1, 159–208. <http://www.cs.washington.edu/research/jair/home.html>.
- Lacher, R., Hruska, S., & Kuncicky, D. (1991). *Backpropagation learning in expert networks* (Dept. of Computer Science Technical Report TR91-015). Florida State University, Tallahassee.
- Maclin, R., & Shavlik, J. (1993). Using knowledge-based neural networks to improve algorithms: Refining the Chou-Fasman algorithm for protein folding. *Machine Learning*, 11(3), 195–215.
- Mitchell, T. M., & Thrun, S. B. (1993a). Explanation-based neural network learning for robot control. In S. Hanson, J. Cowan, & C. Giles (Eds.), *Advances in neural information processing systems 5* (pp. 287–294). San Mateo, CA: Morgan-Kaufmann Press.
- Mitchell, T. M., & Thrun, S. B. (1993b). Explanation-based learning: A comparison of symbolic and neural network approaches. *Tenth International Conference on Machine Learning*, Amherst, MA.
- Mooney, R. (1993). Induction over the unexplained: Using overly-general domain theories to aid concept learning. *Machine Learning*, 10(1).
- O’Sullivan, J., Mitchell, T., & Thrun, S. (1997). Explanation-based learning for mobile robot perception. In K. Ikeuchi & M. Veloso (Eds.), *Symbolic Visual Learning* (pp. 295–324).
- Ourston, D., & Mooney, R. J. (1994). Theory refinement combining analytical and empirical methods. *Artificial Intelligence*, 66(2).
- Pazzani, M. J., & Brunk, C. (1993). Finding accurate frontiers: A knowledge-intensive approach to relational learning. *Proceedings of the 1993 National Conference on Artificial Intelligence* (pp. 328–334). AAAI Press.
- Pazzani, M. J., Brunk, C. A., & Silverstein, G. (1991). A knowledge-intensive approach to learning relational concepts. *Proceedings of the Eighth International Workshop on Machine Learning* (pp. 432–436). San Mateo, CA: Morgan Kaufmann.
- Pazzani, M. J., & Kibler, D. (1992). The utility of knowledge in inductive learning. *Machine Learning*, 9(1), 57–94.
- Pratt, L. Y. (1993a). *Transferring previously learned BACKPROPAGATION neural networks to new learning tasks* (Ph.D. thesis). Department of Computer Science, Rutgers University, New Jersey. (Also Rutgers Computer Science Technical Report ML-TR-37.)
- Pratt, L. Y. (1993b). Discriminability-based transfer among neural networks. In J. E. Moody et al. (Eds.), *Advances in Neural Information Processing Systems 5*. San Mateo, CA: Morgan Kaufmann.
- Rosenbloom, P. S., & Aasman, J. (1990). Knowledge level and inductive uses of chunking (ebL). *Proceedings of the Eighth National Conference on Artificial Intelligence* (pp. 821–827). AAAI Press.
- Russell, S., Binder, J., Koller, D., & Kanazawa, K. (1995). Local learning in probabilistic networks with hidden variables. *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, Montreal. Morgan Kaufmann.
- Shavlik, J., & Towell, G. (1989). An approach to combining explanation-based and neural learning algorithms. *Connection Science*, 1(3), 233–255.
- Simard, P. S., Victorri, B., LeCun, Y., & Denker, J. (1992). Tangent prop—A formalism for specifying selected invariances in an adaptive network. In J. Moody et al. (Eds.), *Advances in Neural Information Processing Systems 4*. San Mateo, CA: Morgan Kaufmann.
- Sudharth, S. C., & Holden, A. D. C. (1991). Symbolic-neural systems and the use of hints for developing complex systems. *International Journal of Man-Machine Studies*, 35(3), 291–311.

- Thrun, S. (1996). *Explanation based neural network learning: A lifelong learning approach*. Boston: Kluwer Academic Publishers.
- Thrun, S., & Mitchell, T. M. (1993). Integrating inductive neural network learning and explanation-based learning. *Proceedings of the 1993 International Joint Conference on Artificial Intelligence*.
- Thrun, S., & Mitchell, T. M. (1995). Learning one more thing. *Proceedings of the 1995 International Joint Conference on Artificial Intelligence*, Montreal.
- Towell, G., & Shavlik, J. (1989). An approach to combining explanation-based and neural learning algorithms. *Connection Science*, (1), 233–255.
- Towell, G., & Shavlik, J. (1994). Knowledge-based artificial neural networks. *Artificial Intelligence*, 70(1–2), 119–165.
- Towell, G., Shavlik, J., & Noordewier, M. (1990). Refinement of approximate domain theories by knowledge-based neural networks. *Proceedings of the Eighth National Conference on Artificial Intelligence* (pp. 861–866). Cambridge, MA: AAAI, MIT Press.
- Yang, Q., & Bhargava, V. (1990). Building expert systems by a modified perceptron network with rule-transfer algorithms (pp. 77–82). *International Joint Conference on Neural Networks*, IEEE.

CHAPTER 13

REINFORCEMENT LEARNING

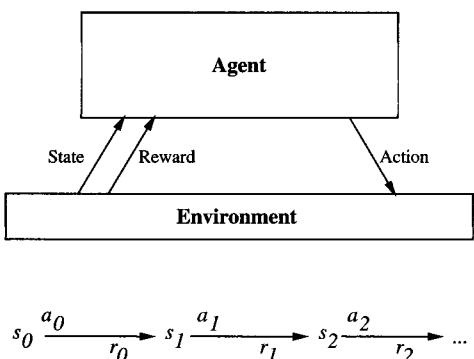
Reinforcement learning addresses the question of how an autonomous agent that senses and acts in its environment can learn to choose optimal actions to achieve its goals. This very generic problem covers tasks such as learning to control a mobile robot, learning to optimize operations in factories, and learning to play board games. Each time the agent performs an action in its environment, a trainer may provide a reward or penalty to indicate the desirability of the resulting state. For example, when training an agent to play a game the trainer might provide a positive reward when the game is won, negative reward when it is lost, and zero reward in all other states. The task of the agent is to learn from this indirect, delayed reward, to choose sequences of actions that produce the greatest cumulative reward. This chapter focuses on an algorithm called Q learning that can acquire optimal control strategies from delayed rewards, even when the agent has no prior knowledge of the effects of its actions on the environment. Reinforcement learning algorithms are related to dynamic programming algorithms frequently used to solve optimization problems.

13.1 INTRODUCTION

Consider building a learning robot. The robot, or *agent*, has a set of sensors to observe the *state* of its environment, and a set of *actions* it can perform to alter this state. For example, a mobile robot may have sensors such as a camera and sonars, and actions such as “move forward” and “turn.” Its task is to learn a control strategy, or *policy*, for choosing actions that achieve its goals. For example, the robot may have a goal of docking onto its battery charger whenever its battery level is low.

This chapter is concerned with how such agents can learn successful control policies by experimenting in their environment. We assume that the goals of the agent can be defined by a *reward* function that assigns a numerical value—an immediate payoff—to each distinct action the agent may take from each distinct state. For example, the goal of docking to the battery charger can be captured by assigning a positive reward (e.g., +100) to state-action transitions that immediately result in a connection to the charger and a reward of zero to every other state-action transition. This reward function may be built into the robot, or known only to an external teacher who provides the reward value for each action performed by the robot. The task of the robot is to perform sequences of actions, observe their consequences, and learn a control policy. The control policy we desire is one that, from any initial state, chooses actions that maximize the reward accumulated over time by the agent. This general setting for robot learning is summarized in Figure 13.1.

As is apparent from Figure 13.1, the problem of learning a control policy to maximize cumulative reward is very general and covers many problems beyond robot learning tasks. In general the problem is one of learning to control sequential processes. This includes, for example, manufacturing optimization problems in which a sequence of manufacturing actions must be chosen, and the reward to be maximized is the value of the goods produced minus the costs involved. It includes sequential scheduling problems such as choosing which taxis to send for passengers in a large city, where the reward to be maximized is a function of the wait time of the passengers and the total fuel costs of the taxi fleet. In general, we are interested in any type of agent that must learn to choose actions that alter the state of its environment and where a cumulative reward function is used to define the quality of any given action sequence. Within this class of problems we will consider specific settings, including settings in which the actions have deterministic or nondeterministic outcomes, and settings in which the agent



Goal: Learn to choose actions that maximize

$$r_0 + \gamma r_1 + \gamma^2 r_2 + \dots, \text{ where } 0 \leq \gamma < 1$$

FIGURE 13.1

An agent interacting with its environment. The agent exists in an environment described by some set of possible states S . It can perform any of a set of possible actions A . Each time it performs an action a_t in some state s_t the agent receives a real-valued reward r_t that indicates the immediate value of this state-action transition. This produces a sequence of states s_t , actions a_t , and immediate rewards r_t as shown in the figure. The agent's task is to learn a control policy, $\pi : S \rightarrow A$, that maximizes the expected sum of these rewards, with future rewards discounted exponentially by their delay.

has or does not have prior knowledge about the effects of its actions on the environment.

Note we have touched on the problem of learning to control sequential processes earlier in this book. In Section 11.4 we discussed explanation-based learning of rules to control search during problem solving. There the problem is for the agent to choose among alternative actions at each step in its search for some goal state. The techniques discussed here differ from those of Section 11.4, in that here we consider problems where the actions may have nondeterministic outcomes and where the learner lacks a domain theory that describes the outcomes of its actions. In Chapter 1 we discussed the problem of learning to choose actions while playing the game of checkers. There we sketched the design of a learning method very similar to those discussed in this chapter. In fact, one highly successful application of the reinforcement learning algorithms of this chapter is to a similar game-playing problem. Tesauro (1995) describes the TD-GAMMON program, which has used reinforcement learning to become a world-class backgammon player. This program, after training on 1.5 million self-generated games, is now considered nearly equal to the best human players in the world and has played competitively against top-ranked players in international backgammon tournaments.

The problem of learning a control policy to choose actions is similar in some respects to the function approximation problems discussed in other chapters. The target function to be learned in this case is a control policy, $\pi : S \rightarrow A$, that outputs an appropriate action a from the set A , given the current state s from the set S . However, this reinforcement learning problem differs from other function approximation tasks in several important respects.

- *Delayed reward.* The task of the agent is to learn a target function π that maps from the current state s to the optimal action $a = \pi(s)$. In earlier chapters we have always assumed that when learning some target function such as π , each training example would be a pair of the form $\langle s, \pi(s) \rangle$. In reinforcement learning, however, training information is not available in this form. Instead, the trainer provides only a sequence of immediate reward values as the agent executes its sequence of actions. The agent, therefore, faces the problem of *temporal credit assignment*: determining which of the actions in its sequence are to be credited with producing the eventual rewards.
- *Exploration.* In reinforcement learning, the agent influences the distribution of training examples by the action sequence it chooses. This raises the question of which experimentation strategy produces most effective learning. The learner faces a tradeoff in choosing whether to favor *exploration* of unknown states and actions (to gather new information), or *exploitation* of states and actions that it has already learned will yield high reward (to maximize its cumulative reward).
- *Partially observable states.* Although it is convenient to assume that the agent's sensors can perceive the entire state of the environment at each time step, in many practical situations sensors provide only partial information. For example, a robot with a forward-pointing camera cannot see what is

behind it. In such cases, it may be necessary for the agent to consider its previous observations together with its current sensor data when choosing actions, and the best policy may be one that chooses actions specifically to improve the observability of the environment.

- *Life-long learning.* Unlike isolated function approximation tasks, robot learning often requires that the robot learn several related tasks within the same environment, using the same sensors. For example, a mobile robot may need to learn how to dock on its battery charger, how to navigate through narrow corridors, and how to pick up output from laser printers. This setting raises the possibility of using previously obtained experience or knowledge to reduce sample complexity when learning new tasks.

13.2 THE LEARNING TASK

In this section we formulate the problem of learning sequential control strategies more precisely. Note there are many ways to do so. For example, we might assume the agent's actions are deterministic or that they are nondeterministic. We might assume that the agent can predict the next state that will result from each action, or that it cannot. We might assume that the agent is trained by an expert who shows it examples of optimal action sequences, or that it must train itself by performing actions of its own choice. Here we define one quite general formulation of the problem, based on Markov decision processes. This formulation of the problem follows the problem illustrated in Figure 13.1.

In a Markov decision process (MDP) the agent can perceive a set S of distinct states of its environment and has a set A of actions that it can perform. At each discrete time step t , the agent senses the current state s_t , chooses a current action a_t , and performs it. The environment responds by giving the agent a reward $r_t = r(s_t, a_t)$ and by producing the succeeding state $s_{t+1} = \delta(s_t, a_t)$. Here the functions δ and r are part of the environment and are not necessarily known to the agent. In an MDP, the functions $\delta(s_t, a_t)$ and $r(s_t, a_t)$ depend only on the current state and action, and not on earlier states or actions. In this chapter we consider only the case in which S and A are finite. In general, δ and r may be nondeterministic functions, but we begin by considering only the deterministic case.

The task of the agent is to learn a *policy*, $\pi : S \rightarrow A$, for selecting its next action a_t based on the current observed state s_t ; that is, $\pi(s_t) = a_t$. How shall we specify precisely which policy π we would like the agent to learn? One obvious approach is to require the policy that produces the greatest possible cumulative reward for the robot over time. To state this requirement more precisely, we define the cumulative value $V^\pi(s_t)$ achieved by following an arbitrary policy π from an arbitrary initial state s_t as follows:

$$\begin{aligned} V^\pi(s_t) &\equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots \\ &= \sum_{i=0}^{\infty} \gamma^i r_{t+i} \end{aligned} \tag{13.1}$$

where the sequence of rewards r_{t+i} is generated by beginning at state s_t and by repeatedly using the policy π to select actions as described above (i.e., $a_t = \pi(s_t)$, $a_{t+1} = \pi(s_{t+1})$, etc.). Here $0 \leq \gamma < 1$ is a constant that determines the relative value of delayed versus immediate rewards. In particular, rewards received i time steps into the future are discounted exponentially by a factor of γ^i . Note if we set $\gamma = 0$, only the immediate reward is considered. As we set γ closer to 1, future rewards are given greater emphasis relative to the immediate reward.

The quantity $V^\pi(s)$ defined by Equation (13.1) is often called the *discounted cumulative reward* achieved by policy π from initial state s . It is reasonable to discount future rewards relative to immediate rewards because, in many cases, we prefer to obtain the reward sooner rather than later. However, other definitions of total reward have also been explored. For example, *finite horizon reward*, $\sum_{i=0}^h r_{t+i}$, considers the undiscounted sum of rewards over a finite number h of steps. Another possibility is *average reward*, $\lim_{h \rightarrow \infty} \frac{1}{h} \sum_{i=0}^h r_{t+i}$, which considers the average reward per time step over the entire lifetime of the agent. In this chapter we restrict ourselves to considering discounted reward as defined by Equation (13.1). Mahadevan (1996) provides a discussion of reinforcement learning when the criterion to be optimized is average reward.

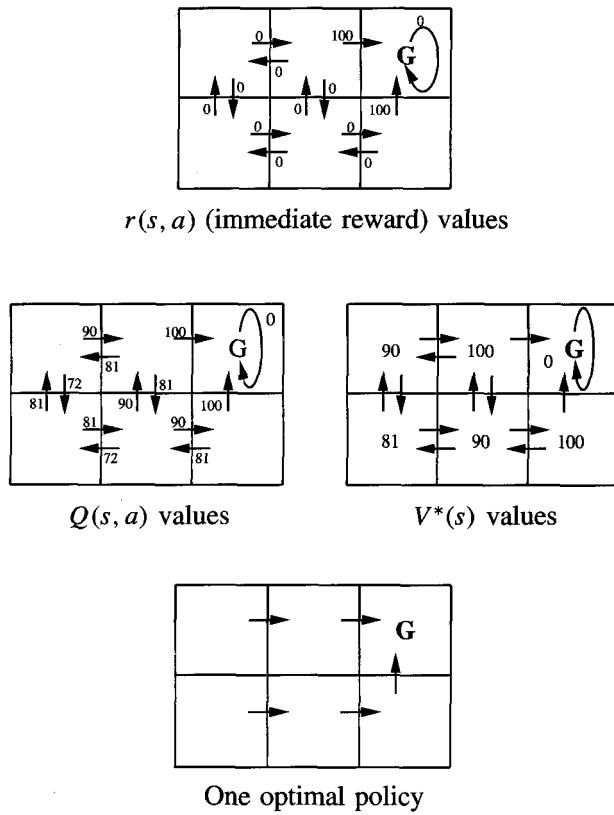
We are now in a position to state precisely the agent's learning task. We require that the agent learn a policy π that maximizes $V^\pi(s)$ for all states s . We will call such a policy an *optimal policy* and denote it by π^* .

$$\pi^* \equiv \underset{\pi}{\operatorname{argmax}} V^\pi(s), (\forall s) \quad (13.2)$$

To simplify notation, we will refer to the value function $V^{\pi^*}(s)$ of such an optimal policy as $V^*(s)$. $V^*(s)$ gives the maximum discounted cumulative reward that the agent can obtain starting from state s ; that is, the discounted cumulative reward obtained by following the optimal policy beginning at state s .

To illustrate these concepts, a simple grid-world environment is depicted in the topmost diagram of Figure 13.2. The six grid squares in this diagram represent six possible states, or locations, for the agent. Each arrow in the diagram represents a possible action the agent can take to move from one state to another. The number associated with each arrow represents the immediate reward $r(s, a)$ the agent receives if it executes the corresponding state-action transition. Note the immediate reward in this particular environment is defined to be zero for all state-action transitions except for those leading into the state labeled **G**. It is convenient to think of the state **G** as the goal state, because the only way the agent can receive reward, in this case, is by entering this state. Note in this particular environment, the only action available to the agent once it enters the state **G** is to remain in this state. For this reason, we call **G** an *absorbing state*.

Once the states, actions, and immediate rewards are defined, and once we choose a value for the discount factor γ , we can determine the optimal policy π^* and its value function $V^*(s)$. In this case, let us choose $\gamma = 0.9$. The diagram at the bottom of the figure shows one optimal policy for this setting (there are others as well). Like any policy, this policy specifies exactly one action that the

**FIGURE 13.2**

A simple deterministic world to illustrate the basic concepts of Q -learning. Each grid square represents a distinct state, each arrow a distinct action. The immediate reward function, $r(s, a)$ gives reward 100 for actions entering the goal state G , and zero otherwise. Values of $V^*(s)$ and $Q(s, a)$ follow from $r(s, a)$, and the discount factor $\gamma = 0.9$. An optimal policy, corresponding to actions with maximal Q values, is also shown.

agent will select in any given state. Not surprisingly, the optimal policy directs the agent along the shortest path toward the state G .

The diagram at the right of Figure 13.2 shows the values of V^* for each state. For example, consider the bottom right state in this diagram. The value of V^* for this state is 100 because the optimal policy in this state selects the “move up” action that receives immediate reward 100. Thereafter, the agent will remain in the absorbing state and receive no further rewards. Similarly, the value of V^* for the bottom center state is 90. This is because the optimal policy will move the agent from this state to the right (generating an immediate reward of zero), then upward (generating an immediate reward of 100). Thus, the discounted future reward from the bottom center state is

$$0 + \gamma 100 + \gamma^2 0 + \gamma^3 0 + \dots = 90$$

Recall that V^* is defined to be the sum of discounted future rewards over the infinite future. In this particular environment, once the agent reaches the absorbing state G its infinite future will consist of remaining in this state and receiving rewards of zero.

13.3 Q LEARNING

How can an agent learn an optimal policy π^* for an arbitrary environment? It is difficult to learn the function $\pi^* : S \rightarrow A$ directly, because the available training data does not provide training examples of the form $\langle s, a \rangle$. Instead, the only training information available to the learner is the sequence of immediate rewards $r(s_i, a_i)$ for $i = 0, 1, 2, \dots$. As we shall see, given this kind of training information it is easier to learn a numerical evaluation function defined over states and actions, then implement the optimal policy in terms of this evaluation function.

What evaluation function should the agent attempt to learn? One obvious choice is V^* . The agent should prefer state s_1 over state s_2 whenever $V^*(s_1) > V^*(s_2)$, because the cumulative future reward will be greater from s_1 . Of course the agent's policy must choose among actions, not among states. However, it can use V^* in certain settings to choose among actions as well. The optimal action in state s is the action a that maximizes the sum of the immediate reward $r(s, a)$ plus the value V^* of the immediate successor state, discounted by γ .

$$\pi^*(s) = \operatorname{argmax}_a [r(s, a) + \gamma V^*(\delta(s, a))] \quad (13.3)$$

(recall that $\delta(s, a)$ denotes the state resulting from applying action a to state s .) Thus, the agent can acquire the optimal policy by learning V^* , *provided it has perfect knowledge of the immediate reward function r and the state transition function δ* . When the agent knows the functions r and δ used by the environment to respond to its actions, it can then use Equation (13.3) to calculate the optimal action for any state s .

Unfortunately, learning V^* is a useful way to learn the optimal policy *only* when the agent has perfect knowledge of δ and r . This requires that it be able to perfectly predict the immediate result (i.e., the immediate reward and immediate successor) for every possible state-action transition. This assumption is comparable to the assumption of a perfect domain theory in explanation-based learning, discussed in Chapter 11. In many practical problems, such as robot control, it is impossible for the agent or its human programmer to predict in advance the exact outcome of applying an arbitrary action to an arbitrary state. Imagine, for example, the difficulty in describing δ for a robot arm shoveling dirt when the resulting state includes the positions of the dirt particles. In cases where either δ or r is unknown, learning V^* is unfortunately of no use for selecting optimal actions because the agent cannot evaluate Equation (13.3). What evaluation function should the agent use in this more general setting? The evaluation function Q , defined in the following section, provides one answer.

13.3.1 The Q Function

Let us define the evaluation function $Q(s, a)$ so that its value is the maximum discounted cumulative reward that can be achieved starting from state s and applying action a as the first action. In other words, the value of Q is the reward received immediately upon executing action a from state s , plus the value (discounted by γ) of following the optimal policy thereafter.

$$Q(s, a) \equiv r(s, a) + \gamma V^*(\delta(s, a)) \quad (13.4)$$

Note that $Q(s, a)$ is exactly the quantity that is maximized in Equation (13.3) in order to choose the optimal action a in state s . Therefore, we can rewrite Equation (13.3) in terms of $Q(s, a)$ as

$$\pi^*(s) = \operatorname{argmax}_a Q(s, a) \quad (13.5)$$

Why is this rewrite important? Because it shows that if the agent learns the Q function instead of the V^* function, it will be able to select optimal actions *even when it has no knowledge of the functions r and δ* . As Equation (13.5) makes clear, it need only consider each available action a in its current state s and choose the action that maximizes $Q(s, a)$.

It may at first seem surprising that one can choose globally optimal action sequences by reacting repeatedly to the local values of Q for the current state. This means the agent can choose the optimal action without ever conducting a lookahead search to explicitly consider what state results from the action. Part of the beauty of Q learning is that the evaluation function is defined to have precisely this property—the value of Q for the current state and action summarizes in a single number all the information needed to determine the discounted cumulative reward that will be gained in the future if action a is selected in state s .

To illustrate, Figure 13.2 shows the Q values for every state and action in the simple grid world. Notice that the Q value for each state-action transition equals the r value for this transition plus the V^* value for the resulting state discounted by γ . Note also that the optimal policy shown in the figure corresponds to selecting actions with maximal Q values.

13.3.2 An Algorithm for Learning Q

Learning the Q function corresponds to learning the optimal policy. How can Q be learned?

The key problem is finding a reliable way to estimate training values for Q , given only a sequence of immediate rewards r spread out over time. This can be accomplished through iterative approximation. To see how, notice the close relationship between Q and V^* ,

$$V^*(s) = \max_{a'} Q(s, a')$$

which allows rewriting Equation (13.4) as

$$Q(s, a) = r(s, a) + \gamma \max_{a'} Q(\delta(s, a), a') \quad (13.6)$$

This recursive definition of Q provides the basis for algorithms that iteratively approximate Q (Watkins 1989). To describe the algorithm, we will use the symbol \hat{Q} to refer to the learner's estimate, or hypothesis, of the actual Q function. In this algorithm the learner represents its hypothesis \hat{Q} by a large table with a separate entry for each state-action pair. The table entry for the pair (s, a) stores the value for $\hat{Q}(s, a)$ —the learner's current hypothesis about the actual but unknown value $Q(s, a)$. The table can be initially filled with random values (though it is easier to understand the algorithm if one assumes initial values of zero). The agent repeatedly observes its current state s , chooses some action a , executes this action, then observes the resulting reward $r = r(s, a)$ and the new state $s' = \delta(s, a)$. It then updates the table entry for $\hat{Q}(s, a)$ following each such transition, according to the rule:

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a') \quad (13.7)$$

Note this training rule uses the agent's current \hat{Q} values for the new state s' to refine its estimate of $\hat{Q}(s, a)$ for the previous state s . This training rule is motivated by Equation (13.6), although the training rule concerns the agent's approximation \hat{Q} , whereas Equation (13.6) applies to the actual Q function. Note although Equation (13.6) describes Q in terms of the functions $\delta(s, a)$ and $r(s, a)$, the agent does not need to know these general functions to apply the training rule of Equation (13.7). Instead it executes the action in its environment and then observes the resulting new state s' and reward r . Thus, it can be viewed as sampling these functions at the current values of s and a .

The above Q learning algorithm for deterministic Markov decision processes is described more precisely in Table 13.1. Using this algorithm the agent's estimate \hat{Q} converges in the limit to the actual Q function, provided the system can be modeled as a deterministic Markov decision process, the reward function r is

Q learning algorithm

For each s, a initialize the table entry $\hat{Q}(s, a)$ to zero.

Observe the current state s

Do forever:

- Select an action a and execute it
- Receive immediate reward r
- Observe the new state s'
- Update the table entry for $\hat{Q}(s, a)$ as follows:

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$$

- $s \leftarrow s'$

TABLE 13.1

Q learning algorithm, assuming deterministic rewards and actions. The discount factor γ may be any constant such that $0 \leq \gamma < 1$.

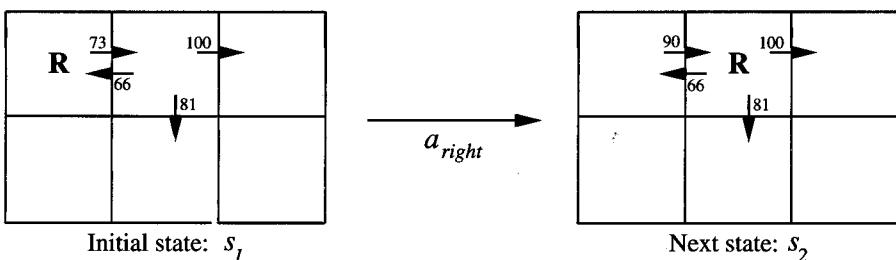
bounded, and actions are chosen so that every state-action pair is visited infinitely often.

13.3.3 An Illustrative Example

To illustrate the operation of the Q learning algorithm, consider a single action taken by an agent, and the corresponding refinement to \hat{Q} shown in Figure 13.3. In this example, the agent moves one cell to the right in its grid world and receives an immediate reward of zero for this transition. It then applies the training rule of Equation (13.7) to refine its estimate \hat{Q} for the state-action transition it just executed. According to the training rule, the new \hat{Q} estimate for this transition is the sum of the received reward (zero) and the highest \hat{Q} value associated with the resulting state (100), discounted by γ (.9).

Each time the agent moves forward from an old state to a new one, Q learning propagates \hat{Q} estimates *backward* from the new state to the old. At the same time, the immediate reward received by the agent for the transition is used to augment these propagated values of \hat{Q} .

Consider applying this algorithm to the grid world and reward function shown in Figure 13.2, for which the reward is zero everywhere, except when entering the goal state. Since this world contains an absorbing goal state, we will assume that training consists of a series of *episodes*. During each episode, the agent begins at some randomly chosen state and is allowed to execute actions until it reaches the absorbing goal state. When it does, the episode ends and



$$\begin{aligned}\hat{Q}(s_1, a_{right}) &\leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a') \\ &\leftarrow 0 + 0.9 \max\{66, 81, 100\} \\ &\leftarrow 90\end{aligned}$$

FIGURE 13.3

The update to \hat{Q} after executing a single action. The diagram on the left shows the initial state s_1 of the robot (R) and several relevant \hat{Q} values in its initial hypothesis. For example, the value $\hat{Q}(s_1, a_{right}) = 72.9$, where a_{right} refers to the action that moves R to its right. When the robot executes the action a_{right} , it receives immediate reward $r = 0$ and transitions to state s_2 . It then updates its estimate $\hat{Q}(s_1, a_{right})$ based on its \hat{Q} estimates for the new state s_2 . Here $\gamma = 0.9$.

the agent is transported to a new, randomly chosen, initial state for the next episode.

How will the values of \hat{Q} evolve as the Q learning algorithm is applied in this case? With all the \hat{Q} values initialized to zero, the agent will make no changes to any \hat{Q} table entry until it happens to reach the goal state and receive a nonzero reward. This will result in refining the \hat{Q} value for the single transition leading into the goal state. On the next episode, if the agent passes through this state adjacent to the goal state, its nonzero \hat{Q} value will allow refining the value for some transition two steps from the goal, and so on. Given a sufficient number of training episodes, the information will propagate from the transitions with nonzero reward back through the entire state-action space available to the agent, resulting eventually in a \hat{Q} table containing the Q values shown in Figure 13.2.

In the next section we prove that under certain assumptions the Q learning algorithm of Table 13.1 will converge to the correct Q function. First consider two general properties of this Q learning algorithm that hold for any deterministic MDP in which the rewards are non-negative, assuming we initialize all \hat{Q} values to zero. The first property is that under these conditions the \hat{Q} values never decrease during training. More formally, let $\hat{Q}_n(s, a)$ denote the learned $\hat{Q}(s, a)$ value after the n th iteration of the training procedure (i.e., after the n th state-action transition taken by the agent). Then

$$(\forall s, a, n) \quad \hat{Q}_{n+1}(s, a) \geq \hat{Q}_n(s, a)$$

A second general property that holds under these same conditions is that throughout the training process every \hat{Q} value will remain in the interval between zero and its true Q value.

$$(\forall s, a, n) \quad 0 \leq \hat{Q}_n(s, a) \leq Q(s, a)$$

13.3.4 Convergence

Will the algorithm of Table 13.1 converge toward a \hat{Q} equal to the true Q function? The answer is yes, under certain conditions. First, we must assume the system is a deterministic MDP. Second, we must assume the immediate reward values are bounded; that is, there exists some positive constant c such that for all states s and actions a , $|r(s, a)| < c$. Third, we assume the agent selects actions in such a fashion that it visits every possible state-action pair infinitely often. By this third condition we mean that if action a is a legal action from state s , then over time the agent must execute action a from state s repeatedly and with nonzero frequency as the length of its action sequence approaches infinity. Note these conditions are in some ways quite general and in others fairly restrictive. They describe a more general setting than illustrated by the example in the previous section, because they allow for environments with arbitrary positive or negative rewards, and for environments where any number of state-action transitions may produce nonzero rewards. The conditions are also restrictive in that they require the agent to visit every distinct state-action transition infinitely often. This is a very strong assumption in large (or continuous!) domains. We will discuss stronger

convergence results later. However, the result described in this section provides the basic intuition for understanding why Q learning works.

The key idea underlying the proof of convergence is that the table entry $\hat{Q}(s, a)$ with the largest error must have its error reduced by a factor of γ whenever it is updated. The reason is that its new value depends only in part on error-prone \hat{Q} estimates, with the remainder depending on the error-free observed immediate reward r .

Theorem 13.1. Convergence of Q learning for deterministic Markov decision processes. Consider a Q learning agent in a deterministic MDP with bounded rewards $(\forall s, a) |r(s, a)| \leq c$. The Q learning agent uses the training rule of Equation (13.7), initializes its table $\hat{Q}(s, a)$ to arbitrary finite values, and uses a discount factor γ such that $0 \leq \gamma < 1$. Let $\hat{Q}_n(s, a)$ denote the agent's hypothesis $\hat{Q}(s, a)$ following the n th update. If each state-action pair is visited infinitely often, then $\hat{Q}_n(s, a)$ converges to $Q(s, a)$ as $n \rightarrow \infty$, for all s, a .

Proof. Since each state-action transition occurs infinitely often, consider consecutive intervals during which each state-action transition occurs at least once. The proof consists of showing that the maximum error over all entries in the \hat{Q} table is reduced by at least a factor of γ during each such interval. \hat{Q}_n is the agent's table of estimated Q values after n updates. Let Δ_n be the maximum error in \hat{Q}_n ; that is

$$\Delta_n \equiv \max_{s, a} |\hat{Q}_n(s, a) - Q(s, a)|$$

Below we use s' to denote $\delta(s, a)$. Now for any table entry $\hat{Q}_n(s, a)$ that is updated on iteration $n + 1$, the magnitude of the error in the revised estimate $\hat{Q}_{n+1}(s, a)$ is

$$\begin{aligned} |\hat{Q}_{n+1}(s, a) - Q(s, a)| &= |(r + \gamma \max_{a'} \hat{Q}_n(s', a')) - (r + \gamma \max_{a'} Q(s', a'))| \\ &= \gamma |\max_{a'} \hat{Q}_n(s', a') - \max_{a'} Q(s', a')| \\ &\leq \gamma \max_{a'} |\hat{Q}_n(s', a') - Q(s', a')| \\ &\leq \gamma \max_{s'', a''} |\hat{Q}_n(s'', a'') - Q(s'', a')| \\ |\hat{Q}_{n+1}(s, a) - Q(s, a)| &\leq \gamma \Delta_n \end{aligned}$$

The third line above follows from the second line because for any two functions f_1 and f_2 the following inequality holds

$$|\max_a f_1(a) - \max_a f_2(a)| \leq \max_a |f_1(a) - f_2(a)|$$

In going from the third line to the fourth line above, note we introduce a new variable s'' over which the maximization is performed. This is legitimate because the maximum value will be at least as great when we allow this additional variable to vary. Note that by introducing this variable we obtain an expression that matches the definition of Δ_n .

Thus, the updated $\hat{Q}_{n+1}(s, a)$ for any s, a is at most γ times the maximum error in the \hat{Q}_n table, Δ_n . The largest error in the initial table, Δ_0 , is bounded because values of $\hat{Q}_0(s, a)$ and $Q(s, a)$ are bounded for all s, a . Now after the first interval

during which each s, a is visited, the largest error in the table will be at most $\gamma \Delta_0$. After k such intervals, the error will be at most $\gamma^k \Delta_0$. Since each state is visited infinitely often, the number of such intervals is infinite, and $\Delta_n \rightarrow 0$ as $n \rightarrow \infty$. This proves the theorem. \square

13.3.5 Experimentation Strategies

Notice the algorithm of Table 13.1 does not specify how actions are chosen by the agent. One obvious strategy would be for the agent in state s to select the action a that maximizes $\hat{Q}(s, a)$, thereby exploiting its current approximation \hat{Q} . However, with this strategy the agent runs the risk that it will overcommit to actions that are found during early training to have high \hat{Q} values, while failing to explore other actions that have even higher values. In fact, the convergence theorem above requires that each state-action transition occur infinitely often. This will clearly not occur if the agent always selects actions that maximize its current $\hat{Q}(s, a)$. For this reason, it is common in Q learning to use a probabilistic approach to selecting actions. Actions with higher \hat{Q} values are assigned higher probabilities, but every action is assigned a nonzero probability. One way to assign such probabilities is

$$P(a_i|s) = \frac{k \hat{Q}(s, a_i)}{\sum_j k \hat{Q}(s, a_j)}$$

where $P(a_i|s)$ is the probability of selecting action a_i , given that the agent is in state s , and where $k > 0$ is a constant that determines how strongly the selection favors actions with high \hat{Q} values. Larger values of k will assign higher probabilities to actions with above average \hat{Q} , causing the agent to *exploit* what it has learned and seek actions it believes will maximize its reward. In contrast, small values of k will allow higher probabilities for other actions, leading the agent to *explore* actions that do not currently have high \hat{Q} values. In some cases, k is varied with the number of iterations so that the agent favors exploration during early stages of learning, then gradually shifts toward a strategy of exploitation.

13.3.6 Updating Sequence

One important implication of the above convergence theorem is that Q learning need not train on optimal action sequences in order to converge to the optimal policy. In fact, it can learn the Q function (and hence the optimal policy) while training from actions chosen completely at random at each step, as long as the resulting training sequence visits every state-action transition infinitely often. This fact suggests changing the sequence of training example transitions in order to improve training efficiency without endangering final convergence. To illustrate, consider again learning in an MDP with a single absorbing goal state, such as the one in Figure 13.1. Assume as before that we train the agent with a sequence of episodes. For each episode, the agent is placed in a random initial state and is allowed to perform actions and to update its \hat{Q} table until it reaches the absorbing goal state. A new training episode is then begun by removing the agent from the

goal state and placing it at a new random initial state. As noted earlier, if we begin with all \hat{Q} values initialized to zero, then after the first full episode only one entry in the agent's \hat{Q} table will have been changed: the entry corresponding to the final transition into the goal state. Note that if the agent happens to follow the same sequence of actions from the same random initial state in its second full episode, then a second table entry would be made nonzero, and so on. If we run repeated identical episodes in this fashion, the frontier of nonzero \hat{Q} values will creep backward from the goal state at the rate of one new state-action transition per episode. Now consider training on these same state-action transitions, but in reverse chronological order for each episode. That is, we apply the same update rule from Equation (13.7) for each transition considered, but perform these updates in reverse order. In this case, after the first full episode the agent will have updated its \hat{Q} estimate for every transition along the path it took to the goal. This training process will clearly converge in fewer iterations, although it requires that the agent use more memory to store the entire episode before beginning the training for that episode.

A second strategy for improving the rate of convergence is to store past state-action transitions, along with the immediate reward that was received, and retrain on them periodically. Although at first it might seem a waste of effort to retrain on the same transition, recall that the updated $\hat{Q}(s, a)$ value is determined by the values $\hat{Q}(s', a)$ of the successor state $s' = \delta(s, a)$. Therefore, if subsequent training changes one of the $\hat{Q}(s', a)$ values, then retraining on the transition $\langle s, a \rangle$ may result in an altered value for $\hat{Q}(s, a)$. In general, the degree to which we wish to replay old transitions versus obtain new ones from the environment depends on the relative costs of these two operations in the specific problem domain. For example, in a robot domain with navigation actions that might take several seconds to perform, the delay in collecting a new state-action transition from the external world might be several orders of magnitude more costly than internally replaying a previously observed transition. This difference can be very significant given that Q learning can often require thousands of training iterations to converge.

Note throughout the above discussion we have kept our assumption that the agent does not know the state-transition function $\delta(s, a)$ used by the environment to create the successor state $s' = \delta(s, a)$, or the function $r(s, a)$ used to generate rewards. If it does know these two functions, then many more efficient methods are possible. For example, if performing external actions is expensive the agent may simply ignore the environment and instead simulate it internally, efficiently generating simulated actions and assigning the appropriate simulated rewards. Sutton (1991) describes the DYNA architecture that performs a number of simulated actions after each step executed in the external world. Moore and Atkeson (1993) describe an approach called *prioritized sweeping* that selects promising states to update next, focusing on predecessor states when the current state is found to have a large update. Peng and Williams (1994) describe a similar approach. A large number of efficient algorithms from the field of dynamic programming can be applied when the functions δ and r are known. Kaelbling et al. (1996) survey a number of these.

13.4 NONDETERMINISTIC REWARDS AND ACTIONS

Above we considered Q learning in deterministic environments. Here we consider the nondeterministic case, in which the reward function $r(s, a)$ and action transition function $\delta(s, a)$ may have probabilistic outcomes. For example, in Tesauro's (1995) backgammon playing program, action outcomes are inherently probabilistic because each move involves a roll of the dice. Similarly, in robot problems with noisy sensors and effectors it is often appropriate to model actions and rewards as nondeterministic. In such cases, the functions $\delta(s, a)$ and $r(s, a)$ can be viewed as first producing a probability distribution over outcomes based on s and a , and then drawing an outcome at random according to this distribution. When these probability distributions depend solely on s and a (e.g., they do not depend on previous states or actions), then we call the system a nondeterministic Markov decision process.

In this section we extend the Q learning algorithm for the deterministic case to handle nondeterministic MDPs. To accomplish this, we retrace the line of argument that led to the algorithm for the deterministic case, revising it where needed.

In the nondeterministic case we must first restate the objective of the learner to take into account the fact that outcomes of actions are no longer deterministic. The obvious generalization is to redefine the value V^π of a policy π to be the *expected value* (over these nondeterministic outcomes) of the discounted cumulative reward received by applying this policy

$$V^\pi(s_t) \equiv E \left[\sum_{i=0}^{\infty} \gamma^i r_{t+i} \right]$$

where, as before, the sequence of rewards r_{t+i} is generated by following policy π beginning at state s . Note this is a generalization of Equation (13.1), which covered the deterministic case.

As before, we define the optimal policy π^* to be the policy π that maximizes $V^\pi(s)$ for all states s . Next we generalize our earlier definition of Q from Equation (13.4), again by taking its expected value.

$$\begin{aligned} Q(s, a) &\equiv E[r(s, a) + \gamma V^*(\delta(s, a))] \\ &= E[r(s, a)] + \gamma E[V^*(\delta(s, a))] \\ &= E[r(s, a)] + \gamma \sum_{s'} P(s'|s, a) V^*(s') \end{aligned} \quad (13.8)$$

where $P(s'|s, a)$ is the probability that taking action a in state s will produce the next state s' . Note we have used $P(s'|s, a)$ here to rewrite the expected value of $V^*(\delta(s, a))$ in terms of the probabilities associated with the possible outcomes of the probabilistic δ .

As before we can re-express Q recursively

$$Q(s, a) = E[r(s, a)] + \gamma \sum_{s'} P(s'|s, a) \max_{a'} Q(s', a') \quad (13.9)$$

which is the generalization of the earlier Equation (13.6). To summarize, we have simply redefined $Q(s, a)$ in the nondeterministic case to be the expected value of its previously defined quantity for the deterministic case.

Now that we have generalized the definition of Q to accommodate the nondeterministic environment functions r and δ , a new training rule is needed. Our earlier training rule derived for the deterministic case (Equation 13.7) fails to converge in this nondeterministic setting. Consider, for example, a nondeterministic reward function $r(s, a)$ that produces different rewards each time the transition (s, a) is repeated. In this case, the training rule will repeatedly alter the values of $\hat{Q}(s, a)$, even if we initialize the \hat{Q} table values to the correct Q function. In brief, this training rule does not converge. This difficulty can be overcome by modifying the training rule so that it takes a decaying weighted average of the current \hat{Q} value and the revised estimate. Writing \hat{Q}_n to denote the agent's estimate on the n th iteration of the algorithm, the following revised training rule is sufficient to assure convergence of \hat{Q} to Q :

$$\hat{Q}_n(s, a) \leftarrow (1 - \alpha_n)\hat{Q}_{n-1}(s, a) + \alpha_n[r + \gamma \max_{a'} \hat{Q}_{n-1}(s', a')] \quad (13.10)$$

where

$$\alpha_n = \frac{1}{1 + \text{visits}_n(s, a)} \quad (13.11)$$

where s and a here are the state and action updated during the n th iteration, and where $\text{visits}_n(s, a)$ is the total number of times this state-action pair has been visited up to and including the n th iteration.

The key idea in this revised rule is that revisions to \hat{Q} are made more gradually than in the deterministic case. Notice if we were to set α_n to 1 in Equation (13.10) we would have exactly the training rule for the deterministic case. With smaller values of α , this term is now averaged in with the current $\hat{Q}(s, a)$ to produce the new updated value. Notice that the value of α_n in Equation (13.11) decreases as n increases, so that updates become smaller as training progresses. By reducing α at an appropriate rate during training, we can achieve convergence to the correct Q function. The choice of α_n given above is one of many that satisfy the conditions for convergence, according to the following theorem due to Watkins and Dayan (1992).

Theorem 13.2. Convergence of Q learning for nondeterministic Markov decision processes. Consider a Q learning agent in a nondeterministic MDP with bounded rewards $(\forall s, a)|r(s, a)| \leq c$. The Q learning agent uses the training rule of Equation (13.10), initializes its table $\hat{Q}(s, a)$ to arbitrary finite values, and uses a discount factor γ such that $0 \leq \gamma < 1$. Let $n(i, s, a)$ be the iteration corresponding to the i th time that action a is applied to state s . If each state-action pair is visited infinitely often, $0 \leq \alpha_n < 1$, and

$$\sum_{i=1}^{\infty} \alpha_{n(i, s, a)} = \infty, \quad \sum_{i=1}^{\infty} [\alpha_{n(i, s, a)}]^2 < \infty$$

then for all s and a , $\hat{Q}_n(s, a) \rightarrow Q(s, a)$ as $n \rightarrow \infty$, with probability 1.

While Q learning and related reinforcement learning algorithms can be proven to converge under certain conditions, in practice systems that use Q learning often require many thousands of training iterations to converge. For example, Tesauro's TD-GAMMON discussed earlier trained for 1.5 million backgammon games, each of which contained tens of state-action transitions.

13.5 TEMPORAL DIFFERENCE LEARNING

The Q learning algorithm learns by iteratively reducing the discrepancy between Q value estimates for adjacent states. In this sense, Q learning is a special case of a general class of *temporal difference* algorithms that learn by reducing discrepancies between estimates made by the agent at different times. Whereas the training rule of Equation (13.10) reduces the difference between the estimated \hat{Q} values of a state and its immediate successor, we could just as well design an algorithm that reduces discrepancies between this state and more distant descendants or ancestors.

To explore this issue further, recall that our Q learning training rule calculates a training value for $\hat{Q}(s_t, a_t)$ in terms of the values for $\hat{Q}(s_{t+1}, a_{t+1})$ where s_{t+1} is the result of applying action a_t to the state s_t . Let $Q^{(1)}(s_t, a_t)$ denote the training value calculated by this one-step lookahead

$$Q^{(1)}(s_t, a_t) \equiv r_t + \gamma \max_a \hat{Q}(s_{t+1}, a)$$

One alternative way to compute a training value for $Q(s_t, a_t)$ is to base it on the observed rewards for two steps

$$Q^{(2)}(s_t, a_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 \max_a \hat{Q}(s_{t+2}, a)$$

or, in general, for n steps

$$Q^{(n)}(s_t, a_t) \equiv r_t + \gamma r_{t+1} + \cdots + \gamma^{(n-1)} r_{t+n-1} + \gamma^n \max_a \hat{Q}(s_{t+n}, a)$$

Sutton (1988) introduces a general method for blending these alternative training estimates, called $TD(\lambda)$. The idea is to use a constant $0 \leq \lambda \leq 1$ to combine the estimates obtained from various lookahead distances in the following fashion

$$Q^\lambda(s_t, a_t) \equiv (1 - \lambda) [Q^{(1)}(s_t, a_t) + \lambda Q^{(2)}(s_t, a_t) + \lambda^2 Q^{(3)}(s_t, a_t) + \cdots]$$

An equivalent recursive definition for Q^λ is

$$\begin{aligned} Q^\lambda(s_t, a_t) = & r_t + \gamma [(1 - \lambda) \max_a \hat{Q}(s_t, a_t) \\ & + \lambda Q^\lambda(s_{t+1}, a_{t+1})] \end{aligned}$$

Note if we choose $\lambda = 0$ we have our original training estimate $Q^{(1)}$, which considers only one-step discrepancies in the \hat{Q} estimates. As λ is increased, the algorithm places increasing emphasis on discrepancies based on more distant lookaheads. At the extreme value $\lambda = 1$, only the observed r_{t+i} values are considered,

with no contribution from the current \hat{Q} estimate. Note when $\hat{Q} = Q$, the training values given by Q^λ will be identical for all values of λ such that $0 \leq \lambda \leq 1$.

The motivation for the TD(λ) method is that in some settings training will be more efficient if more distant lookaheads are considered. For example, when the agent follows an optimal policy for choosing actions, then Q^λ with $\lambda = 1$ will provide a perfect estimate for the true Q value, regardless of any inaccuracies in \hat{Q} . On the other hand, if action sequences are chosen suboptimally, then the r_{t+i} observed far into the future can be misleading.

Peng and Williams (1994) provide a further discussion and experimental results showing the superior performance of Q^λ in one problem domain. Dayan (1992) shows that under certain assumptions a similar TD(λ) approach applied to learning the V^* function converges correctly for any λ such that $0 \leq \lambda \leq 1$. Tesauro (1995) uses a TD(λ) approach in his TD-GAMMON program for playing backgammon.

13.6 GENERALIZING FROM EXAMPLES

Perhaps the most constraining assumption in our treatment of Q learning up to this point is that the target function is represented as an explicit lookup table, with a distinct table entry for every distinct input value (i.e., state-action pair). Thus, the algorithms we discussed perform a kind of rote learning and make no attempt to estimate the Q value for unseen state-action pairs by generalizing from those that have been seen. This rote learning assumption is reflected in the convergence proof, which proves convergence only if every possible state-action pair is visited (infinitely often!). This is clearly an unrealistic assumption in large or infinite spaces, or when the cost of executing actions is high. As a result, more practical systems often combine function approximation methods discussed in other chapters with the Q learning training rules described here.

It is easy to incorporate function approximation algorithms such as BACKPROPAGATION into the Q learning algorithm, by substituting a neural network for the lookup table and using each $\hat{Q}(s, a)$ update as a training example. For example, we could encode the state s and action a as network inputs and train the network to output the target values of \hat{Q} given by the training rules of Equations (13.7) and (13.10). An alternative that has sometimes been found to be more successful in practice is to train a separate network for each action, using the state as input and \hat{Q} as output. Another common alternative is to train one network with the state as input, but with one \hat{Q} output for each action. Recall that in Chapter 1, we discussed approximating an evaluation function over checkerboard states using a linear function and the LMS algorithm.

In practice, a number of successful reinforcement learning systems have been developed by incorporating such function approximation algorithms in place of the lookup table. Tesauro's successful TD-GAMMON program for playing backgammon used a neural network and the BACKPROPAGATION algorithm together with a TD(λ) training rule. Zhang and Dietterich (1996) use a similar combination of BACKPROPAGATION and TD(λ) for job-shop scheduling tasks. Crites and Barto (1996) describe

a neural network reinforcement learning approach for an elevator scheduling task. Thrun (1996) reports a neural network based approach to Q learning to learn basic control procedures for a mobile robot with sonar and camera sensors. Mahadevan and Connell (1991) describe a Q learning approach based on clustering states, applied to a simple mobile robot control problem.

Despite the success of these systems, for other tasks reinforcement learning fails to converge once a generalizing function approximator is introduced. Examples of such problematic tasks are given by Boyan and Moore (1995), Baird (1995), and Gordon (1995). Note the convergence theorems discussed earlier in this chapter apply only when \hat{Q} is represented by an explicit table. To see the difficulty, consider using a neural network rather than an explicit table to represent \hat{Q} . Note if the learner updates the network to better fit the training Q value for a particular transition (s_i, a_i) , the altered network weights may also change the \hat{Q} estimates for arbitrary other transitions. Because these weight changes may increase the error in \hat{Q} estimates for these other transitions, the argument proving the original theorem no longer holds. Theoretical analyses of reinforcement learning with generalizing function approximators are given by Gordon (1995) and Tsitsiklis (1994). Baird (1995) proposes gradient-based methods that circumvent this difficulty by directly minimizing the sum of squared discrepancies in estimates between adjacent states (also called Bellman residual errors).

13.7 RELATIONSHIP TO DYNAMIC PROGRAMMING

Reinforcement learning methods such as Q learning are closely related to a long line of research on dynamic programming approaches to solving Markov decision processes. This earlier work has typically assumed that the agent possesses perfect knowledge of the functions $\delta(s, a)$ and $r(s, a)$ that define the agent's environment. Therefore, it has primarily addressed the question of how to compute the optimal policy using the least computational effort, assuming the environment could be perfectly simulated and no direct interaction was required. The novel aspect of Q learning is that it assumes the agent does *not* have knowledge of $\delta(s, a)$ and $r(s, a)$, and that instead of moving about in an internal mental model of the state space, it must move about the real world and observe the consequences. In this latter case our primary concern is usually the number of real-world actions that the agent must perform to converge to an acceptable policy, rather than the number of computational cycles it must expend. The reason is that in many practical domains such as manufacturing problems, the costs in time and in dollars of performing actions in the external world dominate the computational costs. Systems that learn by moving about the real environment and observing the results are typically called *online* systems, whereas those that learn solely by simulating actions within an internal model are called *offline* systems.

The close correspondence between these earlier approaches and the reinforcement learning problems discussed here is apparent by considering Bellman's equation, which forms the foundation for many dynamic programming approaches

to solving MDPs. Bellman's equation is

$$(\forall s \in S) V^*(s) = E[r(s, \pi(s)) + \gamma V^*(\delta(s, \pi(s)))]$$

Note the very close relationship between Bellman's equation and our earlier definition of an optimal policy in Equation (13.2). Bellman (1957) showed that the optimal policy π^* satisfies the above equation and that any policy π satisfying this equation is an optimal policy. Early work on dynamic programming includes the Bellman-Ford shortest path algorithm (Bellman 1958; Ford and Fulkerson 1962), which learns paths through a graph by repeatedly updating the estimated distance to the goal for each graph node, based on the distances for its neighbors. In this algorithm the assumption that graph edges and the goal node are known is equivalent to our assumption that $\delta(s, a)$ and $r(s, a)$ are known. Barto et al. (1995) discuss the close relationship between reinforcement learning and dynamic programming.

13.8 SUMMARY AND FURTHER READING

The key points discussed in this chapter include:

- Reinforcement learning addresses the problem of learning control strategies for autonomous agents. It assumes that training information is available in the form of a real-valued reward signal given for each state-action transition. The goal of the agent is to learn an action policy that maximizes the total reward it will receive from any starting state.
- The reinforcement learning algorithms addressed in this chapter fit a problem setting known as a Markov decision process. In Markov decision processes, the outcome of applying any action to any state depends only on this action and state (and not on preceding actions, or states). Markov decision processes cover a wide range of problems including many robot control, factory automation, and scheduling problems.
- Q learning is one form of reinforcement learning in which the agent learns an evaluation function over states and actions. In particular, the evaluation function $Q(s, a)$ is defined as the maximum expected, discounted, cumulative reward the agent can achieve by applying action a to state s . The Q learning algorithm has the advantage that it can be employed even when the learner has no prior knowledge of how its actions affect its environment.
- Q learning can be proven to converge to the correct Q function under certain assumptions, when the learner's hypothesis $\hat{Q}(s, a)$ is represented by a lookup table with a distinct entry for each (s, a) pair. It can be shown to converge in both deterministic and nondeterministic MDPs. In practice, Q learning can require many thousands of training iterations to converge in even modest-sized problems.
- Q learning is a member of a more general class of algorithms, called temporal difference algorithms. In general, temporal difference algorithms learn

by iteratively reducing the discrepancies between the estimates produced by the agent at different times.

- Reinforcement learning is closely related to dynamic programming approaches to Markov decision processes. The key difference is that historically these dynamic programming approaches have assumed that the agent possesses knowledge of the state transition function $\delta(s, a)$ and reward function $r(s, a)$. In contrast, reinforcement learning algorithms such as Q learning typically assume the learner lacks such knowledge.

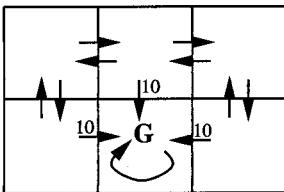
The common theme that underlies much of the work on reinforcement learning is to iteratively reduce the discrepancy between evaluations of successive states. Some of the earliest work on such methods is due to Samuel (1959). His checkers learning program attempted to learn an evaluation function for checkers by using evaluations of later states to generate training values for earlier states. Around the same time, the Bellman-Ford, single-destination, shortest-path algorithm was developed (Bellman 1958; Ford and Fulkerson 1962), which propagated distance-to-goal values from nodes to their neighbors. Research on optimal control led to the solution of Markov decision processes using similar methods (Bellman 1961; Blackwell 1965). Holland's (1986) bucket brigade method for learning classifier systems used a similar method for propagating credit in the face of delayed rewards. Barto et al. (1983) discussed an approach to temporal credit assignment that led to Sutton's paper (1988) defining the $TD(\lambda)$ method and proving its convergence for $\lambda = 0$. Dayan (1992) extended this result to arbitrary values of λ . Watkins (1989) introduced Q learning to acquire optimal policies when the reward and action transition functions are unknown. Convergence proofs are known for several variations on these methods. In addition to the convergence proofs presented in this chapter see, for example, (Baird 1995; Bertsekas 1987; Tsitsiklis 1994, Singh and Sutton 1996).

Reinforcement learning remains an active research area. McCallum (1995) and Littman (1996), for example, discuss the extension of reinforcement learning to settings with hidden state variables that violate the Markov assumption. Much current research seeks to scale up these methods to larger, more practical problems. For example, Maclin and Shavlik (1996) describe an approach in which a reinforcement learning agent can accept imperfect advice from a trainer, based on an extension to the KBANN algorithm (Chapter 12). Lin (1992) examines the role of teaching by providing suggested action sequences. Methods for scaling up by employing a hierarchy of actions are suggested by Singh (1993) and Lin (1993). Dietterich and Flann (1995) explore the integration of explanation-based methods with reinforcement learning, and Mitchell and Thrun (1993) describe the application of the EBNN algorithm (Chapter 12) to Q learning. Ring (1994) explores continual learning by the agent over multiple tasks.

Recent surveys of reinforcement learning are given by Kaelbling et al. (1996); Barto (1992); Barto et al. (1995); Dean et al. (1993).

EXERCISES

- 13.1.** Give a second optimal policy for the problem illustrated in Figure 13.2.
- 13.2.** Consider the deterministic grid world shown below with the absorbing goal-state G. Here the immediate rewards are 10 for the labeled transitions and 0 for all unlabeled transitions.
- Give the V^* value for every state in this grid world. Give the $Q(s, a)$ value for every transition. Finally, show an optimal policy. Use $\gamma = 0.8$.
 - Suggest a change to the reward function $r(s, a)$ that alters the $Q(s, a)$ values, but does not alter the optimal policy. Suggest a change to $r(s, a)$ that alters $Q(s, a)$ but does not alter $V^*(s, a)$.
 - Now consider applying the Q learning algorithm to this grid world, assuming the table of \hat{Q} values is initialized to zero. Assume the agent begins in the bottom left grid square and then travels clockwise around the perimeter of the grid until it reaches the absorbing goal state, completing the first training episode. Describe which \hat{Q} values are modified as a result of this episode, and give their revised values. Answer the question again assuming the agent now performs a second identical episode. Answer it again for a third episode.



- 13.3.** Consider playing Tic-Tac-Toe against an opponent who plays randomly. In particular, assume the opponent chooses with uniform probability any open space, unless there is a forced move (in which case it makes the obvious correct move).
- Formulate the problem of learning an optimal Tic-Tac-Toe strategy in this case as a Q -learning task. What are the states, transitions, and rewards in this non-deterministic Markov decision process?
 - Will your program succeed if the opponent plays optimally rather than randomly?
- 13.4.** Note in many MDPs it is possible to find two policies π_1 and π_2 such that π_1 outperforms π_2 if the agent begins in some state s_1 , but π_2 outperforms π_1 if it begins in some other state s_2 . Put another way, $V^{\pi_1}(s_1) > V^{\pi_2}(s_1)$, but $V^{\pi_2}(s_2) > V^{\pi_1}(s_2)$. Explain why there will always exist a single policy that maximizes $V^\pi(s)$ for *every* initial state s (i.e., an optimal policy π^*). In other words, explain why an MDP always allows a policy π^* such that $(\forall \pi, s) V^{\pi^*}(s) \geq V^\pi(s)$.

REFERENCES

- Baird, L. (1995). Residual algorithms: Reinforcement learning with function approximation. *Proceedings of the Twelfth International Conference on Machine Learning* (pp. 30–37). San Francisco: Morgan Kaufmann.

- Barto, A. (1992). Reinforcement learning and adaptive critic methods. In D. White & S. Sofge (Eds.), *Handbook of intelligent control: Neural, fuzzy, and adaptive approaches* (pp. 469–491). New York: Van Nostrand Reinhold.
- Barto, A., Bradtko, S., & Singh, S. (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence*, Special volume: Computational research on interaction and agency, 72(1), 81–138.
- Barto, A., Sutton, R., & Anderson, C. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13(5), 834–846.
- Bellman, R. E. (1957). *Dynamic Programming*. Princeton, NJ: Princeton University Press.
- Bellman, R. (1958). On a routing problem. *Quarterly of Applied Mathematics*, 16(1), 87–90.
- Bellman, R. (1961). *Adaptive control processes*. Princeton, NJ: Princeton University Press.
- Berenji, R. (1992). Learning and tuning fuzzy controllers through reinforcements. *IEEE Transactions on Neural Networks*, 3(5), 724–740.
- Bertsekas, D. (1987). *Dynamic programming: Deterministic and stochastic models*. Englewood Cliffs, NJ: Prentice Hall.
- Blackwell, D. (1965). Discounted dynamic programming. *Annals of Mathematical Statistics*, 36, 226–235.
- Boyan, J., & Moore, A. (1995). Generalization in reinforcement learning: Safely approximating the value function. In G. Tesauro, D. Touretzky, & T. Leen (Eds.), *Advances in Neural Information Processing Systems 7*. Cambridge, MA: MIT Press.
- Crites, R., & Barto, A. (1996). Improving elevator performance using reinforcement learning. In D. S. Touretzky, M. C. Mozer, & M. C. Hasselmo (Eds.), *Advances in Neural Information Processing Systems*, 8.
- Dayan, P. (1992). The convergence of TD(λ) for general λ . *Machine Learning*, 8, 341–362.
- Dean, T., Basye, K., & Shewchuk, J. (1993). Reinforcement learning for planning and control. In S. Minton (Ed.), *Machine Learning Methods for Planning* (pp. 67–92). San Francisco: Morgan Kaufmann.
- Dieterich, T. G., & Flann, N. S. (1995). Explanation-based learning and reinforcement learning: A unified view. *Proceedings of the 12th International Conference on Machine Learning* (pp. 176–184). San Francisco: Morgan Kaufmann.
- Ford, L., & Fulkerson, D. (1962). *Flows in networks*. Princeton, NJ: Princeton University Press.
- Gordon, G. (1995). Stable function approximation in dynamic programming. *Proceedings of the Twelfth International Conference on Machine Learning* (pp. 261–268). San Francisco: Morgan Kaufmann.
- Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of AI Research*, 4, 237–285. Online journal at <http://www.cs.washington.edu/research/jair/-home.html>.
- Holland, J. H. (1986). Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In Michalski, Carbonell, & Mitchell (Eds.), *Machine learning: An artificial intelligence approach* (Vol. 2, pp. 593–623). San Francisco: Morgan Kaufmann.
- Laird, J. E., & Rosenbloom, P. S. (1990). Integrating execution, planning, and learning in SOAR for external environments. *Proceedings of the Eighth National Conference on Artificial Intelligence* (pp. 1022–1029). Menlo Park, CA: AAAI Press.
- Lin, L. J. (1992). Self-improving reactive agents based on reinforcement learning, planning, and teaching. *Machine Learning*, 8, 293–321.
- Lin, L. J. (1993). Hierarchical learning of robot skills by reinforcement. *Proceedings of the International Conference on Neural Networks*.
- Littman, M. (1996). *Algorithms for sequential decision making* (Ph.D. dissertation and Technical Report CS-96-09). Brown University, Department of Computer Science, Providence, RI.
- Maclin, R., & Shavlik, J. W. (1996). Creating advice-taking reinforcement learners. *Machine Learning*, 22, 251–281.

- Mahadevan, S. (1996). Average reward reinforcement learning: Foundations, algorithms, and empirical results. *Machine Learning*, 22(1), 159–195.
- Mahadevan, S., & Connell, J. (1991). Automatic programming of behavior-based robots using reinforcement learning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*. San Francisco: Morgan Kaufmann.
- McCallum, A. (1995). *Reinforcement learning with selective perception and hidden state* (Ph.D. dissertation). Department of Computer Science, University of Rochester, Rochester, NY.
- Mitchell, T. M., & Thrun, S. B. (1993). Explanation-based neural network learning for robot control. In C. Giles, S. Hanson, & J. Cowan (Eds.), *Advances in Neural Information Processing Systems 5* (pp. 287–294). San Francisco: Morgan-Kaufmann.
- Moore, A., & Atkeson C. (1993). Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning*, 13, 103.
- Peng, J., & Williams, R. (1994). Incremental multi-step *Q*-learning. *Proceedings of the Eleventh International Conference on Machine Learning* (pp. 226–232). San Francisco: Morgan Kaufmann.
- Ring, M. (1994). *Continual learning in reinforcement environments* (Ph.D. dissertation). Computer Science Department, University of Texas at Austin, Austin, TX.
- Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3, 211–229.
- Singh, S. (1992). Reinforcement learning with a hierarchy of abstract models. *Proceedings of the Tenth National Conference on Artificial Intelligence* (pp. 202–207). San Jose, CA: AAAI Press.
- Singh, S. (1993). *Learning to solve markovian decision processes* (Ph.D. dissertation). Also CMPSCI Technical Report 93-77, Department of Computer Science, University of Massachusetts at Amherst.
- Singh, S., & Sutton, R. (1996). Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22, 123.
- Sutton, R. (1988). Learning to predict by the methods of temporal differences. *Machine learning*, 3, 9–44.
- Sutton R. (1991). Planning by incremental dynamic programming. *Proceedings of the Eighth International Conference on Machine Learning* (pp. 353–357). San Francisco: Morgan Kaufmann.
- Tesauro, G. (1995). Temporal difference learning and TD-GAMMON. *Communications of the ACM*, 38(3), 58–68.
- Thrun, S. (1992). The role of exploration in learning control. In D. White & D. Sofge (Eds.), *Handbook of intelligent control: Neural, fuzzy, and adaptive approaches* (pp. 527–559). New York: Van Nostrand Reinhold.
- Thrun, S. (1996). Explanation-based neural network learning: A lifelong learning approach. Boston: Kluwer Academic Publishers.
- Tsitsiklis, J. (1994). Asynchronous stochastic approximation and *Q*-learning. *Machine Learning*, 16(3), 185–202.
- Watkins, C. (1989). *Learning from delayed rewards* (Ph.D. dissertation). King's College, Cambridge, England.
- Watkins, C., & Dayan, P. (1992). *Q*-learning. *Machine Learning*, 8, 279–292.
- Zhang, W., & Dietterich, T. G. (1996). High-performance job-shop scheduling with a time-delay TD(λ) network. In D. S. Touretzky, M. C. Mozer, & M. E. Hasselmo (Eds.), *Advances in neural information processing systems*, 8, 1024–1030.

APPENDIX

NOTATION

Below is a summary of notation used in this book.

(a, b]: Brackets of the form [,], (, and) are used to represent intervals, where square brackets represent intervals including the boundary and round parentheses represent intervals excluding the boundary. For example, (1, 3] represents the interval $1 < x \leq 3$.

$\sum_{i=1}^n x_i$: The sum $x_1 + x_2 + \cdots + x_n$.

$\prod_{i=1}^n x_i$: The product $x_1 \cdot x_2 \cdots x_n$.

\vdash : The symbol for logical entailment. For example, $A \vdash B$ denotes that B follows deductively from A .

$>_g$: The symbol for the *more general than* relation. For example, $h_i >_g h_j$ denotes that hypothesis h_i is more general than h_j .

$\underset{x \in X}{\operatorname{argmax}} f(x)$: The value of x that maximizes $f(x)$. For example,

$$\underset{x \in \{1, 2, -3\}}{\operatorname{argmax}} x^2 = -3$$

$\hat{f}(x)$: A function that approximates the function $f(x)$.

δ : In PAC-learning, a bound on the probability of failure. In artificial neural network learning, the error term associated with a single unit output.

- ϵ : A bound on the error of a hypothesis (in PAC-learning).
- η : The learning rate in neural network and related learning methods.
- μ : The mean of a probability distribution.
- σ : The standard deviation of a probability distribution.
- $\nabla E(\vec{w})$: The gradient of E with respect to the vector \vec{w} .
- C : Class of possible target functions.
- D : The training data.
- \mathcal{D} : A probability distribution over the instance space.
- $E[x]$: The expected value of x .
- $E(\vec{w})$: The sum of squared errors of an artificial neural network whose weights are given by the vector \vec{w} .
- Error*: The error in a discrete-valued hypothesis or prediction.
- H : Hypothesis space.
- $h(x)$: The prediction produced by hypothesis h for instance x .
- $P(x)$: The probability (mass) of x .
- $\Pr(x)$: The probability (mass) of the event x .
- $p(x)$: The probability density of x .
- $Q(s, a)$: The Q function from reinforcement learning.
- \mathfrak{R} : The set of real numbers.
- $VC(H)$: The Vapnik-Chervonenkis dimension of the hypothesis space H .
- $VS_{H,D}$: The Version Space; that is, the set of hypotheses from H that are consistent with D .
- w_{ji} : In artificial neural networks, the weight from node i to node j .
- X : Instance space.

INDEXES

SUBJECT INDEX

Page numbers in italics refer to tables; numbers in bold to figures. An “*n*” following a page number refers to a footnote on that page.

A

- Absorbing state, 371
- ABSTRIPS, 329
- Acyclic neural networks. *See* Multilayer feedforward networks
- Adaline rule. *See* Delta rule
- Additive Chernoff bounds, 210–211
- Adelines, 123
- Agents, in reinforcement learning, 368
- Agnostic learning, 210–211, 225
- ALVINN system, 82–83, **84**
- Analytical-inductive learning. *See* Inductive-analytical learning
- Analytical learning, 307–330
 - inductive learning, comparison with, 310, 328–329, 334–336, 362
- ANN learning. *See* Neural network learning
- ANNs. *See* Neural networks, artificial
- Antecedents of Horn clause, 285
- AQ algorithm, 279–280
- AQ14 algorithm, comparison with GABIL, 256, 258
- Arbitrary functions, representation by feedforward networks, 105–106
- Artificial intelligence, influence on machine learning, 4
- Artificial neural networks. *See* Neural networks, artificial
- ASSISTANT, 77
- Astronomical structures, machine learning classification of, 3
- Attributes:
 - choice of, in sequential vs. simultaneous covering algorithms, 280–281
 - continuous-valued, 72–73
 - cost-sensitive measures, 75–76
 - discrete-valued, 72
 - measures for selection of, 73–74, 77
 - missing values, strategies for, 75

- Autonomous vehicles, 3, 4, 82–83, **84**
- Average reward, 371

B

- Backgammon learning program. *See* TD-GAMMON
- BACKPROPAGATION algorithm, 83, 97, 124
 - applications of, 81, **84**, 85, 96, 113
 - convergence and local minima, 104–105
 - definition of, 98
 - discovery of hidden layer representations, 106–109, 123
- feedforward networks as hypothesis space, 105–106
- gradient descent search, 89, 115–116, 123
- inductive bias of, 106
- KBANN algorithm:
 - comparison with, 344–345
 - use in, 339
- momentum, addition of, 100, 104
- overfitting in, 108, 110–111
- in *Q* learning, 384
- search of hypothesis space, 97, 106, 122–123
 - in decision tree learning, comparison with, 106
 - by genetic algorithms, comparison with, 259
 - by KBANN and TANGENTPROP algorithms, comparison with, 350–351
- stochastic gradient descent version, 98–100, 104–105, 107–108
- TANGENTPROP algorithm, comparison with, 349
- weight update rule:
 - alternative error functions, 117–118
 - derivation of, 101–102
 - for hidden unit weights, 103

- in KBANN algorithm, 343–344
- optimization methods, 119
- for output unit weights, 102–103, 171
- Backtracking, ID3 algorithm and, 62
- Backward chaining search for explanation generation, 314
- Baldwin effect, 250, 267
 - computational models for, 267–268
- Bayes classifier, naive. *See* Naive Bayes classifier
- Bayes optimal classifier, 174–176, 197, 222
 - learning Boolean concepts using version spaces, 176
- Bayes optimal learner. *See* Bayes optimal classifier
- Bayes rule. *See* Bayes theorem
- Bayes theorem, 4, 156–159
 - in BRUTE-FORCE MAP LEARNING algorithm, 160–162
 - concept learning and, 158–163
 - in inductive-analytical learning, 338
- Bayesian belief networks, 184–191
 - choice among alternative networks, 190
 - conditional independence in, 185
 - constraint-based approaches in, 191
 - gradient ascent search in, 188–190
 - inference methods, 187–188
 - joint probability distribution representation, 185–187
 - learning from training data, 188–191
 - naive Bayes classifier, comparison with, 186
 - representation of causal knowledge, 187
- Bayesian classifiers, 198. *See also* Bayes optimal classifier; Naive Bayes classifier
- Bayesian learning, 154–198
 - decision tree learning, comparison with, 198
- Bayesian methods, influence on machine learning, 4
- Beam search:
 - general-to-specific. *See* General-to-specific beam search
 - generate-and-test. *See* Generate-and-test beam search
- Bellman-Ford shortest path algorithm, 386, 387
- Bellman residual errors, 385
- Bellman's equation, 385–386
- BFS-ID3 algorithm, 63
- Binomial distribution, 133–137, 143, 151
- Biological evolution, 249, 250, 266–267
- Biological neural networks, comparison with artificial neural networks, 82
- Bit strings, 252–253, 258–259, 269
- Blocks, stacking of. *See* Stacking problems
- Body of Horn clause, 285
- Boolean conjunctions, PAC learning of, 211–212
- Boolean functions:
 - representation by feedforward networks, 105–106
 - representation by perceptrons, 87–88
- Boundary set representation for version spaces, 31–36
 - definition of, 31
- Bounds:
 - one-sided, 141, 144
 - two-sided, 141
- Brain, neural activity in, 82
- Breadth first search in ID3 algorithm, 63
- BRUTE-FORCE MAP LEARNING algorithm, 159–162
 - Bayes theorem in, 160–162
- C**
- C4.5 algorithm, 55, 77
- GABIL, comparison with, 256, 258
- missing attribute values, method for handling, 75
- rule post-pruning in, 71–72
- CADET system, 241–244
- CANDIDATE-ELIMINATION algorithm, 29–37, 45–47
 - applications of, 29, 302
 - Bayesian interpretation of, 163
 - computation of version spaces, 32–36
 - definition of, 33
 - ID3 algorithm, comparison with, 61–64
 - inductive bias of, 43–46, 63–64
 - limitations of, 29, 37, 41, 42, 46
 - search of hypothesis space, 64
- Candidate specializations:
 - generated by FOCL algorithm, 357–361
 - generated by FOIL algorithm, 287–288, 357–358

- CART system, 77
- CASCADE-CORRELATION algorithm, 121–123
- Case-based reasoning, 231, 240–244, 246, 247
- advantages of, 243–244
 - applications of, 240
 - other instance-based learning methods, comparison with, 240
- Causal knowledge, representation by Bayesian belief networks, 187
- Central Limit Theorem, 133, 142–143, 167
- Checkers learning program, 2–3, 5–14, 387
- algorithms for, 14
 - design, 13
 - as sequential control process, 369
- Chemical mass spectroscopy,
- CANDIDATE-ELIMINATION* algorithm in, 29
- Chess learning program, 308–310
- explanation-based learning in, 325
- Chunking, 327, 330
- CGOL, 302
- Circuit design, genetic programming in, 265–266
- Circuit layout, genetic algorithms in, 256
- Classification problems, 54
- CLASSIFY_NAIVE_BAYES.TEXT, 182–183
- CLAUDIEN, 302
- Clauses, 284, 285
- CLS. *See* Concept Learning System
- Clustering, 191
- CN2 algorithm, 278, 301
- choice of attribute-pairs in, 280–281
- Complexity, sample. *See* Sample complexity
- Computational complexity, 202
- Computational complexity theory, influence on machine learning, 4
- Computational learning theory, 201–227
- Concept learning, 20–47
- algorithms for, 47
 - Bayes theorem and, 158–163
 - definition of, 21
 - genetic algorithms in, 256
 - ID3 algorithm specialized for, 56
 - notation for, 22–23
- search of hypothesis space, 23–25, 46–47
- task design in, 21–22
- Concept Learning System, 77
- Concepts, partially learned, 38–39
- Conditional independence, 185
- in Bayesian belief networks, 186–187
- Confidence intervals, 133, 138–141, 150, 151
- for discrete-valued hypotheses, 131–132, 140–141
 - derivation of, 142–143
 - one-sided, 144, 145
- Conjugate gradient method, 119
- Conjunction of boolean literals, PAC learning of, 211–212
- Consequent of Horn clause, 285
- Consistent learners, 162–163
- bound on sample complexity, 207–210, 225
 - equation for, 209
- Constants, in logic, 284, 285
- Constraint-based approaches in Bayesian belief networks, 191
- Constructive induction, 292
- Continuous functions, representation by feedforward networks, 105–106
- Continuous-valued hypotheses, training error of, 89–90
- Continuous-valued target function, 197
- maximum likelihood (ML) hypothesis for, 164–167
- Control theory, influence on machine learning, 4
- Convergence of *Q* learning algorithm:
- in deterministic environments, 377–380, 386
 - in nondeterministic environments, 382–383, 386
- Credit assignment, 5
- Critic, 12, 13
- Cross entropy, 170
- minimization of, 118
- Cross-validation, 111–112
- for comparison of learning algorithms, 145–151
- k*-fold. *See* *k*-fold cross-validation
- in *k*-NEAREST NEIGHBOR algorithm, 235

leave-one-out, 235
 in neural network learning, 111–112
Crossover mask, 254
Crossover operators, 252–254, 261,
 262
 single-point, 254, 261
 two-point, 254, 257–258
 uniform, 255
Crowding, 259
Cumulative reward, 371
Curse of dimensionality, 235

D

Data mining, 17
Decision tree learning, 52–77
 algorithms for, 55, 77. *See also* C4.5
 algorithm; ID3 algorithm
 applications of, 54
 Bayesian learning, comparison with, 198
 impact of pruning on accuracy, 128–129
 inductive bias in, 63–66
k-NEAREST NEIGHBOR algorithm,
 comparison with, 235
Minimum Description Length principle
 in, 173–174
 neural network learning, comparison
 with, 85
 overfitting in, 67–69, 76–77, 111
 post-pruning in, 68–69, 77
 reduced-error pruning in, 69–71
 rule post-pruning in, 71–72, 281
 search of hypothesis space, 60–62
 by BACKPROPAGATION algorithm,
 comparison with, 106
Deductive learning, 321–322
Degrees of freedom, 147
Delayed learning methods, comparison
 with eager learning, 244–245
Delayed reward, in reinforcement learning,
 369
Delta rule, 11, 88–90, 94, 99, 123
Demes, 268
Determinations, 325
Deterministic environments, *Q* learning
 algorithm for, 375
Directed acyclic neural networks. *See*
 Multilayer feedforward networks
Discounted cumulative reward, 371

Discrete-valued hypotheses:
 confidence intervals for, 131–132,
 140–141
 derivation of, 142–143
 training error of, 205
Discrete-valued target functions,
 approximation by decision tree
 learning, 52
Disjunctive sets of rules, learning by
 sequential covering algorithms,
 275–276
Distance-weighted *k*-NEAREST NEIGHBOR
 algorithm, 233–234
Domain-independent learning algorithms,
 336
Domain theory, 310, 329. *See also*
 Imperfect domain theory; Perfect
 domain theory; Prior knowledge
 in analytical learning, 311–312
 as KBANN neural network, 342–343
 in PROLOG-EBG, 322
 weighting of components in EBNN,
 351–352
DYNA, 380
Dynamic programming:
 applications to reinforcement learning,
 380
 reinforcement learning and, 385–387

E

Eager learning methods, comparison with
 lazy learning, 244–245
EBG algorithm, 313
EBNN algorithm, 351–356, 362, 387
 other explanation-based learning
 methods, comparison with, 356
 prior knowledge and gradient descent in,
 339
TANGENTPROP algorithm in, 353
 weighting of inductive-analytical
 components in, 355, 362
EGGS algorithm, 313
EM algorithm, 190–196, 197
 applications of, 191, 194
 derivation of algorithm for *k*-means,
 195–196
 search for maximum likelihood (ML)
 hypothesis, 194–195

- Entailment, 321*n*
 relationship with θ -subsumption and more_general_than partial ordering, 299–300
- Entropy, 55–57, 282
 of optimal code, 172*n*
- Environment, in reinforcement learning, 368
- Equivalent sample size, 179–180
- Error bars for discrete-valued hypotheses.
See Confidence intervals, for discrete-valued hypotheses
- Error of hypotheses:
 sample. *See* Sample error
 training. *See* Training error
 true. *See* True error
- Estimation bias, 133, 137–138, 151
- Estimator, 133, 137–138, 143, 150–151
- Evolution of populations:
 argument for Occam’s razor, 66
 in genetic algorithms, 260–262
- Evolutionary computation, 250, 262
 applications of, 269
- Example-driven search, comparison with generate-and-test beam search, 281
- Expected value, 133, 136
- Experiment generator, 12–13
- Explanation-based learning, 312–330
 applications of, 325–328
 derivation of new features, 320–321
 inductive bias in, 322–323
 inductive learning and, 330
 lazy methods in, 328
 limitations of, 308, 329
 prior knowledge in, 308–309
 reinforcement learning and, 330
 utility analysis in, 327–328
- Explanations generated by backward chaining search, 314
- Explicit prior knowledge, 329
- Exploration in reinforcement learning, 369
- F**
- Face recognition, 17
 BACKPROPAGATION algorithm in, 81, 112–117
- Feedforward networks. *See* Multilayer feedforward networks
- FIND-S algorithm, 26–28, 46
 Bayesian interpretation of, 162–163
 definition of, 26
 inductive bias of, 45
 limitations of, 28–29
 mistake-bound learning in, 220–221
 PAC learning of boolean conjunctions with, 212
 search of hypothesis space, 27–28
- Finite horizon reward, 371
- First-order Horn clauses, 283–284, 318–319. *See also* First-order rules in analytical learning, 311 in PROLOG-EBG, 313, 314
- First-order logic, basic definitions, 285
- First-order representations, applications of, 275
- First-order resolution rule, 296–297
- First-order rules, 274–275, 283, 301, 302.
See also First-order Horn clauses in FOIL algorithm, 285–291 propositional rules, comparison with, 283
- Fitness function, 250–252, 255–256, 258
- Fitness proportionate selection, 255
- Fitness sharing, 259
- FOCL algorithm, 302
 extensions to FOIL, 357
 search step alteration with prior knowledge, 339–340
- FOIL algorithm, 286, 290–291, 302
 extensions in FOCL, 357
 information gain measure in, 289
 LEARN-ONE-RULE and sequential covering algorithms, comparison with, 287
 learning first-order rules in, 285–291
 post-pruning in, 291
 recursive rule learning in, 290
- Function approximation, 8
- Function approximation algorithms: choice of, 9–11
 as lookup table substitute, 384
- Functions, in logic, 284, 285
- G**
- GABIL, 256–259, 269
 C4.5 and AQ14 algorithms, comparison with, 256, 258

- extensions to, 258–259
 ID5R algorithm, comparison with, 258
Gain ratio, 73–74
GAs. *See* Genetic algorithms
Gaussian distribution. *See* Normal distribution
Gaussian kernel function, 238–240
General-to-specific beam search, 277–279, 302
 advantages of, 281
 in CN2 algorithm, 278
 in FOCL algorithm, 357–361
 in FOIL algorithm, 287, 357–358
General-to-specific ordering of hypotheses, 24–25, 45–46. *See also* More-general-than partial ordering
Generalization accuracy in neural networks, 110–111
Generalizer, 12, 13
Generate-and-test beam search, 250
 example-driven search, comparison with, 281
 inverse entailment operators, comparison with, 299
 inverse resolution, comparison with, 298–299
Genetic algorithms, 249–270
 advantages of, 250
 applications of, 256, 269
 fitness function in, 255–256
 limitations of, 259
 parallelization of, 268
 representation of hypotheses, 252–253
 search of hypothesis space, 259, 268–269
Genetic operators, 252–255, 257, 261–262
Genetic programming, 250, 262–266, 269
 applications of, 265, 269
 performance of, 266
 representation in, 262–263
Gibbs algorithm, 176
Global method, 234
GOLEM, 281
GP. *See* Genetic programming
Gradient ascent search, 170–171
 in Bayesian belief networks, 188–190
Gradient descent search, 89–91, 93, 97, 115–116, 123
 in EBNN algorithm, 339
least-squared error hypothesis in, 167
limitations of, 92
weight update rule, 91–92, 237
 stochastic approximation to, 92–94, 98–100, 104–105, 107–108
Gradient of error, 91
Greedy search:
 in sequential covering algorithms, 276–278
 in PROLOG-EBG, 323
GRENDEL program, 303
Ground literal, 285
- ## H
- HALVING algorithm**, 223
 mistake-bound learning in, 221–222
Handwriting recognition, 3–4
BACKPROPAGATION algorithm in, 81
TANGENTPROP algorithm in, 348–349
Head of Horn clause, 285
Hidden layer representations, discovery by BACKPROPAGATION algorithm, 106–109, 123
Hidden units:
 BACKPROPAGATION weight tuning rule for, 103
CASCADE-CORRELATION algorithm, addition by, 121–123
 choice in radial basis function networks, 239–240
 in face recognition task, 115–117
Hill-climbing search:
 in FOIL algorithm, 286, 287
 in genetic algorithms, 268
 in ID3 algorithm, 60–61
Hoeffding bounds, 210–211
Horn clauses, 284, 285
Horn clauses, first-order. *See* First-order Horn clauses
Human learning:
 explanations in, 309
 prior knowledge in, 330
Hypotheses. *See also* Discrete-valued hypotheses; General-to-specific ordering of hypotheses; Hypothesis space
 error differences between two, 143–144
 estimation of accuracy, 129–130

- Hypotheses, estimation of accuracy
(continued)
- bias and variance in estimate, 129, 151, 152
 - errors in, 129–131, 151
 - evaluation of, 128–129
 - justification of, in inductive vs. analytical learning, 334–336
 - representations of, 23
 - testing of, 144–145
- Hypothesis space, 14–15
- bias in, 40–42, 46, 129
 - finite, sample complexity for, 207–214, 225
 - infinite, sample complexity for, 214–220
 - VC dimension of, 214–217
- Hypothesis space search
- by BACKPROPAGATION algorithm, 97, 106, 122–123
 - comparison with decision tree learning, 106
 - comparison with KBANN and TANGENTPROP algorithms, 350–351
 - by CANDIDATE-ELIMINATION algorithm, 64
 - in concept learning, 23–25, 46–47
 - constraints on, 302–303
 - by FIND-S algorithm, 27–28
 - by FOIL algorithm, 286–287, 357–361
 - by genetic algorithms, 250, 259
 - by gradient descent, 90–91
 - by ID3 algorithm, 60–62, 64, 76
 - by KBANN algorithm, 346
 - by learning algorithms, 24
 - by LEARN-ONE-RULE, 277
 - in machine learning, 14–15, 18
 - use of prior knowledge, 339–340, 362
- I**
- ID3 algorithm, 55–64, 77
- backtracking and, 62
 - CANDIDATE-ELIMINATION algorithm, comparison with, 61–62
 - choice of attributes in, 280–281
 - choice of decision tree, 63
 - cost-sensitive measures, 75–76
 - extensions to, 77. *See also* C4.5 algorithm
- inductive bias of, 63–64, 76
- LEARN-ONE-RULE, search comparison with, 277
- limitations of, 61–62
- overfitting in, 67–68
- search of hypothesis space, 60–62, 64, 76
- sequential covering algorithms, comparison with, 280–281
- specialized for concept learning, 56
- use of information gain in, 58–60
- ID5R algorithm, comparison with GABIL, 258
- ILP. *See* Inductive logic programming
- Image encoding in face recognition, 114
- Imperfect domain theory:
- in EBNN algorithm, 356
 - in explanation-based learning, 330
 - in FOCL algorithm, 360
 - in KBANN algorithm, 344–345
- Incremental explanation methods, 328
- Incremental gradient descent. *See* Stochastic gradient descent
- INCREMENTAL VERSION SPACE MERGING algorithm, 47
- Inductive-analytical learning, 334–363
- advantages of, 362
 - explanation-based learning and, 330
 - learning problem, 337–338
 - prior knowledge methods to alter search, 339–340, 362
 - properties of ideal systems, 337
 - weighting of components in EBNN algorithm, 351–352, 355
 - weighting prior knowledge in, 338
- Inductive bias, 39–45, 137–138. *See also* Occam's razor; Preference bias; Restriction bias
- of BACKPROPAGATION algorithm, 106
 - bias-free learning, 40–42
 - of CANDIDATE-ELIMINATION algorithm, 43–46, 63–64
 - in decision tree learning, 63–66
 - definition of, 43
 - in explanation-based learning, 322–323
 - of FIND-S algorithm, 45
 - of ID3 algorithm, 63–64, 76
 - of inductive learning algorithms, 42–46
 - of *k*-NEAREST NEIGHBOR algorithm, 234

- of LMS algorithm, 64
 of ROTE-LEARNER algorithm, 44–45
- I**
 Inductive inference. *See* Inductive learning
 Inductive learning, 42, 307–308. *See also* Decision tree learning; Genetic algorithms; Inductive logic programming; Neural network learning
 analytical learning, comparison with, 310, 328–329, 334–336, 362
 inductive bias in, 42–46
 Inductive learning hypothesis, 23
 Inductive logic programming, 275, 291
 PROLOG-EBG, comparison with, 322
 Information gain, 73
 definition of, 57–58
 in FOIL algorithm, 289
 in ID3 algorithm, 55, 58–60
 Information theory:
 influence on machine learning, 4
 Minimum Description Length principle and, 172
 Initialize-the-hypothesis approach, 339–346
 Bayesian belief networks in, 346
 Instance-based learning, 230–247. *See also* Case-based reasoning; *k*-NEAREST NEIGHBOR algorithm; Locally weighted regression
 advantages, 245–246
 case-based reasoning, comparison with other methods, 240
 limitations of, 231
 Inverse entailment, 292, 302
 first-order, 297
 generate-and-test beam search, comparison with, 299
 in PROGOL, 300–302
 Inverse resolution, 294–296, 302
 first-order, 297–298
 generate-and-test beam search, comparison with, 298–299
 limitations of, 300
 Inverted deduction, 291–293
- J**
 Jacobian, 354
 Job-shop scheduling, genetic algorithms in, 256
- K**
 Joint probability distribution, in Bayesian belief networks, 185–187
- L**
k-fold cross-validation, 112, 147, 150
k-means problem, 191–193
 derivation of EM algorithm for, 195–196
k-NEAREST NEIGHBOR algorithm, 231–233, 246
 applications of, 234
 cross-validation in, 235
 decision tree and rule learning, comparison with, 235
 distance-weighted, 233–234
 inductive bias of, 234
 memory indexing in, 236
k-term CNF expressions, 213–214
k-term DNF expressions, 213–214
 K2 algorithm, 190–191
 KBANN algorithm, 340–347, 362, 387
 advantages of, 344
 BACKPROPAGATION algorithm, comparison with, 344–345
 BACKPROPAGATION weight update rule in, 343–344
 hypothesis space search by
 BACKPROPAGATION and
 TANGENTPROP, comparison with, 350–351
 limitations of, 345
 prior knowledge in, 339
 kd-tree, 236
 Kernel function, 236, 238, 246
 Kernel function, Gaussian. *See* Gaussian kernel function
 Knowledge-Based Artificial Neural Network (KBANN) algorithm. *See* KBANN algorithm
 Knowledge compilation, 320
 Knowledge level learning, 323–325
 Knowledge reformulation, 320
- M**
 Lamarckian evolution, 266
 Language bias. *See* Restriction bias
 Lazy explanation methods, 328
 Lazy learning methods, comparison with eager learning, 244–245
 LEARN_NAIVE_BAYES_TEXT, 182–183

- LEARN-ONE-RULE** algorithm:
 FOIL algorithm, comparison with, 287
 ID3 algorithm, search comparison with,
 277
 rule performance in, 282
 rule post-pruning in, 281
 variations of, 279–280, 286
- Learning:**
 human. *See* Human learning
 machine. *See* Machine learning
- Learning algorithms**
 consistent learners, 162–163
 design of, 9–11, 17
 domain-independent, 336
 error differences between two, 145–151
 search of hypothesis space, 24
- Learning problems**, 2–5, 17
 computational theory of, 201–202
 in inductive-analytical learning, 337–338
- Learning rate**, 88, 91
- Learning systems:**
 design of, 5–14, 17
 program modules, 11–14
- Least mean squares algorithm.** *See* LMS algorithm
- Least-squared error hypothesis:**
 classifiers for, 198
 gradient descent in, 167
 maximum likelihood (ML) hypothesis
 and, 164–167
- Leave-one-out cross-validation**, 235
- Legal case reasoning**, case-based reasoning
 in, 240
- LEMMA-ENUMERATOR** algorithm, 324
- Lifelong learning**, 370
- Line search**, 119
- Linear programming**, as weight update
 algorithm, 95
- Linearly separable sets**, 86, 89, 95
- List-THEN-ELIMINATE** algorithm, 30
- Literal**, 284, 285
- LMS algorithm**, 11, 15
 inductive bias of, 64
- LMS weight update rule.** *See* Delta rule
- Local method**, 234
- Locally weighted regression**, 231,
 236–238, 246
 limitations of, 238
 weight update rules in, 237–238
- Logical constants, 284, 285
 Logical terms, 284, 285
 Logistic function, 96, 104
Lookup table:
 function approximation algorithms as
 substitute, 384
 neural network as substitute, 384
- Lower bound on sample complexity**,
 217–218
- M**
- m*-estimate of probability, 179–180, 198,
 282
- Machine learning**, 15. *See also* entries
 beginning with Learning
 applications, 3, 17
 definition of, 2
 influence of other disciplines on, 4, 17
 search of hypothesis space, 14–15, 18
- Manufacturing process control**, 17
- MAP hypothesis.** *See* Maximum
 a posteriori hypothesis
- MAP LEARNING algorithm**, BRUTE-FORCE.
See BRUTE-FORCE MAP LEARNING
 algorithm
- Markov decision processes (MDP)**, 370,
 387
 applications of, 386
- MARKUS**, 302
- MARVIN**, 302
- Maximally general hypotheses**,
 computation by CANDIDATE-
 ELIMINATION algorithm, 31,
 46
- Maximally specific hypotheses**:
 computation by CANDIDATE-
 ELIMINATION algorithm, 31,
 46
 computation by FIND-S algorithm,
 26–28, 62–63
- Maximum a posteriori (MAP) hypothesis**,
 157, 197. *See also* BRUTE-FORCE
 MAP LEARNING algorithm
- naive Bayes classifier and, 178
 output of consistent learners, 162–163
- Maximum likelihood (ML) hypothesis**, 157
 EM algorithm search for, 194–195
 least-squared error hypothesis and,
 164–167

- prediction of probabilities with, 167–170
 MDP. *See* Markov decision processes
 Mean error, 143
 Mean value, 133, 136
 Means-ends planner, 326
 Mechanical design, case-based reasoning in, 240–244
 Medical diagnosis:
 attribute selection measure, 76
 Bayes theorem in, 157–158
 META-DENDRAL, 302
 mFOIL, 302
 Minimum Description Length principle, 66, 69, 171–173, 197, 198
 in decision tree learning, 173–174
 in inductive logic programming, 292–293
 MIS, 302
 Mistake-bound learning, 202, 220, 226
 in CANDIDATE-ELIMINATION algorithm, 221–222
 in FIND-S algorithm, 220–221
 in HALVING algorithm, 221–222
 in LIST-THEN-ELIMINATE algorithm, 221–222
 in WEIGHTED-MAJORITY algorithm, 224–225
 Mistake bounds, optimal. *See* Optimal mistake bounds
 ML hypothesis. *See* Maximum likelihood hypothesis
 Momentum, addition to BACKPROPAGATION algorithm, 100, 104
 More-general-than partial ordering, 24–28, 46
 in CANDIDATE-ELIMINATION algorithm, 29
 in FIND-S algorithm, 26–28
 θ -subsumption, entailment, and, 299–300
 in version spaces, 31
 Multilayer feedforward networks
 BACKPROPAGATION algorithm in, 95–101
 function representation in, 105–106, 115
 representation of decision surfaces, 96
 training of multiple networks, 105
 VC dimension of, 218–220
 Mutation operator, 252, 253, 255, 257, 262
- N**
- Naive Bayes classifier, 154–155, 177–179, 197
 Bayesian belief network, comparison with, 186
 maximum a posteriori (MAP) hypothesis and, 178
 use in text classification, 180–184
 Naive Bayes learner. *See* Naive Bayes classifier
 Negation-as-failure strategy, 279, 319, 321n
 Negative literal, 284, 285
 Neural network learning, 81–124.
 See also BACKPROPAGATION algorithm; CASCADE-CORRELATION algorithm; EBNN algorithm; KBANN algorithm; TANGENTPROP algorithm
 applications of, 83, 85
 in face recognition, 113
 cross-validation in, 111–112
 decision tree learning, comparison with, 85
 discovery of hidden layer representations in, 107
 overfitting in, 123
 in Q learning, 384
 representation in, 82–83, 84, 105–106
 Neural networks, artificial, 81–124.
 See also Multilayer feedforward networks; Radial basis function networks; Recurrent networks
 biological neural networks, comparison with, 82
 creation by KBANN algorithm, 342–343
 VC dimension of, 218–220
 Neural networks, biological, 82
 Neurobiology, influence on machine learning, 4, 82
 New features:
 derivation in BACKPROPAGATION algorithm, 106–109, 123
 derivation in explanation-based learning, 320–321
 NEWSWEEDER system, 183–184
 Nondeterministic environments, Q learning in, 381–383

Normal distribution, 133, 139–140, 143, 151, 165
for noise, 167
in paired tests, 149

O

Occam's razor, 4, 65–66, 171
Offline learning systems, 385
One-sided bounds, 141, 144
Online learning systems, 385
Optimal brain damage approach, 122
Optimal code, 172
Optimal mistake bounds, 222–223
Optimal policy for selecting actions, 371–372
Optimization problems:
 explanation-based learning in, 325
 genetic algorithms in, 256, 269
 reinforcement learning in, 256
Output encoding in face recognition, 114–115
Output units, BACKPROPAGATION weight update rule for, 102–103
Overfitting, 123
 in BACKPROPAGATION algorithm, 108, 110–111
 in decision tree learning, 66–69, 76–77, 111
 definition of, 67
Minimum Description Length principle and, 174
in neural network learning, 123

P

PAC learning, 203–207, 225, 226
of boolean conjunctions, 211–212
definition of, 206–207
training error in, 205
true error in, 204–205
Paired tests, 147–150, 152
Parallelization in genetic algorithms, 268
Partially learned concepts, 38–39
Partially observable states in reinforcement learning, 369–370
Perceptron training rule, 88–89, 94, 95
Perceptrons, 86, 95, 96, 123
 representation of boolean functions, 87–88

VC dimension of, 219
weight update rule, 88–89, 94, 95
Perfect domain theory, 312–313
Performance measure, 6
Performance system, 11–12, 13
Philosophy, influence on machine learning, 4
Planning problems:
 PRODIGY in, 327
 case-based reasoning in, 240–241
Policy for selecting actions, 370–372
Population evolution, in genetic algorithms, 260–262
Positive literal, 284, 285
Post-pruning:
 in decision tree learning, 68–69, 77, 281
 in FOIL algorithm, 291
 in LEARN-ONE-RULE, 281
Posterior probability, 155–156, 162
Power law of practice, 4
Power set, 40–42
Predicates, 284, 285
Preference bias, 64, 76, 77
Prior knowledge, 155–156, 336. *See also Domain theory*
 to augment search operators, 357–361
 in Bayesian learning, 155
 derivatives of target function, 346–356, 362
 in explanation-based learning, 308–309
 explicit, use in learning, 329
 in human learning, 330
 initialize-the-hypothesis approach, 339–346, 362
 in PROLOG-EBG, 313
 search alteration in inductive-analytical learning, 339–340, 362
 weighting in inductive-analytical learning, 338, 362
Prioritized sweeping, 380
Probabilistic reasoning, 163
Probabilities:
 estimation of, 179–180
 formulas, 159
 maximum likelihood (ML) hypothesis for prediction of, 167–170
Probability density, 165

Probability distribution, 133. *See also* Binomial distribution; Normal distribution
 Probably approximately correct (PAC) learning. *See* PAC learning
 Process control in manufacturing, 17
 PRODIGY, 326–327, 330
 Product rule, 159
 PROGOL, 300–302
 PROLOG, 275, 302, 330
 PROLOG-EBG, 313–321, 328–329
 applications of, 325
 deductive learning in, 321–322
 definition of, 314
 derivation of new features in, 320–321
 domain theory in, 322
 EBNN algorithm, comparison with, 356
 explanation of training examples, 314–318
 weakest preimage in, 329
 inductive bias in, 322–323
 inductive logic programming, comparison with, 322
 limitations of, 329
 perfect domain theory in, 313
 prior knowledge in, 313
 properties of, 319
 regression process in, 316–318
 Propositional rules:
 learning by sequential covering algorithms, 275
 learning first-order rules, comparison with, 283
 Psychology, influence on machine learning, 4

Q

Q function:
 in deterministic environments, 374
 convergence of *Q* learning towards, 377–380
 in nondeterministic environments, 381
 convergence of *Q* learning towards, 382
Q learning algorithm, 372–376. *See also* Reinforcement learning
 advantages of, 386
 in deterministic environments, 375

convergence, 377–380
 training rule, 375–376
 experimentation strategies in, 379
 lookup table, neural network substitution for, 384
 in nondeterministic environments, 381–383
 convergence, 382–383
 training rule, 382
 updating sequence, 379
 Query strategies, 37–38

R

Radial basis function networks, 231, 238–240, 245, 246, 247
 advantages of, 240
 Random variable, 133, 134, 137, 151
 Randomized method, 150
 Rank selection, 256
 RBF networks. *See* Radial basis function networks
 RDT program, 303
 Real-valued target function. *See* Continuous-valued target function
 Recurrent networks, 119–121. *See also* Neural networks, artificial
 Recursive rules, 284
 learning by FOIL algorithm, 290
 Reduced-error pruning, in decision tree learning, 69–71
 REGRESS algorithm, 317–318
 Regression, 236
 in PROLOG-EBG, 316–381
 Reinforcement learning, 367–387. *See also* *Q* learning algorithm
 applications of, 387
 differences from other methods, 369–370
 dynamic programming and, 380, 385–387
 explanation-based learning and, 330
 function approximation algorithms in, 384–385
 Relational descriptions, learning of, 302
 Relative frequency, 282
 Relative mistake bound for WEIGHTED-MAJORITY algorithm, 224–225
 Residual, 236

Resolution rule, 293–294

 first-order, 296–297

 inverse entailment operator and,
 294–296

 propositional, 294

Restriction bias, 64

Reward function, in reinforcement learning, 368

Robot control:

 by BACKPROPAGATION and EBNN

 algorithms, comparison of, 356

 genetic programming in, 269

Robot driving. *See* Autonomous vehicles

Robot perception, attribute cost measures in, 76

Robot planning problems, explanation-based learning in, 327

ROUTE-LEARNER algorithm, inductive bias of, 44–45

Roulette wheel selection, 255

Rule for estimating training values, 10, 383

Rule learning, 274–303

 in decision trees, 71–72

 in explanation-based learning, 311–319

 by FOCL algorithm, 357–360

 by genetic algorithms, 256–259,
 269–270, 274

Rule post-pruning, in decision tree learning, 71–72

Rules:

 disjunctive sets of, learning by sequential covering algorithms, 275–276

 first-order. *See* First-order rules

 propositional. *See* Propositional rules

S

SafeToStack, 310–312

Sample complexity, 202. *See also* Training examples

 bound for consistent learners, 207–210,
 225

 equation for, 209

 for finite hypothesis spaces, 207–214

 for infinite hypothesis spaces, 214–220

 of k -term CNF and DNF expressions,
 213–214

 of unbiased concepts, 212–213

 VC dimension bound, 217–218

Sample error, 130–131, 133–134, 143

 training error and, 205

Sampling theory, 132–141

Scheduling problems:

 case-based reasoning in, 241

 explanation-based learning in, 325

 PRODIGY in, 327

 reinforcement learning in, 368

Schema theorem, 260–262

 genetic operators in, 261–262

Search bias. *See* Preference bias

Search control problems:

 explanation-based learning in, 325–328,
 329, 330

 limitations of, 327–328

 as sequential control processes, 369

Search of hypothesis space. *See* Hypothesis space search

Sequential control processes, 368–369

 learning task in, 370–373

 search control problems in, 369

Sequential covering algorithms, 274,
 275–279, 301, 313, 363

 choice of attribute-pairs in, 280–282

 definition of, 276

 FOIL algorithm, comparison with, 287,
 301–302

 ID3 algorithm, comparison with,
 280–281

 simultaneous covering algorithms,
 comparison with, 280–282
 variations of, 279–280, 286

Shattering, 214–215

Shepard's method, 234

Sigmoid function, 97, 104

Sigmoid units, 95–96, 115

Simultaneous covering algorithms:

 choice of attributes in, 280–281

 sequential covering algorithms,
 comparison with, 280–282

Single-point crossover operator, 254, 261

SOAR, 327, 330

Specific-to-general search, 281

 in FOIL algorithm, 287

Speech recognition, 3

 BACKPROPAGATION algorithm in, 81

 representation by multilayer network,
 95, 96

 weight sharing in, 118

Speedup learning, 325, 330
SPHINX, 3
 Split information, 73–74
 Squashing function, 96
 Stacking problems. *See also* SafeToStack
 analytical learning in, 310
 explanation-based learning in, 310
 genetic programming in, 263–265
 PRODIGY in, 327
 Standard deviation, 133, 136–137
 State-transition function, 380
 Statistics:
 basic definitions, 133
 influence on machine learning, 4
 Stochastic gradient descent, 93–94,
 98–100, 104–105
 Student *t* tests, 147–150, 152
 Substitution, 285, 296
 Sum rule, 159

T

t tests, 147–150, 152
TANGENTPROP algorithm, 347–350, 362
 BACKPROPAGATION algorithm,
 comparison with, 349
 in EBNN algorithm, 352
 search of hypothesis space
 by KBANN and BACKPROPAGATION
 algorithms, comparison with,
 350–351
 tanh function, 97
 Target concept, 22–23, 40–41
 PAC learning of, 211–213
 Target function, 7–8, 17
 continuous-valued. *See* Continuous-
 valued target function
 representation of, 8–9, 14, 17
TD-GAMMON, 3, 14, 369, 383
 $\text{TD}(\lambda)$ and BACKPROPAGATION algorithm
 in, 384
TD(λ), 383–384, 387
 Temporal credit assignment, in
 reinforcement learning, 369
 Temporal difference learning, 383–384,
 386–387
 Terms, in logic, 284, 285
 Text classification, naive Bayes classifier
 in, 180–184

Theorem of total probability, 159
 θ -subsumption, 302
 relationship with entailment and
 more general than partial ordering,
 299–300

Tournament selection, 256
 Training and validation set approach, 69.
See also Validation set

Training derivatives, 117–118

Training error:
 of continuous-valued hypotheses, 89–90
 of discrete-valued hypotheses, 205
 in multilayer networks, 98
 alternative error functions, 117–118

Training examples, 5–6, 17, 23. *See also*
 Sample complexity
 explanation in PROLOG-EBG, 314–318
 in PAC learning, 205–207
 bounds on, 226

Voronoi diagram of, 233

Training experience, 5–6, 17

Training values, rule for estimation of, 10

True error, 130–131, 133, 137, 150,

204–205

of two hypotheses, differences in,
 143–144

in version spaces, 208–209

Two-point crossover operator, 255,
 257–258

Two-sided bounds, 141

U

Unbiased estimator, 133, 137

Unbiased learners, 40–42

 sample complexity of, 212–213

Uniform crossover operator, 255

Unifying substitution, 285, 296

Unsupervised learning, 191

Utility analysis, in explanation-based
 learning, 327–328

V

Validation set. *See also* Training and
 validation set approach

 cross-validation and, 111–112

 error over, 110

Vapnik-Chervonenkis (VC) dimension. *See*

 VC dimension

- Variables, in logic, 284, 285
Variance, 133, 136–137, 138, 143
VC dimension, 214–217, 226
 bound on sample complexity, 217–218
 definition of, 215
 of neural networks, 218–220
Version space representation theorem, 32
Version spaces, 29–39, 46, 47, 207–208
 Bayes optimal classifier and, 176
 definition of, 30
 exhaustion of, 208–210, 226
 representations of, 30–32
Voronoi diagram, 233
- W**
- Weakest preimage, 316, 329
Weight decay, 111, 117
Weight sharing, 118

- Weight update rules, 10–11
 BACKPROPAGATION weight update rule, 101–103
 alternative error functions, 117–118
 in KBANN algorithm, 343–344
 optimization methods, 119
 output units, 171
 delta rule, 11, 88–90, 94
 gradient ascent, 170–171
 gradient descent, 91–92, 95
 linear programming, 95
 perceptron training rule, 88–89
 stochastic gradient descent, 93–94
WEIGHTED-MAJORITY algorithm, 222–226
 mistake-bound learning in, 224–225
Weighted voting, 222, 223, 226
Widrow-Hoff rule. *See* Delta rule