CHAPTER 1

INTRODUCTION

Ever since computers were invented, we have wondered whether they might be made to learn. If we could understand how to program them to learn—to improve automatically with experience—the impact would be dramatic. Imagine computers learning from medical records which treatments are most effective for new diseases, houses learning from experience to optimize energy costs based on the particular usage patterns of their occupants, or personal software assistants learning the evolving interests of their users in order to highlight especially relevant stories from the online morning newspaper. A successful understanding of how to make computers learn would open up many new uses of computers and new levels of competence and customization. And a detailed understanding of information-processing algorithms for machine learning might lead to a better understanding of human learning abilities (and disabilities) as well.

We do not yet know how to make computers learn nearly as well as people learn. However, algorithms have been invented that are effective for certain types of learning tasks, and a theoretical understanding of learning is beginning to emerge. Many practical computer programs have been developed to exhibit useful types of learning, and significant commercial applications have begun to appear. For problems such as speech recognition, algorithms based on machine learning outperform all other approaches that have been attempted to date. In the field known as data mining, machine learning algorithms are being used routinely to discover valuable knowledge from large commercial databases containing equipment maintenance records, loan applications, financial transactions, medical records, and the like. As our understanding of computers continues to mature, it

seems inevitable that machine learning will play an increasingly central role in computer science and computer technology.

A few specific achievements provide a glimpse of the state of the art: programs have been developed that successfully learn to recognize spoken words (Waibel 1989; Lee 1989), predict recovery rates of pneumonia patients (Cooper et al. 1997), detect fraudulent use of credit cards, drive autonomous vehicles on public highways (Pomerleau 1989), and play games such as backgammon at levels approaching the performance of human world champions (Tesauro 1992, 1995). Theoretical results have been developed that characterize the fundamental relationship among the number of training examples observed, the number of hypotheses under consideration, and the expected error in learned hypotheses. We are beginning to obtain initial models of human and animal learning and to understand their relationship to learning algorithms developed for computers (e.g., Laird et al. 1986; Anderson 1991; Qin et al. 1992; Chi and Bassock 1989; Ahn and Brewer 1993). In applications, algorithms, theory, and studies of biological systems, the rate of progress has increased significantly over the past decade. Several recent applications of machine learning are summarized in Table 1.1. Langley and Simon (1995) and Rumelhart et al. (1994) survey additional applications of machine learning.

This book presents the field of machine learning, describing a variety of learning paradigms, algorithms, theoretical results, and applications. Machine learning is inherently a multidisciplinary field. It draws on results from artificial intelligence, probability and statistics, computational complexity theory, control theory, information theory, philosophy, psychology, neurobiology, and other fields. Table 1.2 summarizes key ideas from each of these fields that impact the field of machine learning. While the material in this book is based on results from many diverse fields, the reader need not be an expert in any of them. Key ideas are presented from these fields using a nonspecialist's vocabulary, with unfamiliar terms and concepts introduced as the need arises.

1.1 WELL-POSED LEARNING PROBLEMS

Let us begin our study of machine learning by considering a few learning tasks. For the purposes of this book we will define learning broadly, to include any computer program that improves its performance at some task through experience. Put more precisely,

> ***Definition***: A computer program is said to **learn** from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

For example, a computer program that learns to play checkers might improve its performance *as measured by its ability to win* at the class of tasks involving *playing checkers games*, through experience *obtained by playing games against itself*. In general, to have a well-defined learning problem, we must identity these

- Learning to recognize spoken words.

  All of the most successful speech recognition systems employ machine learning in some form. For example, the SPHINX system (e.g., Lee 1989) learns speaker-specific strategies for recognizing the primitive sounds (phonemes) and words from the observed speech signal. Neural network learning methods (e.g., Waibel et al. 1989) and methods for learning hidden Markov models (e.g., Lee 1989) are effective for automatically customizing to individual speakers, vocabularies, microphone characteristics, background noise, etc. Similar techniques have potential applications in many signal-interpretation problems.

- Learning to drive an autonomous vehicle.

  Machine learning methods have been used to train computer-controlled vehicles to steer correctly when driving on a variety of road types. For example, the ALVINN system (Pomerleau 1989) has used its learned strategies to drive unassisted at 70 miles per hour for 90 miles on public highways among other cars. Similar techniques have possible applications in many sensor-based control problems.

- Learning to classify new astronomical structures.

  Machine learning methods have been applied to a variety of large databases to learn general regularities implicit in the data. For example, decision tree learning algorithms have been used by NASA to learn how to classify celestial objects from the second Palomar Observatory Sky Survey (Fayyad et al. 1995). This system is now used to automatically classify all objects in the Sky Survey, which consists of three terrabytes of image data.

- Learning to play world-class backgammon.

  The most successful computer programs for playing games such as backgammon are based on machine learning algorithms. For example, the world's top computer program for backgammon, TD-GAMMON (Tesauro 1992, 1995), learned its strategy by playing over one million practice games against itself. It now plays at a level competitive with the human world champion. Similar techniques have applications in many practical problems where very large search spaces must be examined efficiently.

TABLE 1.1
Some successful applications of machine learning.

three features: the class of tasks, the measure of performance to be improved, and the source of experience.

A checkers learning problem:

- Task T: playing checkers
- Performance measure P: percent of games won against opponents
- Training experience E: playing practice games against itself

We can specify many learning problems in this fashion, such as learning to recognize handwritten words, or learning to drive a robotic automobile autonomously.

A handwriting recognition learning problem:

- Task T: recognizing and classifying handwritten words within images
- Performance measure P: percent of words correctly classified

- Artificial intelligence

  Learning symbolic representations of concepts. Machine learning as a search problem. Learning as an approach to improving problem solving. Using prior knowledge together with training data to guide learning.
- Bayesian methods

  Bayes' theorem as the basis for calculating probabilities of hypotheses. The naive Bayes classifier. Algorithms for estimating values of unobserved variables.
- Computational complexity theory

  Theoretical bounds on the inherent complexity of different learning tasks, measured in terms of the computational effort, number of training examples, number of mistakes, etc. required in order to learn.
- Control theory

  Procedures that learn to control processes in order to optimize predefined objectives and that learn to predict the next state of the process they are controlling.
- Information theory

  Measures of entropy and information content. Minimum description length approaches to learning. Optimal codes and their relationship to optimal training sequences for encoding a hypothesis.
- Philosophy

  Occam's razor, suggesting that the simplest hypothesis is the best. Analysis of the justification for generalizing beyond observed data.
- Psychology and neurobiology

  The power law of practice, which states that over a very broad range of learning problems, people's response time improves with practice according to a power law. Neurobiological studies motivating artificial neural network models of learning.
- Statistics

  Characterization of errors (e.g., bias and variance) that occur when estimating the accuracy of a hypothesis based on a limited sample of data. Confidence intervals, statistical tests.

TABLE 1.2
Some disciplines and examples of their influence on machine learning.

- Training experience E: a database of handwritten words with given classifications

A robot driving learning problem:

- Task T: driving on public four-lane highways using vision sensors
- Performance measure P: average distance traveled before an error (as judged by human overseer)
- Training experience E: a sequence of images and steering commands recorded while observing a human driver

Our definition of learning is broad enough to include most tasks that we would conventionally call "learning" tasks, as we use the word in everyday language. It is also broad enough to encompass computer programs that improve from experience in quite straightforward ways. For example, a database system

that allows users to update data entries would fit our definition of a learning system: it improves its performance at answering database queries, based on the experience gained from database updates. Rather than worry about whether this type of activity falls under the usual informal conversational meaning of the word "learning," we will simply adopt our technical definition of the class of programs that improve through experience. Within this class we will find many types of problems that require more or less sophisticated solutions. Our concern here is not to analyze the meaning of the English word "learning" as it is used in everyday language. Instead, our goal is to define precisely a class of problems that encompasses interesting forms of learning, to explore algorithms that solve such problems, and to understand the fundamental structure of learning problems and processes.

1.2 DESIGNING A LEARNING SYSTEM

In order to illustrate some of the basic design issues and approaches to machine learning, let us consider designing a program to learn to play checkers, with the goal of entering it in the world checkers tournament. We adopt the obvious performance measure: the percent of games it wins in this world tournament.

1.2.1 Choosing the Training Experience

The first design choice we face is to choose the type of training experience from which our system will learn. The type of training experience available can have a significant impact on success or failure of the learner. One key attribute is whether the training experience provides direct or indirect feedback regarding the choices made by the performance system. For example, in learning to play checkers, the system might learn from *direct* training examples consisting of individual checkers board states and the correct move for each. Alternatively, it might have available only *indirect* information consisting of the move sequences and final outcomes of various games played. In this later case, information about the correctness of specific moves early in the game must be inferred indirectly from the fact that the game was eventually won or lost. Here the learner faces an additional problem of *credit assignment*, or determining the degree to which each move in the sequence deserves credit or blame for the final outcome. Credit assignment can be a particularly difficult problem because the game can be lost even when early moves are optimal, if these are followed later by poor moves. Hence, learning from direct training feedback is typically easier than learning from indirect feedback.

A second important attribute of the training experience is the degree to which the learner controls the sequence of training examples. For example, the learner might rely on the teacher to select informative board states and to provide the correct move for each. Alternatively, the learner might itself propose board states that it finds particularly confusing and ask the teacher for the correct move. Or the learner may have complete control over both the board states and (indirect) training classifications, as it does when it learns by playing against itself with no teacher

present. Notice in this last case the learner may choose between experimenting with novel board states that it has not yet considered, or honing its skill by playing minor variations of lines of play it currently finds most promising. Subsequent chapters consider a number of settings for learning, including settings in which training experience is provided by a random process outside the learner's control, settings in which the learner may pose various types of queries to an expert teacher, and settings in which the learner collects training examples by autonomously exploring its environment.

A third important attribute of the training experience is how well it represents the distribution of examples over which the final system performance P must be measured. In general, learning is most reliable when the training examples follow a distribution similar to that of future test examples. In our checkers learning scenario, the performance metric P is the percent of games the system wins in the world tournament. If its training experience E consists only of games played against itself, there is an obvious danger that this training experience might not be fully representative of the distribution of situations over which it will later be tested. For example, the learner might never encounter certain crucial board states that are very likely to be played by the human checkers champion. In practice, it is often necessary to learn from a distribution of examples that is somewhat different from those on which the final system will be evaluated (e.g., the world checkers champion might not be interested in teaching the program!). Such situations are problematic because mastery of one distribution of examples will not necessary lead to strong performance over some other distribution. We shall see that most current theory of machine learning rests on the crucial assumption that the distribution of training examples is identical to the distribution of test examples. Despite our need to make this assumption in order to obtain theoretical results, it is important to keep in mind that this assumption must often be violated in practice.

To proceed with our design, let us decide that our system will train by playing games against itself. This has the advantage that no external trainer need be present, and it therefore allows the system to generate as much training data as time permits. We now have a fully specified learning task.

A checkers learning problem:

- Task T: playing checkers
- Performance measure P: percent of games won in the world tournament
- Training experience E: games played against itself

In order to complete the design of the learning system, we must now choose

1. the exact type of knowledge to be learned
2. a representation for this target knowledge
3. a learning mechanism

1.2.2 Choosing the Target Function

The next design choice is to determine exactly what type of knowledge will be learned and how this will be used by the performance program. Let us begin with a checkers-playing program that can generate the *legal* moves from any board state. The program needs only to learn how to choose the *best* move from among these legal moves. This learning task is representative of a large class of tasks for which the legal moves that define some large search space are known a priori, but for which the best search strategy is not known. Many optimization problems fall into this class, such as the problems of scheduling and controlling manufacturing processes where the available manufacturing steps are well understood, but the best strategy for sequencing them is not.

Given this setting where we must learn to choose among the legal moves, the most obvious choice for the type of information to be learned is a program, or function, that chooses the best move for any given board state. Let us call this function *ChooseMove* and use the notation $ChooseMove : B \rightarrow M$ to indicate that this function accepts as input any board from the set of legal board states B and produces as output some move from the set of legal moves M. Throughout our discussion of machine learning we will find it useful to reduce the problem of improving performance P at task T to the problem of learning some particular *target function* such as *ChooseMove*. The choice of the target function will therefore be a key design choice.

Although *ChooseMove* is an obvious choice for the target function in our example, this function will turn out to be very difficult to learn given the kind of indirect training experience available to our system. An alternative target function—and one that will turn out to be easier to learn in this setting—is an evaluation function that assigns a numerical score to any given board state. Let us call this target function V and again use the notation $V : B \rightarrow \Re$ to denote that V maps any legal board state from the set B to some real value (we use $\Re$ to denote the set of real numbers). We intend for this target function V to assign higher scores to better board states. If the system can successfully learn such a target function V, then it can easily use it to select the best move from any current board position. This can be accomplished by generating the successor board state produced by every legal move, then using V to choose the best successor state and therefore the best legal move.

What exactly should be the value of the target function V for any given board state? Of course any evaluation function that assigns higher scores to better board states will do. Nevertheless, we will find it useful to define one particular target function V among the many that produce optimal play. As we shall see, this will make it easier to design a training algorithm. Let us therefore define the target value $V(b)$ for an arbitrary board state b in B, as follows:

1. if b is a final board state that is won, then $V(b) = 100$
2. if b is a final board state that is lost, then $V(b) = -100$
3. if b is a final board state that is drawn, then $V(b) = 0$

4. if b is a not a final state in the game, then $V(b) = V(b')$, where b' is the best final board state that can be achieved starting from b and playing optimally until the end of the game (assuming the opponent plays optimally, as well).

While this recursive definition specifies a value of $V(b)$ for every board state b, this definition is not usable by our checkers player because it is not efficiently computable. Except for the trivial cases (cases 1–3) in which the game has already ended, determining the value of $V(b)$ for a particular board state requires (case 4) searching ahead for the optimal line of play, all the way to the end of the game! Because this definition is not efficiently computable by our checkers playing program, we say that it is a *nonoperational* definition. The goal of learning in this case is to discover an *operational* description of V; that is, a description that can be used by the checkers-playing program to evaluate states and select moves within realistic time bounds.

Thus, we have reduced the learning task in this case to the problem of discovering an *operational description of the ideal target function* V. It may be very difficult in general to learn such an operational form of V perfectly. In fact, we often expect learning algorithms to acquire only some *approximation* to the target function, and for this reason the process of learning the target function is often called *function approximation*. In the current discussion we will use the symbol $\hat{V}$ to refer to the function that is actually learned by our program, to distinguish it from the ideal target function V.

1.2.3 Choosing a Representation for the Target Function

Now that we have specified the ideal target function V, we must choose a representation that the learning program will use to describe the function $\hat{V}$ that it will learn. As with earlier design choices, we again have many options. We could, for example, allow the program to represent $\hat{V}$ using a large table with a distinct entry specifying the value for each distinct board state. Or we could allow it to represent $\hat{V}$ using a collection of rules that match against features of the board state, or a quadratic polynomial function of predefined board features, or an artificial neural network. In general, this choice of representation involves a crucial tradeoff. On one hand, we wish to pick a very expressive representation to allow representing as close an approximation as possible to the ideal target function V. On the other hand, the more expressive the representation, the more training data the program will require in order to choose among the alternative hypotheses it can represent. To keep the discussion brief, let us choose a simple representation: for any given board state, the function $\hat{V}$ will be calculated as a linear combination of the following board features:

- x_1: the number of black pieces on the board
- x_2: the number of red pieces on the board
- x_3: the number of black kings on the board
- x_4: the number of red kings on the board

- x_5: the number of black pieces threatened by red (i.e., which can be captured on red's next turn)
- x_6: the number of red pieces threatened by black

Thus, our learning program will represent $\hat{V}(b)$ as a linear function of the form

$$\hat{V}(b) = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4 + w_5 x_5 + w_6 x_6$$

where w_0 through w_6 are numerical coefficients, or weights, to be chosen by the learning algorithm. Learned values for the weights w_1 through w_6 will determine the relative importance of the various board features in determining the value of the board, whereas the weight w_0 will provide an additive constant to the board value.

To summarize our design choices thus far, we have elaborated the original formulation of the learning problem by choosing a type of training experience, a target function to be learned, and a representation for this target function. Our elaborated learning task is now

Partial design of a checkers learning program:

- Task T: playing checkers
- Performance measure P: percent of games won in the world tournament
- Training experience E: games played against itself
- *Target function:* $V: Board \rightarrow \Re$
- *Target function representation*

$$\hat{V}(b) = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4 + w_5 x_5 + w_6 x_6$$

The first three items above correspond to the specification of the learning task, whereas the final two items constitute design choices for the implementation of the learning program. Notice the net effect of this set of design choices is to reduce the problem of learning a checkers strategy to the problem of learning values for the coefficients w_0 through w_6 in the target function representation.

1.2.4 Choosing a Function Approximation Algorithm

In order to learn the target function $\hat{V}$ we require a set of training examples, each describing a specific board state b and the training value $V_{train}(b)$ for b. In other words, each training example is an ordered pair of the form $\langle b, V_{train}(b)\rangle$. For instance, the following training example describes a board state b in which black has won the game (note $x_2 = 0$ indicates that red has no remaining pieces) and for which the target function value $V_{train}(b)$ is therefore +100.

$$\langle\langle x_1 = 3, x_2 = 0, x_3 = 1, x_4 = 0, x_5 = 0, x_6 = 0\rangle, +100\rangle$$

Below we describe a procedure that first derives such training examples from the indirect training experience available to the learner, then adjusts the weights w_i to best fit these training examples.

1.2.4.1 ESTIMATING TRAINING VALUES

Recall that according to our formulation of the learning problem, the only training information available to our learner is whether the game was eventually won or lost. On the other hand, we require training examples that assign specific scores to specific board states. While it is easy to assign a value to board states that correspond to the end of the game, it is less obvious how to assign training values to the more numerous *intermediate* board states that occur before the game's end. Of course the fact that the game was eventually won or lost does not necessarily indicate that *every* board state along the game path was necessarily good or bad. For example, even if the program loses the game, it may still be the case that board states occurring early in the game should be rated very highly and that the cause of the loss was a subsequent poor move.

Despite the ambiguity inherent in estimating training values for intermediate board states, one simple approach has been found to be surprisingly successful. This approach is to assign the training value of $V_{train}(b)$ for any intermediate board state b to be $\hat{V}(Successor(b))$, where $\hat{V}$ is the learner's current approximation to V and where $Successor(b)$ denotes the next board state following b for which it is again the program's turn to move (i.e., the board state following the program's move and the opponent's response). This rule for estimating training values can be summarized as

Rule for estimating training values.

$$V_{train}(b) \leftarrow \hat{V}(Successor(b)) \tag{1.1}$$

While it may seem strange to use the current version of $\hat{V}$ to estimate training values that will be used to refine this very same function, notice that we are using estimates of the value of the $Successor(b)$ to estimate the value of board state b. Intuitively, we can see this will make sense if $\hat{V}$ tends to be more accurate for board states closer to game's end. In fact, under certain conditions (discussed in Chapter 13) the approach of iteratively estimating training values based on estimates of successor state values can be proven to converge toward perfect estimates of V_{train}.

1.2.4.2 ADJUSTING THE WEIGHTS

All that remains is to specify the learning algorithm for choosing the weights w_i to best fit the set of training examples $\{\langle b, V_{train}(b)\rangle\}$. As a first step we must define what we mean by the *best fit* to the training data. One common approach is to define the best hypothesis, or set of weights, as that which minimizes the squared error E between the training values and the values predicted by the hypothesis $\hat{V}$.

$$E \equiv \sum_{\langle b, V_{train}(b)\rangle \in \text{ training examples}} (V_{train}(b) - \hat{V}(b))^2$$

Thus, we seek the weights, or equivalently the $\hat{V}$, that minimize E for the observed training examples. Chapter 6 discusses settings in which minimizing the sum of squared errors is equivalent to finding the most probable hypothesis given the observed training data.

Several algorithms are known for finding weights of a linear function that minimize E defined in this way. In our case, we require an algorithm that will incrementally refine the weights as new training examples become available and that will be robust to errors in these estimated training values. One such algorithm is called the least mean squares, or LMS training rule. For each observed training example it adjusts the weights a small amount in the direction that reduces the error on this training example. As discussed in Chapter 4, this algorithm can be viewed as performing a stochastic gradient-descent search through the space of possible hypotheses (weight values) to minimize the squared error E. The LMS algorithm is defined as follows:

LMS weight update rule.

For each training example $\langle b, V_{train}(b) \rangle$

- Use the current weights to calculate $\hat{V}(b)$
- For each weight w_i, update it as

$$w_i \leftarrow w_i + \eta \, (V_{train}(b) - \hat{V}(b)) \, x_i$$

Here η is a small constant (e.g., 0.1) that moderates the size of the weight update. To get an intuitive understanding for why this weight update rule works, notice that when the error $(V_{train}(b) - \hat{V}(b))$ is zero, no weights are changed. When $(V_{train}(b) - \hat{V}(b))$ is positive (i.e., when $\hat{V}(b)$ is too low), then each weight is increased in proportion to the value of its corresponding feature. This will raise the value of $\hat{V}(b)$, reducing the error. Notice that if the value of some feature x_i is zero, then its weight is not altered regardless of the error, so that the only weights updated are those whose features actually occur on the training example board. Surprisingly, in certain settings this simple weight-tuning method can be proven to converge to the least squared error approximation to the V_{train} values (as discussed in Chapter 4).

1.2.5 The Final Design

The final design of our checkers learning system can be naturally described by four distinct program modules that represent the central components in many learning systems. These four modules, summarized in Figure 1.1, are as follows:

- The **Performance System** is the module that must solve the given performance task, in this case playing checkers, by using the learned target function(s). It takes an instance of a new problem (new game) as input and produces a trace of its solution (game history) as output. In our case, the

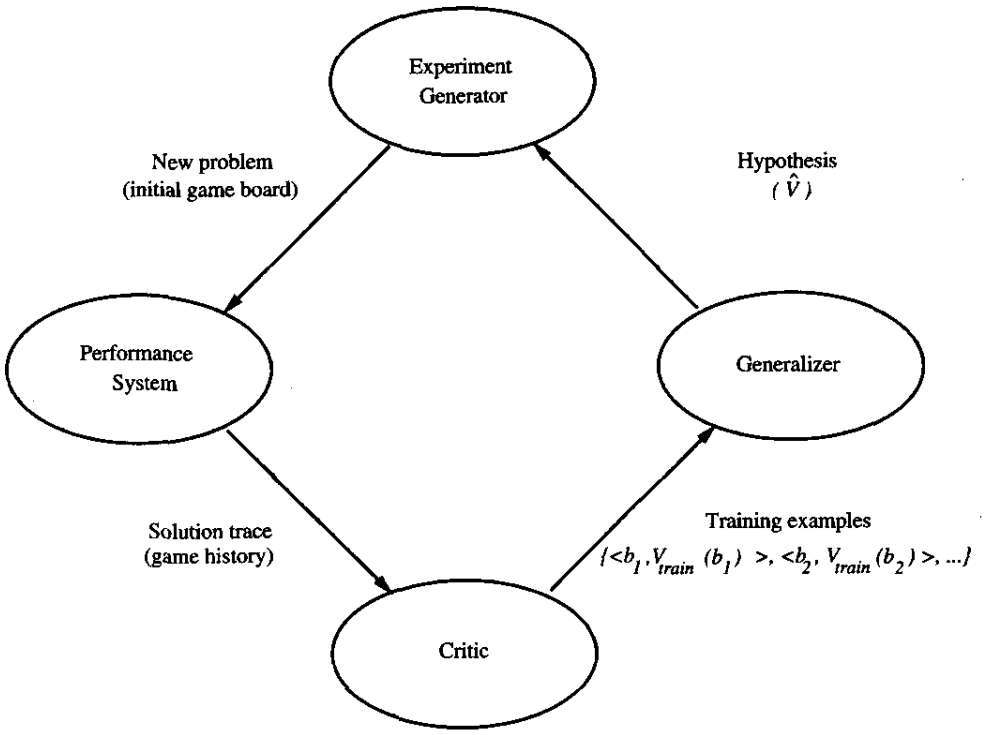


FIGURE 1.1
Final design of the checkers learning program.

strategy used by the Performance System to select its next move at each step is determined by the learned $\hat{V}$ evaluation function. Therefore, we expect its performance to improve as this evaluation function becomes increasingly accurate.

- The **Critic** takes as input the history or trace of the game and produces as output a set of training examples of the target function. As shown in the diagram, each training example in this case corresponds to some game state in the trace, along with an estimate V_{train} of the target function value for this example. In our example, the Critic corresponds to the training rule given by Equation (1.1).
- The **Generalizer** takes as input the training examples and produces an output hypothesis that is its estimate of the target function. It generalizes from the specific training examples, hypothesizing a general function that covers these examples and other cases beyond the training examples. In our example, the Generalizer corresponds to the LMS algorithm, and the output hypothesis is the function $\hat{V}$ described by the learned weights $w_0, \ldots, w_6$.
- The **Experiment Generator** takes as input the current hypothesis (currently learned function) and outputs a new problem (i.e., initial board state) for the Performance System to explore. Its role is to pick new practice problems that will maximize the learning rate of the overall system. In our example, the Experiment Generator follows a very simple strategy: It always proposes the same initial game board to begin a new game. More sophisticated strategies

could involve creating board positions designed to explore particular regions of the state space.

Together, the design choices we made for our checkers program produce specific instantiations for the performance system, critic, generalizer, and experiment generator. Many machine learning systems can be usefully characterized in terms of these four generic modules.

The sequence of design choices made for the checkers program is summarized in Figure 1.2. These design choices have constrained the learning task in a number of ways. We have restricted the type of knowledge that can be acquired to a single linear evaluation function. Furthermore, we have constrained this evaluation function to depend on only the six specific board features provided. If the true target function V can indeed be represented by a linear combination of these

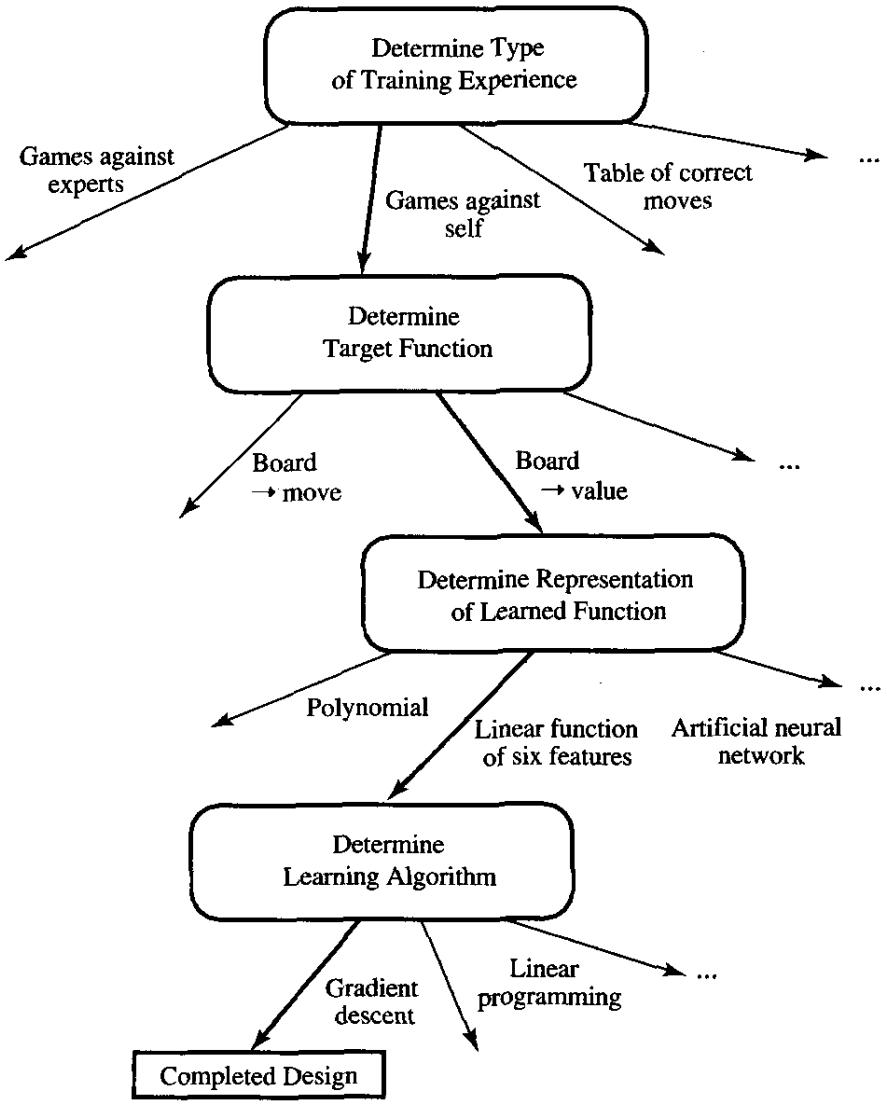


FIGURE 1.2
Summary of choices in designing the checkers learning program.

particular features, then our program has a good chance to learn it. If not, then the best we can hope for is that it will learn a good approximation, since a program can certainly never learn anything that it cannot at least represent.

Let us suppose that a good approximation to the true V function can, in fact, be represented in this form. The question then arises as to whether this learning technique is guaranteed to find one. Chapter 13 provides a theoretical analysis showing that under rather restrictive assumptions, variations on this approach do indeed converge to the desired evaluation function for certain types of search problems. Fortunately, practical experience indicates that this approach to learning evaluation functions is often successful, even outside the range of situations for which such guarantees can be proven.

Would the program we have designed be able to learn well enough to beat the human checkers world champion? Probably not. In part, this is because the linear function representation for $\hat{V}$ is too simple a representation to capture well the nuances of the game. However, given a more sophisticated representation for the target function, this general approach can, in fact, be quite successful. For example, Tesauro (1992, 1995) reports a similar design for a program that learns to play the game of backgammon, by learning a very similar evaluation function over states of the game. His program represents the learned evaluation function using an artificial neural network that considers the complete description of the board state rather than a subset of board features. After training on over one million self-generated training games, his program was able to play very competitively with top-ranked human backgammon players.

Of course we could have designed many alternative algorithms for this checkers learning task. One might, for example, simply store the given training examples, then try to find the "closest" stored situation to match any new situation (nearest neighbor algorithm, Chapter 8). Or we might generate a large number of candidate checkers programs and allow them to play against each other, keeping only the most successful programs and further elaborating or mutating these in a kind of simulated evolution (genetic algorithms, Chapter 9). Humans seem to follow yet a different approach to learning strategies, in which they analyze, or explain to themselves, the reasons underlying specific successes and failures encountered during play (explanation-based learning, Chapter 11). Our design is simply one of many, presented here to ground our discussion of the decisions that must go into designing a learning method for a specific class of tasks.

1.3 PERSPECTIVES AND ISSUES IN MACHINE LEARNING

One useful perspective on machine learning is that it involves searching a very large space of possible hypotheses to determine one that best fits the observed data and any prior knowledge held by the learner. For example, consider the space of hypotheses that could in principle be output by the above checkers learner. This hypothesis space consists of all evaluation functions that can be represented by some choice of values for the weights w_0 through w_6. The learner's task is thus to search through this vast space to locate the hypothesis that is most consistent with

the available training examples. The LMS algorithm for fitting weights achieves this goal by iteratively tuning the weights, adding a correction to each weight each time the hypothesized evaluation function predicts a value that differs from the training value. This algorithm works well when the hypothesis representation considered by the learner defines a continuously parameterized space of potential hypotheses.

Many of the chapters in this book present algorithms that search a hypothesis space defined by some underlying representation (e.g., linear functions, logical descriptions, decision trees, artificial neural networks). These different hypothesis representations are appropriate for learning different kinds of target functions. For each of these hypothesis representations, the corresponding learning algorithm takes advantage of a different underlying structure to organize the search through the hypothesis space.

Throughout this book we will return to this perspective of learning as a search problem in order to characterize learning methods by their search strategies and by the underlying structure of the search spaces they explore. We will also find this viewpoint useful in formally analyzing the relationship between the size of the hypothesis space to be searched, the number of training examples available, and the confidence we can have that a hypothesis consistent with the training data will correctly generalize to unseen examples.

1.3.1 Issues in Machine Learning

Our checkers example raises a number of generic questions about machine learning. The field of machine learning, and much of this book, is concerned with answering questions such as the following:

- What algorithms exist for learning general target functions from specific training examples? In what settings will particular algorithms converge to the desired function, given sufficient training data? Which algorithms perform best for which types of problems and representations?
- How much training data is sufficient? What general bounds can be found to relate the confidence in learned hypotheses to the amount of training experience and the character of the learner's hypothesis space?
- When and how can prior knowledge held by the learner guide the process of generalizing from examples? Can prior knowledge be helpful even when it is only approximately correct?
- What is the best strategy for choosing a useful next training experience, and how does the choice of this strategy alter the complexity of the learning problem?
- What is the best way to reduce the learning task to one or more function approximation problems? Put another way, what specific functions should the system attempt to learn? Can this process itself be automated?
- How can the learner automatically alter its representation to improve its ability to represent and learn the target function?

1.4 HOW TO READ THIS BOOK

This book contains an introduction to the primary algorithms and approaches to machine learning, theoretical results on the feasibility of various learning tasks and the capabilities of specific algorithms, and examples of practical applications of machine learning to real-world problems. Where possible, the chapters have been written to be readable in any sequence. However, some interdependence is unavoidable. If this is being used as a class text, I recommend first covering Chapter 1 and Chapter 2. Following these two chapters, the remaining chapters can be read in nearly any sequence. A one-semester course in machine learning might cover the first seven chapters, followed by whichever additional chapters are of greatest interest to the class. Below is a brief survey of the chapters.

- Chapter 2 covers concept learning based on symbolic or logical representations. It also discusses the general-to-specific ordering over hypotheses, and the need for inductive bias in learning.
- Chapter 3 covers decision tree learning and the problem of overfitting the training data. It also examines Occam's razor—a principle recommending the shortest hypothesis among those consistent with the data.
- Chapter 4 covers learning of artificial neural networks, especially the well-studied BACKPROPAGATION algorithm, and the general approach of gradient descent. This includes a detailed example of neural network learning for face recognition, including data and algorithms available over the World Wide Web.
- Chapter 5 presents basic concepts from statistics and estimation theory, focusing on evaluating the accuracy of hypotheses using limited samples of data. This includes the calculation of confidence intervals for estimating hypothesis accuracy and methods for comparing the accuracy of learning methods.
- Chapter 6 covers the Bayesian perspective on machine learning, including both the use of Bayesian analysis to characterize non-Bayesian learning algorithms and specific Bayesian algorithms that explicitly manipulate probabilities. This includes a detailed example applying a naive Bayes classifier to the task of classifying text documents, including data and software available over the World Wide Web.
- Chapter 7 covers computational learning theory, including the Probably Approximately Correct (PAC) learning model and the Mistake-Bound learning model. This includes a discussion of the WEIGHTED MAJORITY algorithm for combining multiple learning methods.
- Chapter 8 describes instance-based learning methods, including nearest neighbor learning, locally weighted regression, and case-based reasoning.
- Chapter 9 discusses learning algorithms modeled after biological evolution, including genetic algorithms and genetic programming.

- Chapter 10 covers algorithms for learning sets of rules, including Inductive Logic Programming approaches to learning first-order Horn clauses.
- Chapter 11 covers explanation-based learning, a learning method that uses prior knowledge to explain observed training examples, then generalizes based on these explanations.
- Chapter 12 discusses approaches to combining approximate prior knowledge with available training data in order to improve the accuracy of learned hypotheses. Both symbolic and neural network algorithms are considered.
- Chapter 13 discusses reinforcement learning—an approach to control learning that accommodates indirect or delayed feedback as training information. The checkers learning algorithm described earlier in Chapter 1 is a simple example of reinforcement learning.

The end of each chapter contains a summary of the main concepts covered, suggestions for further reading, and exercises. Additional updates to chapters, as well as data sets and implementations of algorithms, are available on the World Wide Web at http://www.cs.cmu.edu/~tom/mlbook.html.

1.5 SUMMARY AND FURTHER READING

Machine learning addresses the question of how to build computer programs that improve their performance at some task through experience. Major points of this chapter include:

- Machine learning algorithms have proven to be of great practical value in a variety of application domains. They are especially useful in (a) data mining problems where large databases may contain valuable implicit regularities that can be discovered automatically (e.g., to analyze outcomes of medical treatments from patient databases or to learn general rules for credit worthiness from financial databases); (b) poorly understood domains where humans might not have the knowledge needed to develop effective algorithms (e.g., human face recognition from images); and (c) domains where the program must dynamically adapt to changing conditions (e.g., controlling manufacturing processes under changing supply stocks or adapting to the changing reading interests of individuals).
- Machine learning draws on ideas from a diverse set of disciplines, including artificial intelligence, probability and statistics, computational complexity, information theory, psychology and neurobiology, control theory, and philosophy.
- A well-defined learning problem requires a well-specified task, performance metric, and source of training experience.
- Designing a machine learning approach involves a number of design choices, including choosing the type of training experience, the target function to be learned, a representation for this target function, and an algorithm for learning the target function from training examples.

- Learning involves search: searching through a space of possible hypotheses to find the hypothesis that best fits the available training examples and other prior constraints or knowledge. Much of this book is organized around different learning methods that search different hypothesis spaces (e.g., spaces containing numerical functions, neural networks, decision trees, symbolic rules) and around theoretical results that characterize conditions under which these search methods converge toward an optimal hypothesis.

There are a number of good sources for reading about the latest research results in machine learning. Relevant journals include *Machine Learning*, *Neural Computation*, *Neural Networks*, *Journal of the American Statistical Association*, and the *IEEE Transactions on Pattern Analysis and Machine Intelligence*. There are also numerous annual conferences that cover different aspects of machine learning, including the International Conference on Machine Learning, Neural Information Processing Systems, Conference on Computational Learning Theory, International Conference on Genetic Algorithms, International Conference on Knowledge Discovery and Data Mining, European Conference on Machine Learning, and others.

EXERCISES

1.1. Give three computer applications for which machine learning approaches seem appropriate and three for which they seem inappropriate. Pick applications that are not already mentioned in this chapter, and include a one-sentence justification for each.

1.2. Pick some learning task not mentioned in this chapter. Describe it informally in a paragraph in English. Now describe it by stating as precisely as possible the task, performance measure, and training experience. Finally, propose a target function to be learned and a target representation. Discuss the main tradeoffs you considered in formulating this learning task.

1.3. Prove that the LMS weight update rule described in this chapter performs a gradient descent to minimize the squared error. In particular, define the squared error E as in the text. Now calculate the derivative of E with respect to the weight w_i, assuming that $\hat{V}(b)$ is a linear function as defined in the text. Gradient descent is achieved by updating each weight in proportion to $-\frac{\partial E}{\partial w_i}$. Therefore, you must show that the LMS training rule alters weights in this proportion for each training example it encounters.

1.4. Consider alternative strategies for the Experiment Generator module of Figure 1.2. In particular, consider strategies in which the Experiment Generator suggests new board positions by

- Generating random legal board positions
- Generating a position by picking a board state from the previous game, then applying one of the moves that was not executed
- A strategy of your own design

Discuss tradeoffs among these strategies. Which do you feel would work best if the number of training examples was held constant, given the performance measure of winning the most games at the world championships?

1.5. Implement an algorithm similar to that discussed for the checkers problem, but use the simpler game of tic-tac-toe. Represent the learned function $\hat{V}$ as a linear com-

bination of board features of your choice. To train your program, play it repeatedly against a second copy of the program that uses a fixed evaluation function you create by hand. Plot the percent of games won by your system, versus the number of training games played.

REFERENCES

Ahn, W., & Brewer, W. F. (1993). Psychological studies of explanation-based learning. In G. DeJong (Ed.), *Investigating explanation-based learning*. Boston: Kluwer Academic Publishers.

Anderson, J. R. (1991). The place of cognitive architecture in rational analysis. In K. VanLehn (Ed.), *Architectures for intelligence* (pp. 1–24). Hillsdale, NJ: Erlbaum.

Chi, M. T. H., & Bassock, M. (1989). Learning from examples via self-explanations. In L. Resnick (Ed.), *Knowing, learning, and instruction: Essays in honor of Robert Glaser*. Hillsdale, NJ: L. Erlbaum Associates.

Cooper, G., et al. (1997). An evaluation of machine-learning methods for predicting pneumonia mortality. *Artificial Intelligence in Medicine*, (to appear).

Fayyad, U. M., Uthurusamy, R. (Eds.) (1995). *Proceedings of the First International Conference on Knowledge Discovery and Data Mining*. Menlo Park, CA: AAAI Press.

Fayyad, U. M., Smyth, P., Weir, N., Djorgovski, S. (1995). Automated analysis and exploration of image databases: Results, progress, and challenges. *Journal of Intelligent Information Systems*, 4, 1–19.

Laird, J., Rosenbloom, P., & Newell, A. (1986). SOAR: The anatomy of a general learning mechanism. *Machine Learning*, 1(1), 11–46.

Langley, P., & Simon, H. (1995). Applications of machine learning and rule induction. *Communications of the ACM*, 38(11), 55–64.

Lee, K. (1989). *Automatic speech recognition: The development of the Sphinx system*. Boston: Kluwer Academic Publishers.

Pomerleau, D. A. (1989). *ALVINN: An autonomous land vehicle in a neural network*. (Technical Report CMU-CS-89-107). Pittsburgh, PA: Carnegie Mellon University.

Qin, Y., Mitchell, T., & Simon, H. (1992). Using EBG to simulate human learning from examples and learning by doing. *Proceedings of the Florida AI Research Symposium* (pp. 235–239).

Rudnicky, A. I., Hauptmann, A. G., & Lee, K. -F. (1994). Survey of current speech technology in artificial intelligence. *Communications of the ACM*, 37(3), 52–57.

Rumelhart, D., Widrow, B., & Lehr, M. (1994). The basic ideas in neural networks. *Communications of the ACM*, 37(3), 87–92.

Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine Learning*, 8, 257.

Tesauro, G. (1995). Temporal difference learning and TD-gammon. *Communications of the ACM*, 38(3), 58–68.

Waibel, A., Hanazawa, T., Hinton, G., Shikano, K., & Lang, K. (1989). Phoneme recognition using time-delay neural networks. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 37(3), 328–339.

CHAPTER 2

CONCEPT LEARNING AND THE GENERAL-TO-SPECIFIC ORDERING

The problem of inducing general functions from specific training examples is central to learning. This chapter considers concept learning: acquiring the definition of a general category given a sample of positive and negative training examples of the category. Concept learning can be formulated as a problem of searching through a predefined space of potential hypotheses for the hypothesis that best fits the training examples. In many cases this search can be efficiently organized by taking advantage of a naturally occurring structure over the hypothesis space—a general-to-specific ordering of hypotheses. This chapter presents several learning algorithms and considers situations under which they converge to the correct hypothesis. We also examine the nature of inductive learning and the justification by which any program may successfully generalize beyond the observed training data.

2.1 INTRODUCTION

Much of learning involves acquiring general concepts from specific training examples. People, for example, continually learn general concepts or categories such as "bird," "car," "situations in which I should study more in order to pass the exam," etc. Each such concept can be viewed as describing some subset of objects or events defined over a larger set (e.g., the subset of animals that constitute

birds). Alternatively, each concept can be thought of as a boolean-valued function defined over this larger set (e.g., a function defined over all animals, whose value is true for birds and false for other animals).

In this chapter we consider the problem of automatically inferring the general definition of some concept, given examples labeled as members or nonmembers of the concept. This task is commonly referred to as *concept learning*, or approximating a boolean-valued function from examples.

Concept learning. Inferring a boolean-valued function from training examples of its input and output.

2.2 A CONCEPT LEARNING TASK

To ground our discussion of concept learning, consider the example task of learning the target concept "days on which my friend Aldo enjoys his favorite water sport." Table 2.1 describes a set of example days, each represented by a set of *attributes*. The attribute *EnjoySport* indicates whether or not Aldo enjoys his favorite water sport on this day. The task is to learn to predict the value of *EnjoySport* for an arbitrary day, based on the values of its other attributes.

What hypothesis representation shall we provide to the learner in this case? Let us begin by considering a simple representation in which each hypothesis consists of a conjunction of constraints on the instance attributes. In particular, let each hypothesis be a vector of six constraints, specifying the values of the six attributes *Sky, AirTemp, Humidity, Wind, Water,* and *Forecast*. For each attribute, the hypothesis will either

- indicate by a "?" that any value is acceptable for this attribute,
- specify a single required value (e.g., *Warm*) for the attribute, or
- indicate by a "$\emptyset$" that no value is acceptable.

If some instance x satisfies all the constraints of hypothesis h, then h classifies x as a positive example ($h(x) = 1$). To illustrate, the hypothesis that Aldo enjoys his favorite sport only on cold days with high humidity (independent of the values of the other attributes) is represented by the expression

$$\langle ?, Cold, High, ?, ?, ? \rangle$$

Example	*Sky*	*AirTemp*	*Humidity*	*Wind*	*Water*	*Forecast*	*EnjoySport*
1	Sunny	Warm	Normal	Strong	Warm	Same	Yes
2	Sunny	Warm	High	Strong	Warm	Same	Yes
3	Rainy	Cold	High	Strong	Warm	Change	No
4	Sunny	Warm	High	Strong	Cool	Change	Yes

TABLE 2.1
Positive and negative training examples for the target concept *EnjoySport*.

The most general hypothesis—that every day is a positive example—is represented by

$$\langle ?, ?, ?, ?, ?, ? \rangle$$

and the most specific possible hypothesis—that *no* day is a positive example—is represented by

$$\langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$$

To summarize, the *EnjoySport* concept learning task requires learning the set of days for which $EnjoySport = yes$, describing this set by a conjunction of constraints over the instance attributes. In general, any concept learning task can be described by the set of instances over which the target function is defined, the target function, the set of candidate hypotheses considered by the learner, and the set of available training examples. The definition of the *EnjoySport* concept learning task in this general form is given in Table 2.2.

2.2.1 Notation

Throughout this book, we employ the following terminology when discussing concept learning problems. The set of items over which the concept is defined is called the set of *instances*, which we denote by X. In the current example, X is the set of all possible days, each represented by the attributes *Sky*, *AirTemp*, *Humidity*, *Wind*, *Water*, and *Forecast*. The concept or function to be learned is called the *target concept*, which we denote by c. In general, c can be any boolean-valued function defined over the instances X; that is, $c : X \rightarrow \{0, 1\}$. In the current example, the target concept corresponds to the value of the attribute *EnjoySport* (i.e., $c(x) = 1$ if $EnjoySport = Yes$, and $c(x) = 0$ if $EnjoySport = No$).

- **Given:**
  - Instances X: Possible days, each described by the attributes
    - *Sky* (with possible values *Sunny*, *Cloudy*, and *Rainy*),
    - *AirTemp* (with values *Warm* and *Cold*),
    - *Humidity* (with values *Normal* and *High*),
    - *Wind* (with values *Strong* and *Weak*),
    - *Water* (with values *Warm* and *Cool*), and
    - *Forecast* (with values *Same* and *Change*).
  - Hypotheses H: Each hypothesis is described by a conjunction of constraints on the attributes *Sky*, *AirTemp*, *Humidity*, *Wind*, *Water*, and *Forecast*. The constraints may be "?" (any value is acceptable), "∅" (no value is acceptable), or a specific value.
  - Target concept c: $EnjoySport : X \rightarrow \{0, 1\}$
  - Training examples D: Positive and negative examples of the target function (see Table 2.1).
- **Determine:**
  - A hypothesis h in H such that $h(x) = c(x)$ for all x in X.

TABLE 2.2
The *EnjoySport* concept learning task.

When learning the target concept, the learner is presented a set of *training examples*, each consisting of an instance x from X, along with its target concept value $c(x)$ (e.g., the training examples in Table 2.1). Instances for which $c(x) = 1$ are called *positive examples*, or members of the target concept. Instances for which $c(x) = 0$ are called *negative examples*, or nonmembers of the target concept. We will often write the ordered pair $\langle x, c(x) \rangle$ to describe the training example consisting of the instance x and its target concept value $c(x)$. We use the symbol D to denote the set of available training examples.

Given a set of training examples of the target concept c, the problem faced by the learner is to hypothesize, or estimate, c. We use the symbol H to denote the set of *all possible hypotheses* that the learner may consider regarding the identity of the target concept. Usually H is determined by the human designer's choice of hypothesis representation. In general, each hypothesis h in H represents a boolean-valued function defined over X; that is, $h : X \rightarrow \{0, 1\}$. The goal of the learner is to find a hypothesis h such that $h(x) = c(x)$ for a'l x in X.

2.2.2 The Inductive Learning Hypothesis

Notice that although the learning task is to determine a hypothesis h identical to the target concept c over the entire set of instances X, the only information available about c is its value over the training examples. Therefore, inductive learning algorithms can at best guarantee that the output hypothesis fits the target concept over the training data. Lacking any further information, our assumption is that the best hypothesis regarding unseen instances is the hypothesis that best fits the observed training data. This is the fundamental assumption of inductive learning, and we will have much more to say about it throughout this book. We state it here informally and will revisit and analyze this assumption more formally and more quantitatively in Chapters 5, 6, and 7.

> **The inductive learning hypothesis.** Any hypothesis found to approximate the target function well over a sufficiently large set of training examples will also approximate the target function well over other unobserved examples.

2.3 CONCEPT LEARNING AS SEARCH

Concept learning can be viewed as the task of searching through a large space of hypotheses implicitly defined by the hypothesis representation. The goal of this search is to find the hypothesis that best fits the training examples. It is important to note that by selecting a hypothesis representation, the designer of the learning algorithm implicitly defines the space of all hypotheses that the program can ever represent and therefore can ever learn. Consider, for example, the instances X and hypotheses H in the *EnjoySport* learning task. Given that the attribute *Sky* has three possible values, and that *AirTemp*, *Humidity*, *Wind*, *Water*, and *Forecast* each have two possible values, the instance space X contains exactly

$3 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 96$ distinct instances. A similar calculation shows that there are $5 \cdot 4 \cdot 4 \cdot 4 \cdot 4 \cdot 4 = 5120$ *syntactically distinct* hypotheses within H. Notice, however, that every hypothesis containing one or more "$\emptyset$" symbols represents the empty set of instances; that is, it classifies every instance as negative. Therefore, the number of *semantically distinct* hypotheses is only $1 + (4 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3) = 973$. Our *EnjoySport* example is a very simple learning task, with a relatively small, finite hypothesis space. Most practical learning tasks involve much larger, sometimes infinite, hypothesis spaces.

If we view learning as a search problem, then it is natural that our study of learning algorithms will examine different strategies for searching the hypothesis space. We will be particularly interested in algorithms capable of efficiently searching very large or infinite hypothesis spaces, to find the hypotheses that best fit the training data.

2.3.1 General-to-Specific Ordering of Hypotheses

Many algorithms for concept learning organize the search through the hypothesis space by relying on a very useful structure that exists for any concept learning problem: a general-to-specific ordering of hypotheses. By taking advantage of this naturally occurring structure over the hypothesis space, we can design learning algorithms that exhaustively search even infinite hypothesis spaces without explicitly enumerating every hypothesis. To illustrate the general-to-specific ordering, consider the two hypotheses

$$h_1 = \langle Sunny, ?, ?, Strong, ?, ? \rangle$$

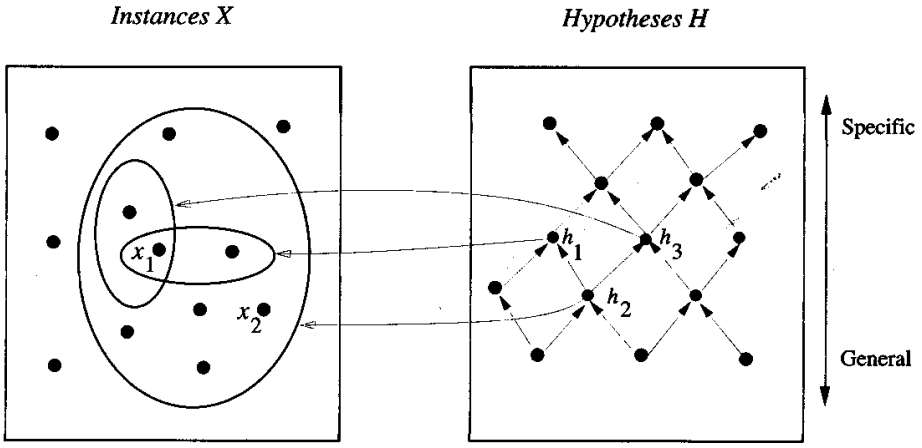
$$h_2 = \langle Sunny, ?, ?, ?, ?, ? \rangle$$

Now consider the sets of instances that are classified positive by h_1 and by h_2. Because h_2 imposes fewer constraints on the instance, it classifies more instances as positive. In fact, any instance classified positive by h_1 will also be classified positive by h_2. Therefore, we say that h_2 is more general than h_1.

This intuitive "more general than" relationship between hypotheses can be defined more precisely as follows. First, for any instance x in X and hypothesis h in H, we say that x *satisfies* h if and only if $h(x) = 1$. We now define the *more_general_than_or_equal_to* relation in terms of the sets of instances that satisfy the two hypotheses: Given hypotheses h_j and h_k, h_j is *more_general_than_or_equal_to* h_k if and only if any instance that satisfies h_k also satisfies h_j.

> ***Definition***: Let h_j and h_k be boolean-valued functions defined over X. Then h_j is **more_general_than_or_equal_to** h_k (written $h_j \geq_g h_k$) if and only if
>
> $$(\forall x \in X)[(h_k(x) = 1) \rightarrow (h_j(x) = 1)]$$

We will also find it useful to consider cases where one hypothesis is strictly more general than the other. Therefore, we will say that h_j is (strictly) *more_general_than*



x_1= <Sunny, Warm, High, Strong, Cool, Same>
x_2= <Sunny, Warm, High, Light, Warm, Same>

h_1= <Sunny, ?, ?, Strong, ?, ?>
h_2= <Sunny, ?, ?, ?, ?, ?>
h_3= <Sunny, ?, ?, ?, Cool, ?>

FIGURE 2.1
Instances, hypotheses, and the *more_general_than* relation. The box on the left represents the set X of all instances, the box on the right the set H of all hypotheses. Each hypothesis corresponds to some subset of X—the subset of instances that it classifies positive. The arrows connecting hypotheses represent the *more_general_than* relation, with the arrow pointing toward the less general hypothesis. Note the subset of instances characterized by h_2 subsumes the subset characterized by h_1, hence h_2 is *more_general_than* h_1.

h_k (written $h_j >_g h_k$) if and only if $(h_j \geq_g h_k) \wedge (h_k \not\geq_g h_j)$. Finally, we will sometimes find the inverse useful and will say that h_j is *more_specific_than* h_k when h_k is *more_general_than* h_j.

To illustrate these definitions, consider the three hypotheses h_1, h_2, and h_3 from our *EnjoySport* example, shown in Figure 2.1. How are these three hypotheses related by the $\geq_g$ relation? As noted earlier, hypothesis h_2 is more general than h_1 because every instance that satisfies h_1 also satisfies h_2. Similarly, h_2 is more general than h_3. Note that neither h_1 nor h_3 is more general than the other; although the instances satisfied by these two hypotheses intersect, neither set subsumes the other. Notice also that the $\geq_g$ and $>_g$ relations are defined independent of the target concept. They depend only on which instances satisfy the two hypotheses and not on the classification of those instances according to the target concept. Formally, the $\geq_g$ relation defines a partial order over the hypothesis space H (the relation is reflexive, antisymmetric, and transitive). Informally, when we say the structure is a partial (as opposed to total) order, we mean there may be pairs of hypotheses such as h_1 and h_3, such that $h_1 \not\geq_g h_3$ and $h_3 \not\geq_g h_1$.

The $\geq_g$ relation is important because it provides a useful structure over the hypothesis space H for *any* concept learning problem. The following sections present concept learning algorithms that take advantage of this partial order to efficiently organize the search for hypotheses that fit the training data.

1. Initialize h to the most specific hypothesis in H
2. For each positive training instance x
   - For each attribute constraint a_i in h
     If the constraint a_i is satisfied by x
     Then do nothing
     Else replace a_i in h by the next more general constraint that is satisfied by x
3. Output hypothesis h

TABLE 2.3
FIND-S Algorithm.

2.4 FIND-S: FINDING A MAXIMALLY SPECIFIC HYPOTHESIS

How can we use the *more_general_than* partial ordering to organize the search for a hypothesis consistent with the observed training examples? One way is to begin with the most specific possible hypothesis in H, then generalize this hypothesis each time it fails to cover an observed positive training example. (We say that a hypothesis "covers" a positive example if it correctly classifies the example as positive.) To be more precise about how the partial ordering is used, consider the FIND-S algorithm defined in Table 2.3.

To illustrate this algorithm, assume the learner is given the sequence of training examples from Table 2.1 for the *EnjoySport* task. The first step of FIND-S is to initialize h to the most specific hypothesis in H

$$h \leftarrow \langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$$

Upon observing the first training example from Table 2.1, which happens to be a positive example, it becomes clear that our hypothesis is too specific. In particular, none of the "$\emptyset$" constraints in h are satisfied by this example, so each is replaced by the next more general constraint that fits the example; namely, the attribute values for this training example.

$$h \leftarrow \langle Sunny, Warm, Normal, Strong, Warm, Same \rangle$$

This h is still very specific; it asserts that all instances are negative except for the single positive training example we have observed. Next, the second training example (also positive in this case) forces the algorithm to further generalize h, this time substituting a "?" in place of any attribute value in h that is not satisfied by the new example. The refined hypothesis in this case is

$$h \leftarrow \langle Sunny, Warm, ?, Strong, Warm, Same \rangle$$

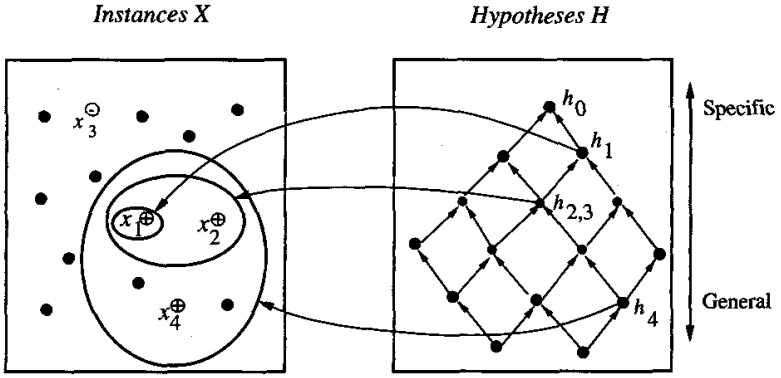
Upon encountering the third training example—in this case a negative example—the algorithm makes no change to h. In fact, the FIND-S algorithm simply *ignores every negative example*! While this may at first seem strange, notice that in the current case our hypothesis h is already consistent with the new negative example (i.e., h correctly classifies this example as negative), and hence no revision

is needed. In the general case, as long as we assume that the hypothesis space H contains a hypothesis that describes the true target concept c and that the training data contains no errors, then the current hypothesis h can never require a revision in response to a negative example. To see why, recall that the current hypothesis h is the most specific hypothesis in H consistent with the observed positive examples. Because the target concept c is also assumed to be in H and to be consistent with the positive training examples, c must be *more_general_than_or_equal_to* h. But the target concept c will never cover a negative example, thus neither will h (by the definition of *more_general_than*). Therefore, no revision to h will be required in response to any negative example.

To complete our trace of FIND-S, the fourth (positive) example leads to a further generalization of h

$$h \leftarrow \langle Sunny, Warm, ?, Strong, ?, ? \rangle$$

The FIND-S algorithm illustrates one way in which the *more_general_than* partial ordering can be used to organize the search for an acceptable hypothesis. The search moves from hypothesis to hypothesis, searching from the most specific to progressively more general hypotheses along one chain of the partial ordering. Figure 2.2 illustrates this search in terms of the instance and hypothesis spaces. At each step, the hypothesis is generalized only as far as necessary to cover the new positive example. Therefore, at each stage the hypothesis is the most specific hypothesis consistent with the training examples observed up to this point (hence the name FIND-S). The literature on concept learning is



x_1 = <Sunny Warm Normal Strong Warm Same>, +
x_2 = <Sunny Warm High Strong Warm Same>, +
x_3 = <Rainy Cold High Strong Warm Change>, -
x_4 = <Sunny Warm High Strong Cool Change>, +

h_0 = <∅, ∅, ∅, ∅, ∅, ∅>
h_1 = <Sunny Warm Normal Strong Warm Same>
h_2 = <Sunny Warm ? Strong Warm Same>
h_3 = <Sunny Warm ? Strong Warm Same>
h_4 = <Sunny Warm ? Strong ? ? >

FIGURE 2.2
The hypothesis space search performed by FIND-S. The search begins (h_0) with the most specific hypothesis in H, then considers increasingly general hypotheses (h_1 through h_4) as mandated by the training examples. In the instance space diagram, positive training examples are denoted by "+," negative by "–," and instances that have not been presented as training examples are denoted by a solid circle.

populated by many different algorithms that utilize this same *more_general_than* partial ordering to organize the search in one fashion or another. A number of such algorithms are discussed in this chapter, and several others are presented in Chapter 10.

The key property of the FIND-S algorithm is that for hypothesis spaces described by conjunctions of attribute constraints (such as H for the *EnjoySport* task), FIND-S is guaranteed to output the most specific hypothesis within H that is consistent with the positive training examples. Its final hypothesis will also be consistent with the negative examples provided the correct target concept is contained in H, and provided the training examples are correct. However, there are several questions still left unanswered by this learning algorithm, such as:

- Has the learner converged to the correct target concept? Although FIND-S will find a hypothesis consistent with the training data, it has no way to determine whether it has found the *only* hypothesis in H consistent with the data (i.e., the correct target concept), or whether there are many other consistent hypotheses as well. We would prefer a learning algorithm that could determine whether it had converged and, if not, at least characterize its uncertainty regarding the true identity of the target concept.

- Why prefer the most specific hypothesis? In case there are multiple hypotheses consistent with the training examples, FIND-S will find the most specific. It is unclear whether we should prefer this hypothesis over, say, the most general, or some other hypothesis of intermediate generality.

- Are the training examples consistent? In most practical learning problems there is some chance that the training examples will contain at least some errors or noise. Such inconsistent sets of training examples can severely mislead FIND-S, given the fact that it ignores negative examples. We would prefer an algorithm that could at least detect when the training data is inconsistent and, preferably, accommodate such errors.

- What if there are several maximally specific consistent hypotheses? In the hypothesis language H for the *EnjoySport* task, there is always a unique, most specific hypothesis consistent with any set of positive examples. However, for other hypothesis spaces (discussed later) there can be several maximally specific hypotheses consistent with the data. In this case, FIND-S must be extended to allow it to backtrack on its choices of how to generalize the hypothesis, to accommodate the possibility that the target concept lies along a different branch of the partial ordering than the branch it has selected. Furthermore, we can define hypothesis spaces for which there is no maximally specific consistent hypothesis, although this is more of a theoretical issue than a practical one (see Exercise 2.7).

2.5 VERSION SPACES AND THE CANDIDATE-ELIMINATION ALGORITHM

This section describes a second approach to concept learning, the CANDIDATE-ELIMINATION algorithm, that addresses several of the limitations of FIND-S. Notice that although FIND-S outputs a hypothesis from H, that is consistent with the training examples, this is just one of many hypotheses from H that might fit the training data equally well. The key idea in the CANDIDATE-ELIMINATION algorithm is to output a description of the set of *all hypotheses consistent with the training examples.* Surprisingly, the CANDIDATE-ELIMINATION algorithm computes the description of this set without explicitly enumerating all of its members. This is accomplished by again using the *more_general_than* partial ordering, this time to maintain a compact representation of the set of consistent hypotheses and to incrementally refine this representation as each new training example is encountered.

The CANDIDATE-ELIMINATION algorithm has been applied to problems such as learning regularities in chemical mass spectroscopy (Mitchell 1979) and learning control rules for heuristic search (Mitchell et al. 1983). Nevertheless, practical applications of the CANDIDATE-ELIMINATION and FIND-S algorithms are limited by the fact that they both perform poorly when given noisy training data. More importantly for our purposes here, the CANDIDATE-ELIMINATION algorithm provides a useful conceptual framework for introducing several fundamental issues in machine learning. In the remainder of this chapter we present the algorithm and discuss these issues. Beginning with the next chapter, we will examine learning algorithms that are used more frequently with noisy training data.

2.5.1 Representation

The CANDIDATE-ELIMINATION algorithm finds all describable hypotheses that are consistent with the observed training examples. In order to define this algorithm precisely, we begin with a few basic definitions. First, let us say that a hypothesis is *consistent* with the training examples if it correctly classifies these examples.

> ***Definition***: A hypothesis h is **consistent** with a set of training examples D if and only if $h(x) = c(x)$ for each example $\langle x, c(x)\rangle$ in D.
>
> $$Consistent(h, D) \equiv (\forall \langle x, c(x)\rangle \in D)\ h(x) = c(x)$$

Notice the key difference between this definition of *consistent* and our earlier definition of *satisfies*. An example x is said to *satisfy* hypothesis h when $h(x) = 1$, regardless of whether x is a positive or negative example of the target concept. However, whether such an example is *consistent* with h depends on the target concept, and in particular, whether $h(x) = c(x)$.

The CANDIDATE-ELIMINATION algorithm represents the set of *all* hypotheses consistent with the observed training examples. This subset of all hypotheses is

called the *version space* with respect to the hypothesis space H and the training examples D, because it contains all plausible versions of the target concept.

> ***Definition***: The **version space**, denoted $VS_{H,D}$, with respect to hypothesis space H and training examples D, is the subset of hypotheses from H consistent with the training examples in D.
>
> $$VS_{H,D} \equiv \{h \in H | Consistent(h, D)\}$$

2.5.2 The LIST-THEN-ELIMINATE Algorithm

One obvious way to represent the version space is simply to list all of its members. This leads to a simple learning algorithm, which we might call the LIST-THEN-ELIMINATE algorithm, defined in Table 2.4.

The LIST-THEN-ELIMINATE algorithm first initializes the version space to contain all hypotheses in H, then eliminates any hypothesis found inconsistent with any training example. The version space of candidate hypotheses thus shrinks as more examples are observed, until ideally just one hypothesis remains that is consistent with all the observed examples. This, presumably, is the desired target concept. If insufficient data is available to narrow the version space to a single hypothesis, then the algorithm can output the entire set of hypotheses consistent with the observed data.

In principle, the LIST-THEN-ELIMINATE algorithm can be applied whenever the hypothesis space H is finite. It has many advantages, including the fact that it is guaranteed to output all hypotheses consistent with the training data. Unfortunately, it requires exhaustively enumerating all hypotheses in H—an unrealistic requirement for all but the most trivial hypothesis spaces.

2.5.3 A More Compact Representation for Version Spaces

The CANDIDATE-ELIMINATION algorithm works on the same principle as the above LIST-THEN-ELIMINATE algorithm. However, it employs a much more compact representation of the version space. In particular, the version space is represented by its most general and least general members. These members form general and specific boundary sets that delimit the version space within the partially ordered hypothesis space.

The LIST-THEN-ELIMINATE Algorithm

1. $VersionSpace \leftarrow$ a list containing every hypothesis in H
2. For each training example, $\langle x, c(x) \rangle$
   remove from $VersionSpace$ any hypothesis h for which $h(x) \neq c(x)$
3. Output the list of hypotheses in $VersionSpace$

TABLE 2.4
The LIST-THEN-ELIMINATE algorithm.

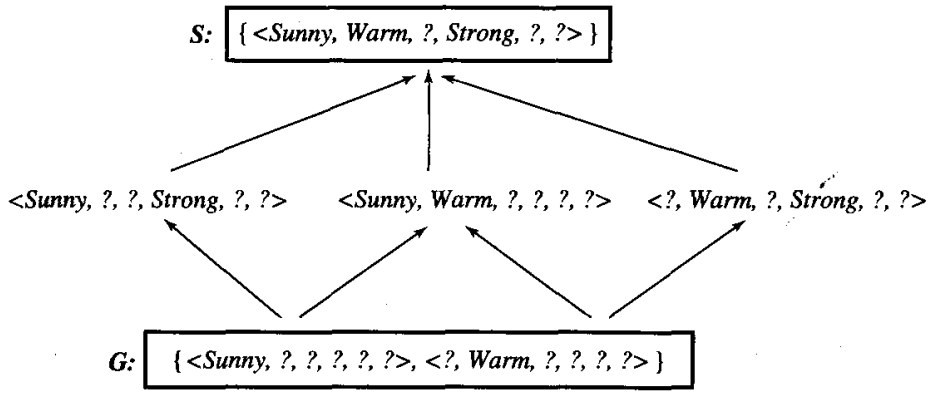


FIGURE 2.3
A version space with its general and specific boundary sets. The version space includes all six hypotheses shown here, but can be represented more simply by S and G. Arrows indicate instances of the *more_general_than* relation. This is the version space for the *EnjoySport* concept learning problem and training examples described in Table 2.1.

To illustrate this representation for version spaces, consider again the *EnjoySport* concept learning problem described in Table 2.2. Recall that given the four training examples from Table 2.1, FIND-S outputs the hypothesis

$$h = \langle Sunny, Warm, ?, Strong, ?, ? \rangle$$

In fact, this is just one of six different hypotheses from H that are consistent with these training examples. All six hypotheses are shown in Figure 2.3. They constitute the version space relative to this set of data and this hypothesis representation. The arrows among these six hypotheses in Figure 2.3 indicate instances of the *more_general_than* relation. The CANDIDATE-ELIMINATION algorithm represents the version space by storing only its most general members (labeled G in Figure 2.3) and its most specific (labeled S in the figure). Given only these two sets S and G, it is possible to enumerate all members of the version space as needed by generating the hypotheses that lie between these two sets in the general-to-specific partial ordering over hypotheses.

It is intuitively plausible that we can represent the version space in terms of its most specific and most general members. Below we define the boundary sets G and S precisely and prove that these sets do in fact represent the version space.

Definition: The **general boundary** G, with respect to hypothesis space H and training data D, is the set of maximally general members of H consistent with D.

$$G \equiv \{g \in H | Consistent(g, D) \wedge (\neg \exists g' \in H)[(g' >_g g) \wedge Consistent(g', D)]\}$$

Definition: The **specific boundary** S, with respect to hypothesis space H and training data D, is the set of minimally general (i.e., maximally specific) members of H consistent with D.

$$S \equiv \{s \in H | Consistent(s, D) \wedge (\neg \exists s' \in H)[(s >_g s') \wedge Consistent(s', D)]\}$$

As long as the sets G and S are well defined (see Exercise 2.7), they completely specify the version space. In particular, we can show that the version space is precisely the set of hypotheses contained in G, plus those contained in S, plus those that lie between G and S in the partially ordered hypothesis space. This is stated precisely in Theorem 2.1.

Theorem 2.1. Version space representation theorem. Let X be an arbitrary set of instances and let H be a set of boolean-valued hypotheses defined over X. Let $c : X \rightarrow \{0, 1\}$ be an arbitrary target concept defined over X, and let D be an arbitrary set of training examples $\{\langle x, c(x)\rangle\}$. For all X, H, c, and D such that S and G are well defined,

$$VS_{H,D} = \{h \in H | (\exists s \in S)(\exists g \in G)(g \geq_g h \geq_g s)\}$$

Proof. To prove the theorem it suffices to show that (1) every h satisfying the right-hand side of the above expression is in $VS_{H,D}$ and (2) every member of $VS_{H,D}$ satisfies the right-hand side of the expression. To show (1) let g be an arbitrary member of G, s be an arbitrary member of S, and h be an arbitrary member of H, such that $g \geq_g h \geq_g s$. Then by the definition of S, s must be satisfied by all positive examples in D. Because $h \geq_g s$, h must also be satisfied by all positive examples in D. Similarly, by the definition of G, g cannot be satisfied by any negative example in D, and because $g \geq_g h$, h cannot be satisfied by any negative example in D. Because h is satisfied by all positive examples in D and by no negative examples in D, h is consistent with D, and therefore h is a member of $VS_{H,D}$. This proves step (1). The argument for (2) is a bit more complex. It can be proven by assuming some h in $VS_{H,D}$ that does not satisfy the right-hand side of the expression, then showing that this leads to an inconsistency. (See Exercise 2.6.) □

2.5.4 CANDIDATE-ELIMINATION Learning Algorithm

The CANDIDATE-ELIMINATION algorithm computes the version space containing all hypotheses from H that are consistent with an observed sequence of training examples. It begins by initializing the version space to the set of all hypotheses in H; that is, by initializing the G boundary set to contain the most general hypothesis in H

$$G_0 \leftarrow \{\langle ?, ?, ?, ?, ?, ?\rangle\}$$

and initializing the S boundary set to contain the most specific (least general) hypothesis

$$S_0 \leftarrow \{\langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset\rangle\}$$

These two boundary sets delimit the entire hypothesis space, because every other hypothesis in H is both more general than S_0 and more specific than G_0. As each training example is considered, the S and G boundary sets are generalized and specialized, respectively, to eliminate from the version space any hypotheses found inconsistent with the new training example. After all examples have been processed, the computed version space contains all the hypotheses consistent with these examples and only these hypotheses. This algorithm is summarized in Table 2.5.

Initialize G to the set of maximally general hypotheses in H
Initialize S to the set of maximally specific hypotheses in H
For each training example d, do

- If d is a positive example
  - Remove from G any hypothesis inconsistent with d
  - For each hypothesis s in S that is not consistent with d
    - Remove s from S
    - Add to S all minimal generalizations h of s such that
      - h is consistent with d, and some member of G is more general than h
    - Remove from S any hypothesis that is more general than another hypothesis in S
- If d is a negative example
  - Remove from S any hypothesis inconsistent with d
  - For each hypothesis g in G that is not consistent with d
    - Remove g from G
    - Add to G all minimal specializations h of g such that
      - h is consistent with d, and some member of S is more specific than h
    - Remove from G any hypothesis that is less general than another hypothesis in G

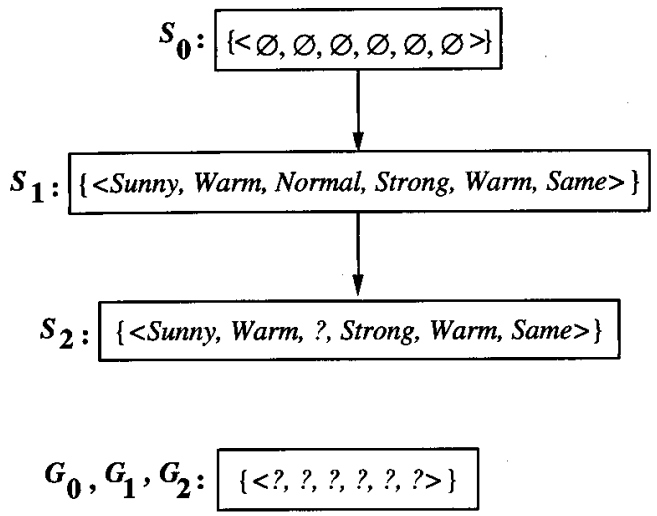
TABLE 2.5
CANDIDATE-ELIMINATION algorithm using version spaces. Notice the duality in how positive and negative examples influence S and G.

Notice that the algorithm is specified in terms of operations such as computing minimal generalizations and specializations of given hypotheses, and identifying nonminimal and nonmaximal hypotheses. The detailed implementation of these operations will depend, of course, on the specific representations for instances and hypotheses. However, the algorithm itself can be applied to any concept learning task and hypothesis space for which these operations are well-defined. In the following example trace of this algorithm, we see how such operations can be implemented for the representations used in the *EnjoySport* example problem.

2.5.5 An Illustrative Example

Figure 2.4 traces the CANDIDATE-ELIMINATION algorithm applied to the first two training examples from Table 2.1. As described above, the boundary sets are first initialized to G_0 and S_0, the most general and most specific hypotheses in H, respectively.

When the first training example is presented (a positive example in this case), the CANDIDATE-ELIMINATION algorithm checks the S boundary and finds that it is overly specific—it fails to cover the positive example. The boundary is therefore revised by moving it to the least more general hypothesis that covers this new example. This revised boundary is shown as S_1 in Figure 2.4. No update of the G boundary is needed in response to this training example because G_0 correctly covers this example. When the second training example (also positive) is observed, it has a similar effect of generalizing S further to S_2, leaving G again unchanged (i.e., $G_2 = G_1 = G_0$). Notice the processing of these first



Training examples:

1. *<Sunny, Warm, Normal, Strong, Warm, Same>, Enjoy Sport = Yes*

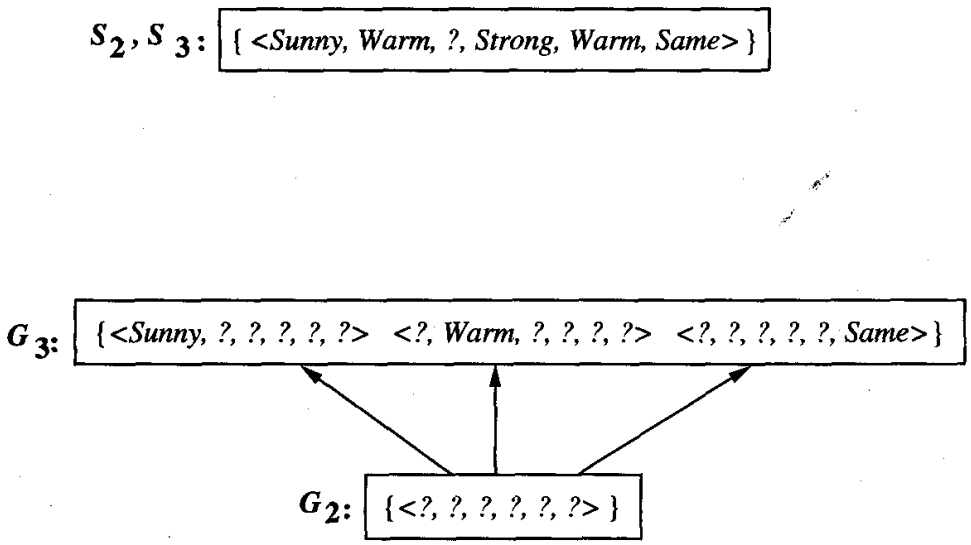
2. *<Sunny, Warm, High, Strong, Warm, Same>, Enjoy Sport = Yes*

FIGURE 2.4
CANDIDATE-ELIMINATION Trace 1. S_0 and G_0 are the initial boundary sets corresponding to the most specific and most general hypotheses. Training examples 1 and 2 force the S boundary to become more general, as in the FIND-S algorithm. They have no effect on the G boundary.

two positive examples is very similar to the processing performed by the FIND-S algorithm.

As illustrated by these first two steps, positive training examples may force the S boundary of the version space to become increasingly general. Negative training examples play the complimentary role of forcing the G boundary to become increasingly specific. Consider the third training example, shown in Figure 2.5. This negative example reveals that the G boundary of the version space is overly general; that is, the hypothesis in G incorrectly predicts that this new example is a positive example. The hypothesis in the G boundary must therefore be specialized until it correctly classifies this new negative example. As shown in Figure 2.5, there are several alternative minimally more specific hypotheses. All of these become members of the new G_3 boundary set.

Given that there are six attributes that could be specified to specialize G_2, why are there only three new hypotheses in G_3? For example, the hypothesis $h = \langle ?, ?, Normal, ?, ?, ? \rangle$ is a minimal specialization of G_2 that correctly labels the new example as a negative example, but it is not included in G_3. The reason this hypothesis is excluded is that it is inconsistent with the previously encountered positive examples. The algorithm determines this simply by noting that h is not more general than the current specific boundary, S_2. In fact, the S boundary of the version space forms a summary of the previously encountered positive examples that can be used to determine whether any given hypothesis



Training Example:

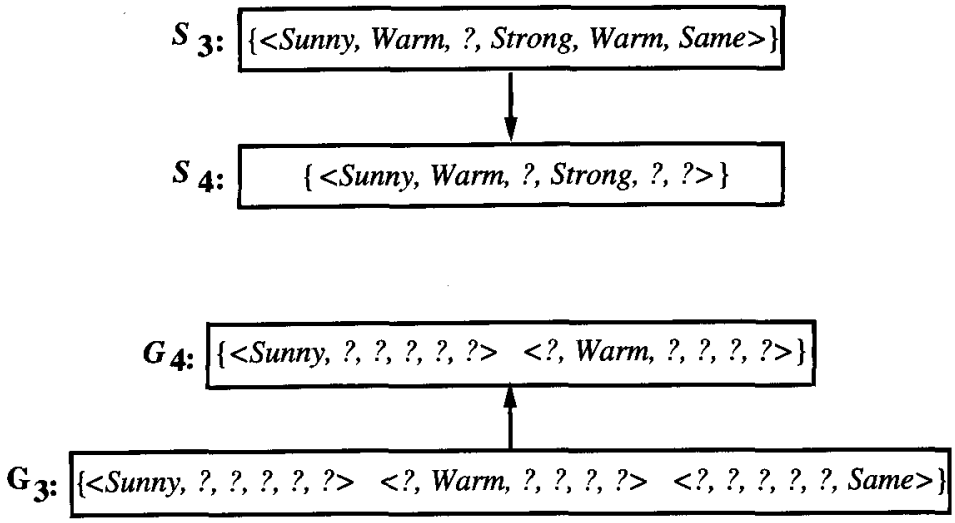
3. *<Rainy, Cold, High, Strong, Warm, Change>, EnjoySport=No*

FIGURE 2.5
CANDIDATE-ELIMINATION Trace 2. Training example 3 is a negative example that forces the G_2 boundary to be specialized to G_3. Note several alternative maximally general hypotheses are included in G_3.

is consistent with these examples. Any hypothesis more general than S will, by definition, cover any example that S covers and thus will cover any past positive example. In a dual fashion, the G boundary summarizes the information from previously encountered negative examples. Any hypothesis more specific than G is assured to be consistent with past negative examples. This is true because any such hypothesis, by definition, cannot cover examples that G does not cover.

The fourth training example, as shown in Figure 2.6, further generalizes the S boundary of the version space. It also results in removing one member of the G boundary, because this member fails to cover the new positive example. This last action results from the first step under the condition "If d is a positive example" in the algorithm shown in Table 2.5. To understand the rationale for this step, it is useful to consider why the offending hypothesis must be removed from G. Notice it cannot be specialized, because specializing it would not make it cover the new example. It also cannot be generalized, because by the definition of G, any more general hypothesis will cover at least one negative training example. Therefore, the hypothesis must be dropped from the G boundary, thereby removing an entire branch of the partial ordering from the version space of hypotheses remaining under consideration.

After processing these four examples, the boundary sets S_4 and G_4 delimit the version space of *all* hypotheses consistent with the set of incrementally observed training examples. The entire version space, including those hypotheses



Training Example:

4. *<Sunny, Warm, High, Strong, Cool, Change>, EnjoySport = Yes*

FIGURE 2.6
CANDIDATE-ELIMINATION Trace 3. The positive training example generalizes the S boundary, from S_3 to S_4. One member of G_3 must also be deleted, because it is no longer more general than the S_4 boundary.

bounded by S_4 and G_4, is shown in Figure 2.7. This learned version space is independent of the sequence in which the training examples are presented (because in the end it contains all hypotheses consistent with the set of examples). As further training data is encountered, the S and G boundaries will move monotonically closer to each other, delimiting a smaller and smaller version space of candidate hypotheses.

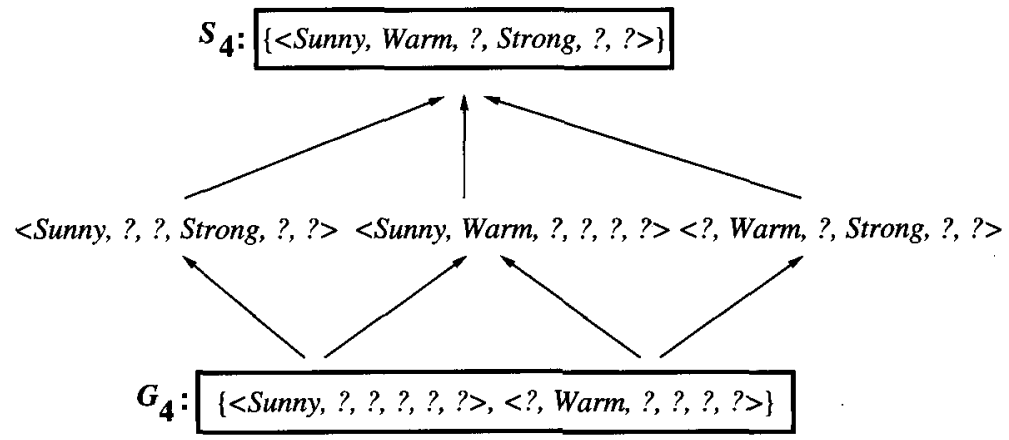


FIGURE 2.7
The final version space for the *EnjoySport* concept learning problem and training examples described earlier.

2.6 REMARKS ON VERSION SPACES AND CANDIDATE-ELIMINATION

2.6.1 Will the CANDIDATE-ELIMINATION Algorithm Converge to the Correct Hypothesis?

The version space learned by the CANDIDATE-ELIMINATION algorithm will converge toward the hypothesis that correctly describes the target concept, provided (1) there are no errors in the training examples, and (2) there is some hypothesis in H that correctly describes the target concept. In fact, as new training examples are observed, the version space can be monitored to determine the remaining ambiguity regarding the true target concept and to determine when sufficient training examples have been observed to unambiguously identify the target concept. The target concept is exactly learned when the S and G boundary sets converge to a single, identical, hypothesis.

What will happen if the training data contains errors? Suppose, for example, that the second training example above is incorrectly presented as a negative example instead of a positive example. Unfortunately, in this case the algorithm is certain to remove the correct target concept from the version space! Because it will remove every hypothesis that is inconsistent with each training example, it will eliminate the true target concept from the version space as soon as this false negative example is encountered. Of course, given sufficient additional training data the learner will eventually detect an inconsistency by noticing that the S and G boundary sets eventually converge to an empty version space. Such an empty version space indicates that there is *no* hypothesis in H consistent with all observed training examples. A similar symptom will appear when the training examples are correct, but the target concept cannot be described in the hypothesis representation (e.g., if the target concept is a disjunction of feature attributes and the hypothesis space supports only conjunctive descriptions). We will consider such eventualities in greater detail later. For now, we consider only the case in which the training examples are correct and the true target concept is present in the hypothesis space.

2.6.2 What Training Example Should the Learner Request Next?

Up to this point we have assumed that training examples are provided to the learner by some external teacher. Suppose instead that the learner is allowed to conduct experiments in which it chooses the next instance, then obtains the correct classification for this instance from an external oracle (e.g., nature or a teacher). This scenario covers situations in which the learner may conduct experiments in nature (e.g., build new bridges and allow nature to classify them as stable or unstable), or in which a teacher is available to provide the correct classification (e.g., propose a new bridge and allow the teacher to suggest whether or not it will be stable). We use the term *query* to refer to such instances constructed by the learner, which are then classified by an external oracle.

Consider again the version space learned from the four training examples of the *EnjoySport* concept and illustrated in Figure 2.3. What would be a good query for the learner to pose at this point? What is a good query strategy in

general? Clearly, the learner should attempt to discriminate among the alternative competing hypotheses in its current version space. Therefore, it should choose an instance that would be classified positive by some of these hypotheses, but negative by others. One such instance is

$$\langle Sunny, Warm, Normal, Light, Warm, Same \rangle$$

Note that this instance satisfies three of the six hypotheses in the current version space (Figure 2.3). If the trainer classifies this instance as a positive example, the S boundary of the version space can then be generalized. Alternatively, if the trainer indicates that this is a negative example, the G boundary can then be specialized. Either way, the learner will succeed in learning more about the true identity of the target concept, shrinking the version space from six hypotheses to half this number.

In general, the optimal query strategy for a concept learner is to generate instances that satisfy exactly half the hypotheses in the current version space. When this is possible, the size of the version space is reduced by half with each new example, and the correct target concept can therefore be found with only $\lceil log_2 |VS| \rceil$ experiments. The situation is analogous to playing the game twenty questions, in which the goal is to ask yes-no questions to determine the correct hypothesis. The optimal strategy for playing twenty questions is to ask questions that evenly split the candidate hypotheses into sets that predict yes and no. While we have seen that it is possible to generate an instance that satisfies precisely half the hypotheses in the version space of Figure 2.3, in general it may not be possible to construct an instance that matches precisely half the hypotheses. In such cases, a larger number of queries may be required than $\lceil log_2 |VS| \rceil$.

2.6.3 How Can Partially Learned Concepts Be Used?

Suppose that no additional training examples are available beyond the four in our example above, but that the learner is now required to classify new instances that it has not yet observed. Even though the version space of Figure 2.3 still contains multiple hypotheses, indicating that the target concept has not yet been fully learned, it is possible to classify certain examples with the same degree of confidence as if the target concept had been uniquely identified. To illustrate, suppose the learner is asked to classify the four new instances shown in Table 2.6.

Note that although instance A was not among the training examples, it is classified as a positive instance by *every* hypothesis in the current version space (shown in Figure 2.3). Because the hypotheses in the version space unanimously agree that this is a positive instance, the learner can classify instance A as positive with the same confidence it would have if it had already converged to the single, correct target concept. Regardless of which hypothesis in the version space is eventually found to be the correct target concept, it is already clear that it will classify instance A as a positive example. Notice furthermore that we need not enumerate every hypothesis in the version space in order to test whether each

Instance	*Sky*	*AirTemp*	*Humidity*	*Wind*	*Water*	*Forecast*	*EnjoySport*
A	Sunny	Warm	Normal	Strong	Cool	Change	?
B	Rainy	Cold	Normal	Light	Warm	Same	?
C	Sunny	Warm	Normal	Light	Warm	Same	?
D	Sunny	Cold	Normal	Strong	Warm	Same	?

TABLE 2.6
New instances to be classified.

classifies the instance as positive. This condition will be met if and only if the instance satisfies every member of S (why?). The reason is that every other hypothesis in the version space is at least as general as some member of S. By our definition of *more_general_than*, if the new instance satisfies all members of S it must also satisfy each of these more general hypotheses.

Similarly, instance B is classified as a negative instance by every hypothesis in the version space. This instance can therefore be safely classified as negative, given the partially learned concept. An efficient test for this condition is that the instance satisfies none of the members of G (why?).

Instance C presents a different situation. Half of the version space hypotheses classify it as positive and half classify it as negative. Thus, the learner cannot classify this example with confidence until further training examples are available. Notice that instance C is the same instance presented in the previous section as an optimal experimental query for the learner. This is to be expected, because those instances whose classification is most ambiguous are precisely the instances whose true classification would provide the most new information for refining the version space.

Finally, instance D is classified as positive by two of the version space hypotheses and negative by the other four hypotheses. In this case we have less confidence in the classification than in the unambiguous cases of instances A and B. Still, the vote is in favor of a negative classification, and one approach we could take would be to output the majority vote, perhaps with a confidence rating indicating how close the vote was. As we will discuss in Chapter 6, if we assume that all hypotheses in H are equally probable a priori, then such a vote provides the most probable classification of this new instance. Furthermore, the proportion of hypotheses voting positive can be interpreted as the probability that this instance is positive given the training data.

2.7 INDUCTIVE BIAS

As discussed above, the CANDIDATE-ELIMINATION algorithm will converge toward the true target concept provided it is given accurate training examples and provided its initial hypothesis space contains the target concept. What if the target concept is not contained in the hypothesis space? Can we avoid this difficulty by using a hypothesis space that includes every possible hypothesis? How does the

size of this hypothesis space influence the ability of the algorithm to generalize to unobserved instances? How does the size of the hypothesis space influence the number of training examples that must be observed? These are fundamental questions for inductive inference in general. Here we examine them in the context of the CANDIDATE-ELIMINATION algorithm. As we shall see, though, the conclusions we draw from this analysis will apply to *any* concept learning system that outputs *any* hypothesis consistent with the training data.

2.7.1 A Biased Hypothesis Space

Suppose we wish to assure that the hypothesis space contains the unknown target concept. The obvious solution is to enrich the hypothesis space to include *every possible* hypothesis. To illustrate, consider again the *EnjoySport* example in which we restricted the hypothesis space to include only conjunctions of attribute values. Because of this restriction, the hypothesis space is unable to represent even simple disjunctive target concepts such as "*Sky* = *Sunny* or *Sky* = *Cloudy*." In fact, given the following three training examples of this disjunctive hypothesis, our algorithm would find that there are zero hypotheses in the version space.

Example	*Sky*	*AirTemp*	*Humidity*	*Wind*	*Water*	*Forecast*	*EnjoySport*
1	Sunny	Warm	Normal	Strong	Cool	Change	Yes
2	Cloudy	Warm	Normal	Strong	Cool	Change	Yes
3	Rainy	Warm	Normal	Strong	Cool	Change	No

To see why there are no hypotheses consistent with these three examples, note that the most specific hypothesis consistent with the first two examples *and representable in the given hypothesis space H* is

$$S_2 : \langle ?, Warm, Normal, Strong, Cool, Change \rangle$$

This hypothesis, although it is the maximally specific hypothesis from H that is consistent with the first two examples, is already overly general: it incorrectly covers the third (negative) training example. The problem is that we have biased the learner to consider only conjunctive hypotheses. In this case we require a more expressive hypothesis space.

2.7.2 An Unbiased Learner

The obvious solution to the problem of assuring that the target concept is in the hypothesis space H is to provide a hypothesis space capable of representing *every teachable concept*; that is, it is capable of representing every possible subset of the instances X. In general, the set of all subsets of a set X is called the *power set* of X.

In the *EnjoySport* learning task, for example, the size of the instance space X of days described by the six available attributes is 96. How many possible concepts can be defined over this set of instances? In other words, how large is

the power set of X? In general, the number of distinct subsets that can be defined over a set X containing $|X|$ elements (i.e., the size of the power set of X) is $2^{|X|}$. Thus, there are 2^{96}, or approximately 10^{28} distinct target concepts that could be defined over this instance space and that our learner might be called upon to learn. Recall from Section 2.3 that our conjunctive hypothesis space is able to represent only 973 of these—a very biased hypothesis space indeed!

Let us reformulate the *EnjoySport* learning task in an unbiased way by defining a new hypothesis space H' that can represent every subset of instances; that is, let H' correspond to the power set of X. One way to define such an H' is to allow arbitrary disjunctions, conjunctions, and negations of our earlier hypotheses. For instance, the target concept "*Sky* = *Sunny* or *Sky* = *Cloudy*" could then be described as

$$\langle Sunny, ?, ?, ?, ?, ? \rangle \vee \langle Cloudy, ?, ?, ?, ?, ? \rangle$$

Given this hypothesis space, we can safely use the CANDIDATE-ELIMINATION algorithm without worrying that the target concept might not be expressible. However, while this hypothesis space eliminates any problems of expressibility, it unfortunately raises a new, equally difficult problem: our concept learning algorithm is now completely unable to generalize beyond the observed examples! To see why, suppose we present three positive examples (x_1, x_2, x_3) and two negative examples (x_4, x_5) to the learner. At this point, the S boundary of the version space will contain the hypothesis which is just the disjunction of the positive examples

$$S : \{(x_1 \vee x_2 \vee x_3)\}$$

because this is the most specific possible hypothesis that covers these three examples. Similarly, the G boundary will consist of the hypothesis that rules out only the observed negative examples

$$G : \{\neg(x_4 \vee x_5)\}$$

The problem here is that with this very expressive hypothesis representation, the S boundary will always be simply the disjunction of the observed positive examples, while the G boundary will always be the negated disjunction of the observed negative examples. Therefore, the only examples that will be unambiguously classified by S and G are the observed training examples themselves. In order to converge to a single, final target concept, we will have to present every single instance in X as a training example!

It might at first seem that we could avoid this difficulty by simply using the partially learned version space and by taking a vote among the members of the version space as discussed in Section 2.6.3. Unfortunately, the only instances that will produce a unanimous vote are the previously observed training examples. For all the other instances, taking a vote will be futile: each unobserved instance will be classified positive by *precisely half* the hypotheses in the version space and will be classified negative by the other half (why?). To see the reason, note that when H is the power set of X and x is some previously unobserved instance, then for any hypothesis h in the version space that covers x, there will be another

hypothesis h' in the power set that is identical to h except for its classification of x. And of course if h is in the version space, then h' will be as well, because it agrees with h on all the observed training examples.

2.7.3 The Futility of Bias-Free Learning

The above discussion illustrates a fundamental property of inductive inference: *a learner that makes no a priori assumptions regarding the identity of the target concept has no rational basis for classifying any unseen instances.* In fact, the only reason that the CANDIDATE-ELIMINATION algorithm was able to generalize beyond the observed training examples in our original formulation of the *EnjoySport* task is that it was biased by the implicit assumption that the target concept could be represented by a conjunction of attribute values. In cases where this assumption is correct (and the training examples are error-free), its classification of new instances will also be correct. If this assumption is incorrect, however, it is certain that the CANDIDATE-ELIMINATION algorithm will misclassify at least some instances from X.

Because inductive learning requires some form of prior assumptions, or inductive bias, we will find it useful to characterize different learning approaches by the inductive bias† they employ. Let us define this notion of inductive bias more precisely. The key idea we wish to capture here is the policy by which the learner generalizes beyond the observed training data, to infer the classification of new instances. Therefore, consider the general setting in which an arbitrary learning algorithm L is provided an arbitrary set of training data $D_c = \{\langle x, c(x)\rangle\}$ of some arbitrary target concept c. After training, L is asked to classify a new instance x_i. Let $L(x_i, D_c)$ denote the classification (e.g., positive or negative) that L assigns to x_i after learning from the training data D_c. We can describe this inductive inference step performed by L as follows

$$(D_c \wedge x_i) \succ L(x_i, D_c)$$

where the notation $y \succ z$ indicates that z is inductively inferred from y. For example, if we take L to be the CANDIDATE-ELIMINATION algorithm, D_c to be the training data from Table 2.1, and x_i to be the first instance from Table 2.6, then the inductive inference performed in this case concludes that $L(x_i, D_c) = (EnjoySport = yes)$.

Because L is an inductive learning algorithm, the result $L(x_i, D_c)$ that it infers will not in general be provably correct; that is, the classification $L(x_i, D_c)$ need not follow deductively from the training data D_c and the description of the new instance x_i. However, it is interesting to ask what additional assumptions could be added to $D_c \wedge x_i$ so that $L(x_i, D_c)$ would follow deductively. We define the inductive bias of L as this set of additional assumptions. More precisely, we define the

†The term *inductive bias* here is not to be confused with the term *estimation bias* commonly used in statistics. Estimation bias will be discussed in Chapter 5.

inductive bias of L to be the set of assumptions B such that for all new instances x_i

$$(B \wedge D_c \wedge x_i) \vdash L(x_i, D_c)$$

where the notation $y \vdash z$ indicates that z follows deductively from y (i.e., that z is provable from y). Thus, we define the inductive bias of a learner as the set of additional assumptions B sufficient to justify its inductive inferences as deductive inferences. To summarize,

> ***Definition***: Consider a concept learning algorithm L for the set of instances X. Let c be an arbitrary concept defined over X, and let $D_c = \{\langle x, c(x)\rangle\}$ be an arbitrary set of training examples of c. Let $L(x_i, D_c)$ denote the classification assigned to the instance x_i by L after training on the data D_c. The **inductive bias** of L is any minimal set of assertions B such that for any target concept c and corresponding training examples D_c
>
> $$(\forall x_i \in X)[(B \wedge D_c \wedge x_i) \vdash L(x_i, D_c)] \tag{2.1}$$

What, then, is the inductive bias of the CANDIDATE-ELIMINATION algorithm? To answer this, let us specify $L(x_i, D_c)$ exactly for this algorithm: given a set of data D_c, the CANDIDATE-ELIMINATION algorithm will first compute the version space VS_{H,D_c}, then classify the new instance x_i by a vote among hypotheses in this version space. Here let us assume that it will output a classification for x_i only if this vote among version space hypotheses is unanimously positive or negative and that it will not output a classification otherwise. Given this definition of $L(x_i, D_c)$ for the CANDIDATE-ELIMINATION algorithm, what is its inductive bias? It is simply the assumption $c \in H$. Given this assumption, each inductive inference performed by the CANDIDATE-ELIMINATION algorithm can be justified deductively.

To see why the classification $L(x_i, D_c)$ follows deductively from $B = \{c \in H\}$, together with the data D_c and description of the instance x_i, consider the following argument. First, notice that if we assume $c \in H$ then it follows deductively that $c \in VS_{H,D_c}$. This follows from $c \in H$, from the definition of the version space VS_{H,D_c} as the set of all hypotheses in H that are consistent with D_c, and from our definition of $D_c = \{\langle x, c(x)\rangle\}$ as training data consistent with the target concept c. Second, recall that we defined the classification $L(x_i, D_c)$ to be the unanimous vote of all hypotheses in the version space. Thus, if L outputs the classification $L(x_i, D_c)$, it must be the case the every hypothesis in VS_{H,D_c} also produces this classification, including the hypothesis $c \in VS_{H,D_c}$. Therefore $c(x_i) = L(x_i, D_c)$. To summarize, the CANDIDATE-ELIMINATION algorithm defined in this fashion can be characterized by the following bias

> **Inductive bias of CANDIDATE-ELIMINATION algorithm.** The target concept c is contained in the given hypothesis space H.

Figure 2.8 summarizes the situation schematically. The inductive CANDIDATE-ELIMINATION algorithm at the top of the figure takes two inputs: the training examples and a new instance to be classified. At the bottom of the figure, a deductive

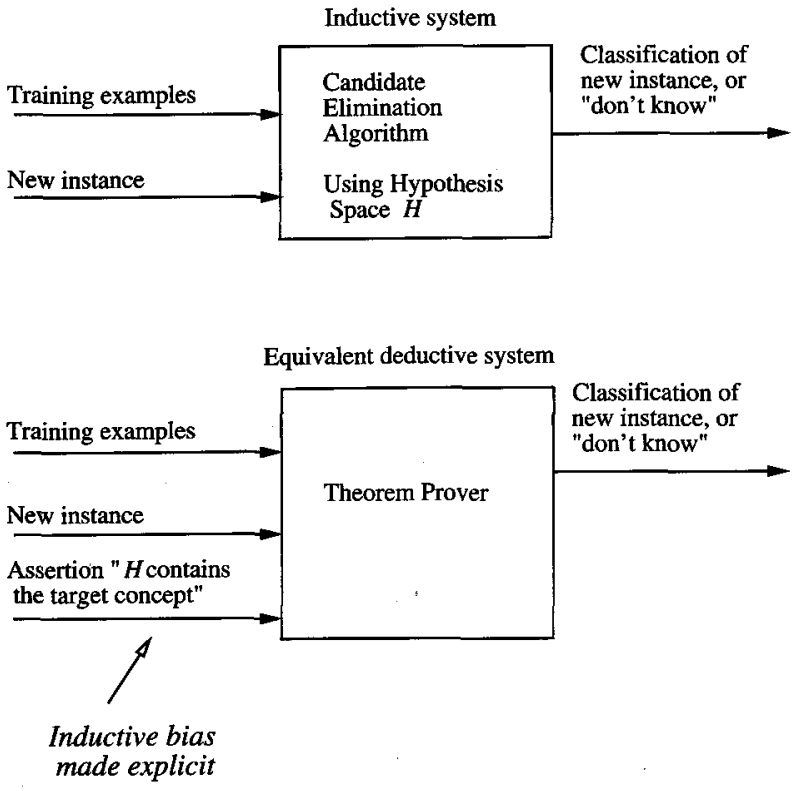


FIGURE 2.8
Modeling inductive systems by equivalent deductive systems. The input-output behavior of the CANDIDATE-ELIMINATION algorithm using a hypothesis space H is identical to that of a deductive theorem prover utilizing the assertion "H contains the target concept." This assertion is therefore called the *inductive bias* of the CANDIDATE-ELIMINATION algorithm. Characterizing inductive systems by their inductive bias allows modeling them by their equivalent deductive systems. This provides a way to compare inductive systems according to their policies for generalizing beyond the observed training data.

theorem prover is given these same two inputs plus the assertion "H contains the target concept." These two systems will in principle produce identical outputs for every possible input set of training examples and every possible new instance in X. Of course the inductive bias that is explicitly input to the theorem prover is only implicit in the code of the CANDIDATE-ELIMINATION algorithm. In a sense, it exists only in the eye of us beholders. Nevertheless, it is a perfectly well-defined set of assertions.

One advantage of viewing inductive inference systems in terms of their inductive bias is that it provides a nonprocedural means of characterizing their policy for generalizing beyond the observed data. A second advantage is that it allows comparison of different learners according to the strength of the inductive bias they employ. Consider, for example, the following three learning algorithms, which are listed from weakest to strongest bias.

1. ROTE-LEARNER: Learning corresponds simply to storing each observed training example in memory. Subsequent instances are classified by looking them

up in memory. If the instance is found in memory, the stored classification is returned. Otherwise, the system refuses to classify the new instance.

2. CANDIDATE-ELIMINATION algorithm: New instances are classified only in the case where all members of the current version space agree on the classification. Otherwise, the system refuses to classify the new instance.
3. FIND-S: This algorithm, described earlier, finds the most specific hypothesis consistent with the training examples. It then uses this hypothesis to classify all subsequent instances.

The ROTE-LEARNER has no inductive bias. The classifications it provides for new instances follow deductively from the observed training examples, with no additional assumptions required. The CANDIDATE-ELIMINATION algorithm has a stronger inductive bias: that the target concept can be represented in its hypothesis space. Because it has a stronger bias, it will classify some instances that the ROTE-LEARNER will not. Of course the correctness of such classifications will depend completely on the correctness of this inductive bias. The FIND-S algorithm has an even stronger inductive bias. In addition to the assumption that the target concept can be described in its hypothesis space, it has an additional inductive bias assumption: that all instances are negative instances unless the opposite is entailed by its other knowledge.†

As we examine other inductive inference methods, it is useful to keep in mind this means of characterizing them and the strength of their inductive bias. More strongly biased methods make more inductive leaps, classifying a greater proportion of unseen instances. Some inductive biases correspond to categorical assumptions that completely rule out certain concepts, such as the bias "the hypothesis space H includes the target concept." Other inductive biases merely rank order the hypotheses by stating preferences such as "more specific hypotheses are preferred over more general hypotheses." Some biases are implicit in the learner and are unchangeable by the learner, such as the ones we have considered here. In Chapters 11 and 12 we will see other systems whose bias is made explicit as a set of assertions represented and manipulated by the learner.

2.8 SUMMARY AND FURTHER READING

The main points of this chapter include:

- Concept learning can be cast as a problem of searching through a large predefined space of potential hypotheses.
- The general-to-specific partial ordering of hypotheses, which can be defined for any concept learning problem, provides a useful structure for organizing the search through the hypothesis space.

†Notice this last inductive bias assumption involves a kind of default, or nonmonotonic reasoning.

- The FIND-S algorithm utilizes this general-to-specific ordering, performing a specific-to-general search through the hypothesis space along one branch of the partial ordering, to find the most specific hypothesis consistent with the training examples.
- The CANDIDATE-ELIMINATION algorithm utilizes this general-to-specific ordering to compute the version space (the set of all hypotheses consistent with the training data) by incrementally computing the sets of maximally specific (S) and maximally general (G) hypotheses.
- Because the S and G sets delimit the entire set of hypotheses consistent with the data, they provide the learner with a description of its uncertainty regarding the exact identity of the target concept. This version space of alternative hypotheses can be examined to determine whether the learner has converged to the target concept, to determine when the training data are inconsistent, to generate informative queries to further refine the version space, and to determine which unseen instances can be unambiguously classified based on the partially learned concept.
- Version spaces and the CANDIDATE-ELIMINATION algorithm provide a useful conceptual framework for studying concept learning. However, this learning algorithm is not robust to noisy data or to situations in which the unknown target concept is not expressible in the provided hypothesis space. Chapter 10 describes several concept learning algorithms based on the general-to-specific ordering, which are robust to noisy data.
- Inductive learning algorithms are able to classify unseen examples only because of their implicit inductive bias for selecting one consistent hypothesis over another. The bias associated with the CANDIDATE-ELIMINATION algorithm is that the target concept can be found in the provided hypothesis space ($c \in H$). The output hypotheses and classifications of subsequent instances follow *deductively* from this assumption together with the observed training data.
- If the hypothesis space is enriched to the point where there is a hypothesis corresponding to every possible subset of instances (the power set of the instances), this will remove any inductive bias from the CANDIDATE-ELIMINATION algorithm. Unfortunately, this also removes the ability to classify any instance beyond the observed training examples. An unbiased learner cannot make inductive leaps to classify unseen examples.

The idea of concept learning and using the general-to-specific ordering have been studied for quite some time. Bruner et al. (1957) provided an early study of concept learning in humans, and Hunt and Hovland (1963) an early effort to automate it. Winston's (1970) widely known Ph.D. dissertation cast concept learning as a search involving generalization and specialization operators. Plotkin (1970, 1971) provided an early formalization of the *more-general-than* relation, as well as the related notion of θ-subsumption (discussed in Chapter 10). Simon and Lea (1973) give an early account of learning as search through a hypothesis

space. Other early concept learning systems include (Popplestone 1969; Michalski 1973; Buchanan 1974; Vere 1975; Hayes-Roth 1974). A very large number of algorithms have since been developed for concept learning based on symbolic representations. Chapter 10 describes several more recent algorithms for concept learning, including algorithms that learn concepts represented in first-order logic, algorithms that are robust to noisy training data, and algorithms whose performance degrades gracefully if the target concept is not representable in the hypothesis space considered by the learner.

Version spaces and the CANDIDATE-ELIMINATION algorithm were introduced by Mitchell (1977, 1982). The application of this algorithm to inferring rules of mass spectroscopy is described in (Mitchell 1979), and its application to learning search control rules is presented in (Mitchell et al. 1983). Haussler (1988) shows that the size of the general boundary can grow exponentially in the number of training examples, even when the hypothesis space consists of simple conjunctions of features. Smith and Rosenbloom (1990) show a simple change to the representation of the G set that can improve complexity in certain cases, and Hirsh (1992) shows that learning can be polynomial in the number of examples in some cases when the G set is not stored at all. Subramanian and Feigenbaum (1986) discuss a method that can generate efficient queries in certain cases by factoring the version space. One of the greatest practical limitations of the CANDIDATE-ELIMINATION algorithm is that it requires noise-free training data. Mitchell (1979) describes an extension that can handle a bounded, predetermined number of misclassified examples, and Hirsh (1990, 1994) describes an elegant extension for handling bounded noise in real-valued attributes that describe the training examples. Hirsh (1990) describes an INCREMENTAL VERSION SPACE MERGING algorithm that generalizes the CANDIDATE-ELIMINATION algorithm to handle situations in which training information can be different types of constraints represented using version spaces. The information from each constraint is represented by a version space and the constraints are then combined by intersecting the version spaces. Sebag (1994, 1996) presents what she calls a disjunctive version space approach to learning disjunctive concepts from noisy data. A separate version space is learned for each positive training example, then new instances are classified by combining the votes of these different version spaces. She reports experiments in several problem domains demonstrating that her approach is competitive with other widely used induction methods such as decision tree learning and k-NEAREST NEIGHBOR.

EXERCISES

2.1. Explain why the size of the hypothesis space in the *EnjoySport* learning task is 973. How would the number of possible instances and possible hypotheses increase with the addition of the attribute *WaterCurrent*, which can take on the values *Light*, *Moderate*, or *Strong*? More generally, how does the number of possible instances and hypotheses grow with the addition of a new attribute A that takes on k possible values?