MACHINE LEARNING

TOM M. MITCHELL

Machine Learning

Tom M. Mitchell

Product Details

- **Hardcover:** 432 pages ; Dimensions (in inches): 0.75 x 10.00 x 6.50
- **Publisher:** McGraw-Hill Science/Engineering/Math; (March 1, 1997)
- **ISBN:** 0070428077
- **Average Customer Review:** Based on 16 reviews.
- **Amazon.com Sales Rank:** 42,816
- **Popular in:** Redmond, WA (#17) , Ithaca, NY (#9)

Editorial Reviews

From Book News, Inc. An introductory text on primary approaches to machine learning and the study of computer algorithms that improve automatically through experience. Introduce basics concepts from statistics, artificial intelligence, information theory, and other disciplines as need arises, with balanced coverage of theory and practice, and presents major algorithms with illustrations of their use. Includes chapter exercises. Online data sets and implementations of several algorithms are available on a Web site. No prior background in artificial intelligence or statistics is assumed. For advanced undergraduates and graduate students in computer science, engineering, statistics, and social sciences, as well as software professionals. *Book News, Inc.®, Portland, OR*

Book Info: Presents the key algorithms and theory that form the core of machine learning. Discusses such theoretical issues as How does learning performance vary with the number of training examples presented? and Which learning algorithms are most appropriate for various types of learning tasks? DLC: Computer algorithms.

Book Description: This book covers the field of machine learning, which is the study of algorithms that allow computer programs to automatically improve through experience. The book is intended to support upper level undergraduate and introductory level graduate courses in machine learning

PREFACE

The field of machine learning is concerned with the question of how to construct computer programs that automatically improve with experience. In recent years many successful machine learning applications have been developed, ranging from data-mining programs that learn to detect fraudulent credit card transactions, to information-filtering systems that learn users' reading preferences, to autonomous vehicles that learn to drive on public highways. At the same time, there have been important advances in the theory and algorithms that form the foundations of this field.

The goal of this textbook is to present the key algorithms and theory that form the core of machine learning. Machine learning draws on concepts and results from many fields, including statistics, artificial intelligence, philosophy, information theory, biology, cognitive science, computational complexity, and control theory. My belief is that the best way to learn about machine learning is to view it from all of these perspectives and to understand the problem settings, algorithms, and assumptions that underlie each. In the past, this has been difficult due to the absence of a broad-based single source introduction to the field. The primary goal of this book is to provide such an introduction.

Because of the interdisciplinary nature of the material, this book makes few assumptions about the background of the reader. Instead, it introduces basic concepts from statistics, artificial intelligence, information theory, and other disciplines as the need arises, focusing on just those concepts most relevant to machine learning. The book is intended for both undergraduate and graduate students in fields such as computer science, engineering, statistics, and the social sciences, and as a reference for software professionals and practitioners. Two principles that guided the writing of the book were that it should be accessible to undergraduate students and that it should contain the material I would want my own Ph.D. students to learn before beginning their doctoral research in machine learning.

A third principle that guided the writing of this book was that it should present a balance of theory and practice. Machine learning theory attempts to answer questions such as "How does learning performance vary with the number of training examples presented?" and "Which learning algorithms are most appropriate for various types of learning tasks?" This book includes discussions of these and other theoretical issues, drawing on theoretical constructs from statistics, computational complexity, and Bayesian analysis. The practice of machine learning is covered by presenting the major algorithms in the field, along with illustrative traces of their operation. Online data sets and implementations of several algorithms are available via the World Wide Web at http://www.cs.cmu.edu/~tom/mlbook.html. These include neural network code and data for face recognition, decision tree learning code and data for financial loan analysis, and Bayes classifier code and data for analyzing text documents. I am grateful to a number of colleagues who have helped to create these online resources, including Jason Rennie, Paul Hsiung, Jeff Shufelt, Matt Glickman, Scott Davies, Joseph O'Sullivan, Ken Lang, Andrew McCallum, and Thorsten Joachims.

ACKNOWLEDGMENTS

In writing this book, I have been fortunate to be assisted by technical experts in many of the subdisciplines that make up the field of machine learning. This book could not have been written without their help. I am deeply indebted to the following scientists who took the time to review chapter drafts and, in many cases, to tutor me and help organize chapters in their individual areas of expertise.

Avrim Blum, Jaime Carbonell, William Cohen, Greg Cooper, Mark Craven, Ken DeJong, Jerry DeJong, Tom Dietterich, Susan Epstein, Oren Etzioni, Scott Fahlman, Stephanie Forrest, David Haussler, Haym Hirsh, Rob Holte, Leslie Pack Kaelbling, Dennis Kibler, Moshe Koppel, John Koza, Miroslav Kubat, John Lafferty, Ramon Lopez de Mantaras, Sridhar Mahadevan, Stan Matwin, Andrew McCallum, Raymond Mooney, Andrew Moore, Katharina Morik, Steve Muggleton, Michael Pazzani, David Poole, Armand Prieditis, Jim Reggia, Stuart Russell, Lorenza Saitta, Claude Sammut, Jeff Schneider, Jude Shavlik, Devika Subramanian, Michael Swain, Gheorgh Tecuci, Sebastian Thrun, Peter Turney, Paul Utgoff, Manuela Veloso, Alex Waibel, Stefan Wrobel, and Yiming Yang.

I am also grateful to the many instructors and students at various universities who have field tested various drafts of this book and who have contributed their suggestions. Although there is no space to thank the hundreds of students, instructors, and others who tested earlier drafts of this book, I would like to thank the following for particularly helpful comments and discussions:

Shumeet Baluja, Andrew Banas, Andy Barto, Jim Blackson, Justin Boyan, Rich Caruana, Philip Chan, Jonathan Cheyer, Lonnie Chrisman, Dayne Freitag, Geoff Gordon, Warren Greiff, Alexander Harm, Tom Ioerger, Thorsten

Joachim, Atsushi Kawamura, Martina Klose, Sven Koenig, Jay Modi, Andrew Ng, Joseph O'Sullivan, Patrawadee Prasangsit, Doina Precup, Bob Price, Choon Quek, Sean Slattery, Belinda Thom, Astro Teller, Will Tracz

I would like to thank Joan Mitchell for creating the index for the book. I also would like to thank Jean Harpley for help in editing many of the figures. Jane Loftus from ETP Harrison improved the presentation significantly through her copyediting of the manuscript and generally helped usher the manuscript through the intricacies of final production. Eric Munson, my editor at McGraw Hill, provided encouragement and expertise in all phases of this project.

As always, the greatest debt one owes is to one's colleagues, friends, and family. In my case, this debt is especially large. I can hardly imagine a more intellectually stimulating environment and supportive set of friends than those I have at Carnegie Mellon. Among the many here who helped, I would especially like to thank Sebastian Thrun, who throughout this project was a constant source of encouragement, technical expertise, and support of all kinds. My parents, as always, encouraged and asked "Is it done yet?" at just the right times. Finally, I must thank my family: Meghan, Shannon, and Joan. They are responsible for this book in more ways than even they know. This book is dedicated to them.

Tom M. Mitchell

CONTENTS

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

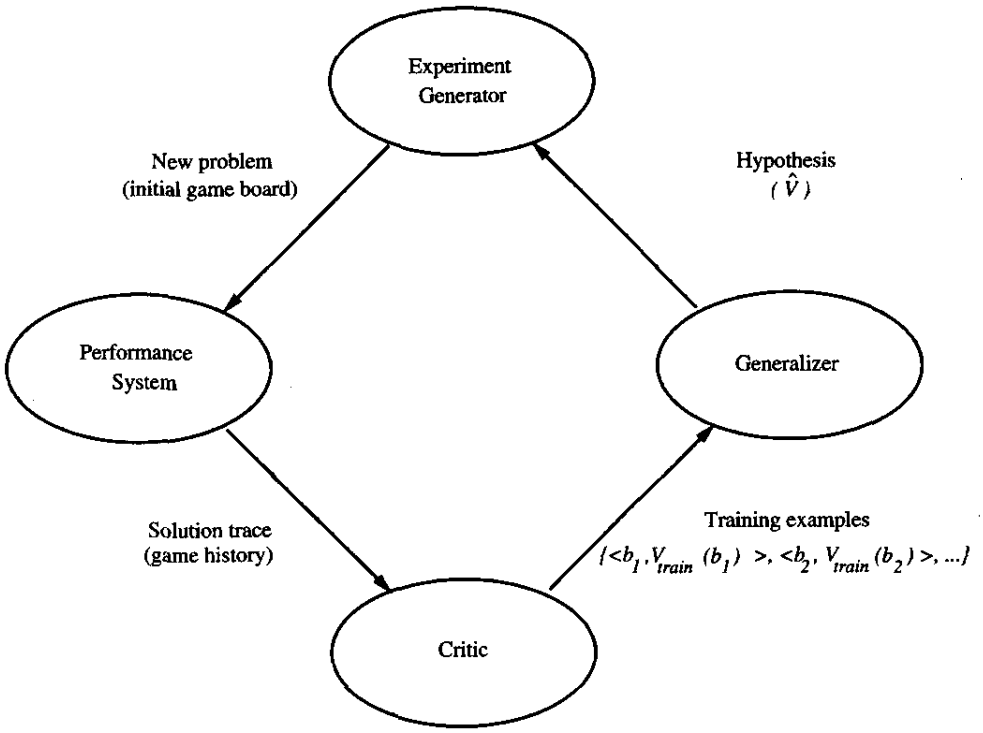


FIGURE 1.1
Final design of the checkers learning program.

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

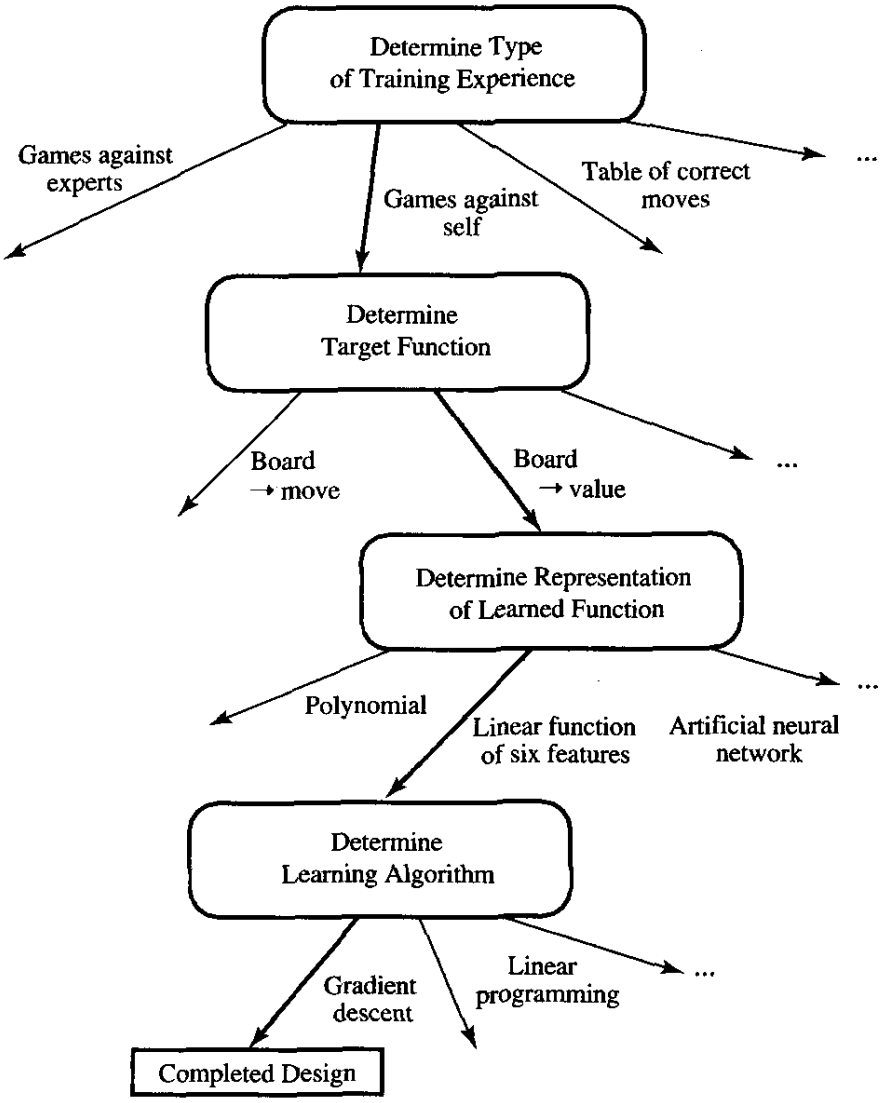


FIGURE 1.2
Summary of choices in designing the checkers learning program.

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), "$\emptyset$" (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 all 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*

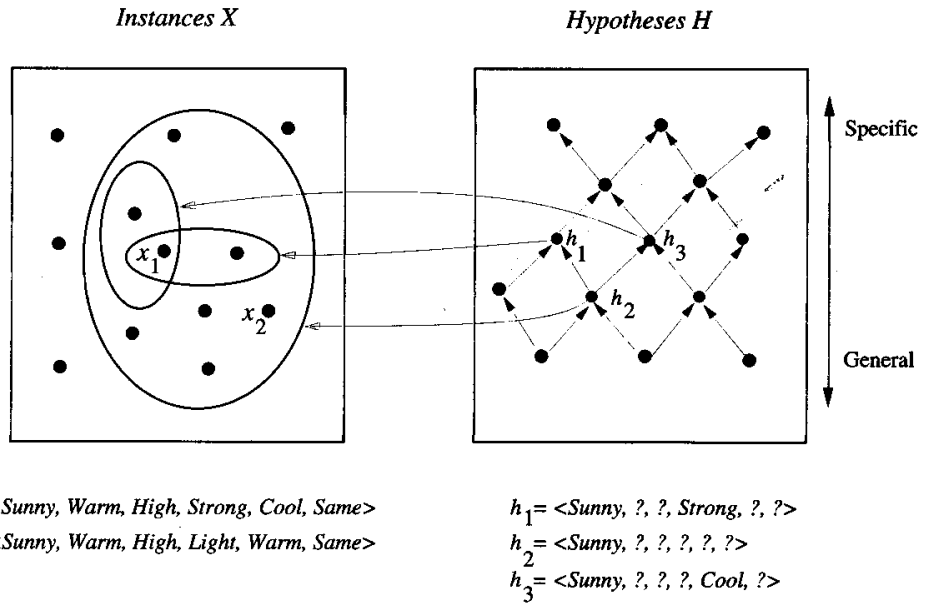


FIGURE 2.1
Instances, hypotheses, and the *more_general_than* relation. The box on the left represents the set X of all instances, the box on the right the set H of all hypotheses. Each hypothesis corresponds to some subset of X—the subset of instances that it classifies positive. The arrows connecting 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

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

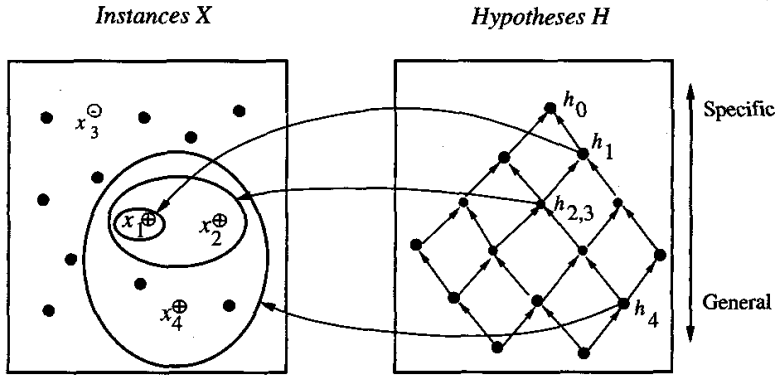
Upon encountering the third training example—in this case a negative example—the algorithm makes no change to h. In fact, the FIND-S algorithm simply *ignores every negative example*! While this may at first seem strange, notice that in the current case our hypothesis h is already consistent with the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

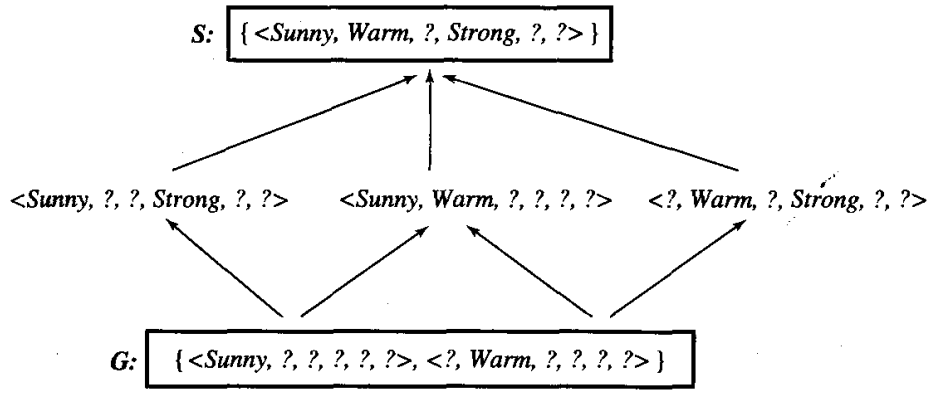


FIGURE 2.3
A version space with its general and specific boundary sets. The version space includes all six hypotheses shown here, but can be represented more simply by S and G. Arrows indicate instances of the *more_general_than* relation. This is the version space for the *EnjoySport* concept learning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

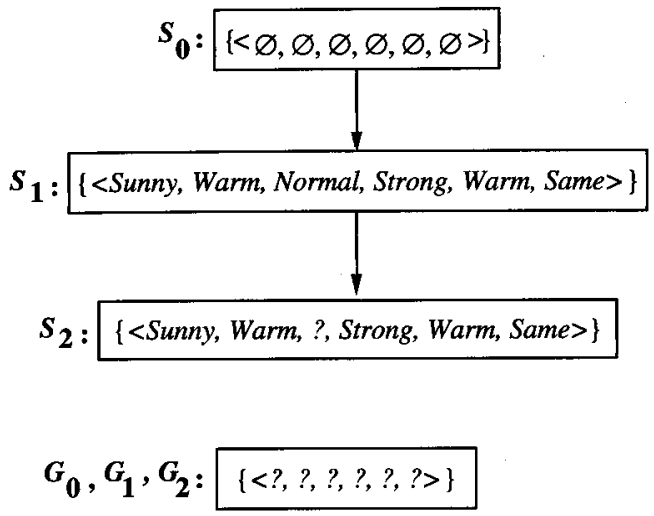
TABLE 2.5
CANDIDATE-ELIMINATION algorithm using version spaces. Notice the duality in how positive and negative examples influence S and G.

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*

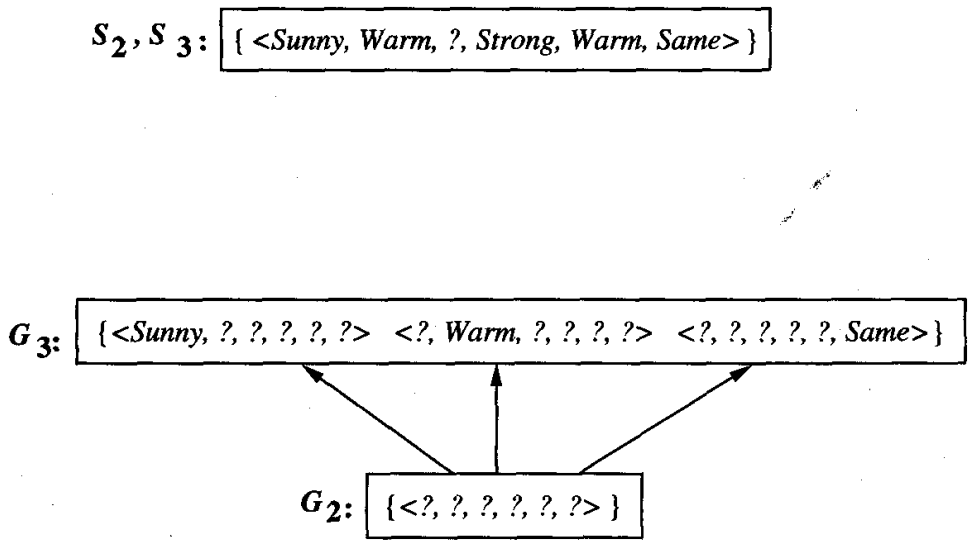
2. *<Sunny, Warm, High, Strong, Warm, Same>, Enjoy Sport = Yes*

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:

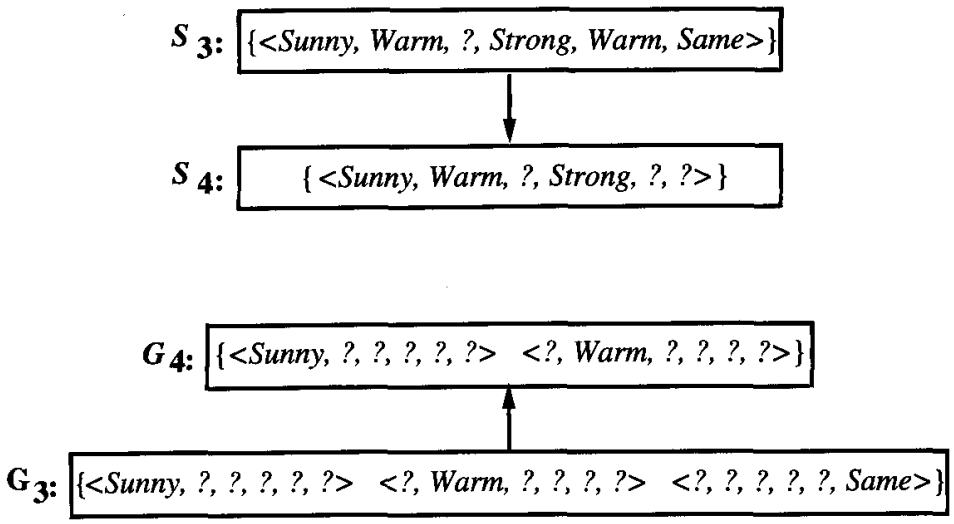
3. *<Rainy, Cold, High, Strong, Warm, Change>, EnjoySport=No*

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.

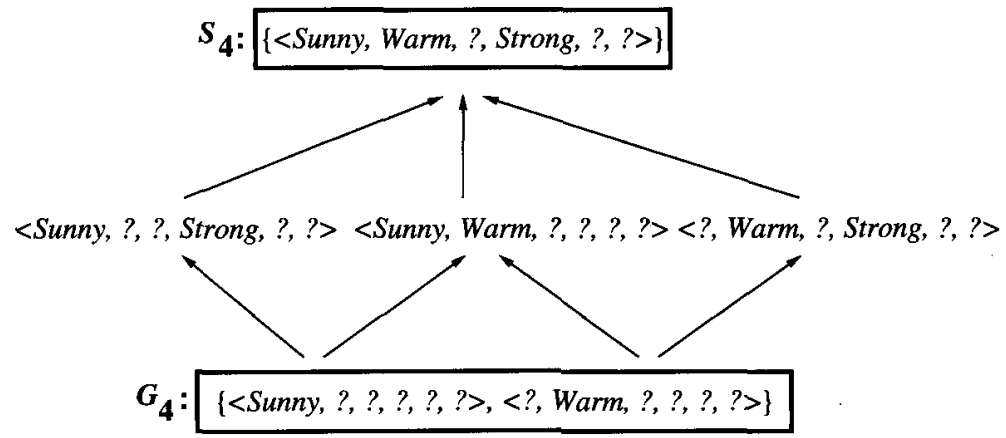


FIGURE 2.7
The final version space for the *EnjoySport* concept learning problem and training examples described earlier.

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

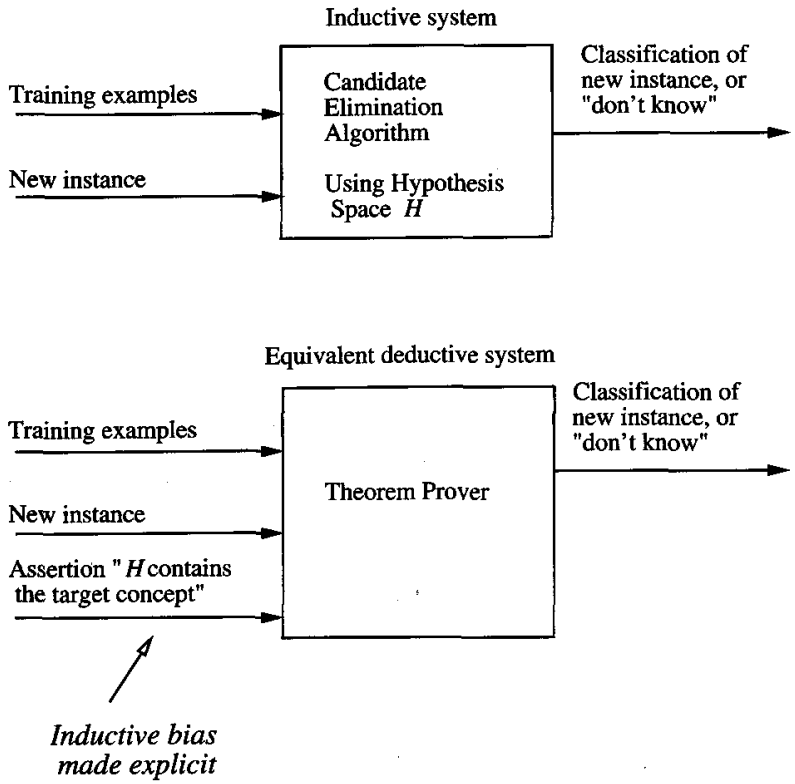


FIGURE 2.8
Modeling inductive systems by equivalent deductive systems. The input-output behavior of the CANDIDATE-ELIMINATION algorithm using a hypothesis space H is identical to that of a deductive theorem prover utilizing the assertion "H contains the target concept." This assertion is therefore called 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?

2.2. Give the sequence of S and G boundary sets computed by the CANDIDATE-ELIMINATION algorithm if it is given the sequence of training examples from Table 2.1 *in reverse order.* Although the final version space will be the same regardless of the sequence of examples (why?), the sets S and G computed at intermediate stages will, of course, depend on this sequence. Can you come up with ideas for ordering the training examples to minimize the sum of the sizes of these intermediate S and G sets for the H used in the *EnjoySport* example?

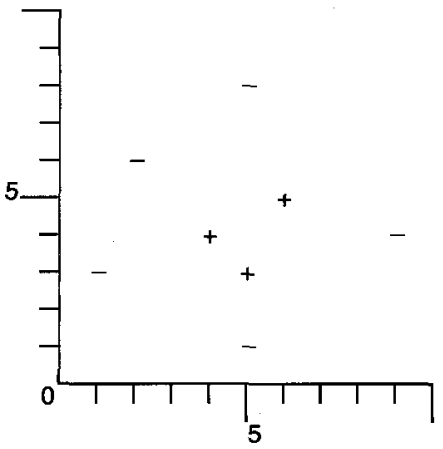
2.3. Consider again the *EnjoySport* learning task and the hypothesis space H described in Section 2.2. Let us define a new hypothesis space H' that consists of all *pairwise* disjunctions of the hypotheses in H. For example, a typical hypothesis in H' is

$$\langle ?, Cold, High, ?, ?, ? \rangle \vee \langle Sunny, ?, High, ?, ?, Same \rangle$$

Trace the CANDIDATE-ELIMINATION algorithm for the hypothesis space H' given the sequence of training examples from Table 2.1 (i.e., show the sequence of S and G boundary sets.)

2.4. Consider the instance space consisting of integer points in the x, y plane and the set of hypotheses H consisting of rectangles. More precisely, hypotheses are of the form $a \leq x \leq b, c \leq y \leq d$, where a, b, c, and d can be any integers.

(a) Consider the version space with respect to the set of positive (+) and negative (−) training examples shown below. What is the S boundary of the version space in this case? Write out the hypotheses and draw them in on the diagram.



(b) What is the G boundary of this version space? Write out the hypotheses and draw them in.

(c) Suppose the learner may now suggest a new x, y instance and ask the trainer for its classification. Suggest a query guaranteed to reduce the size of the version space, regardless of how the trainer classifies it. Suggest one that will not.

(d) Now assume you are a teacher, attempting to teach a particular target concept (e.g., $3 \leq x \leq 5, 2 \leq y \leq 9$). What is the smallest number of training examples you can provide so that the CANDIDATE-ELIMINATION algorithm will perfectly learn the target concept?

2.5. Consider the following sequence of positive and negative training examples describing the concept "pairs of people who live in the same house." Each training example describes an *ordered* pair of people, with each person described by their sex, hair

color (black, brown, or blonde), height (tall, medium, or short), and nationality (US, French, German, Irish, Indian, Japanese, or Portuguese).

$$+ \quad \langle \langle male\ brown\ tall\ US \rangle \langle female\ black\ short\ US \rangle \rangle$$

$$+ \quad \langle \langle male\ brown\ short\ French \rangle \langle female\ black\ short\ US \rangle \rangle$$

$$- \quad \langle \langle female\ brown\ tall\ German \rangle \langle female\ black\ short\ Indian \rangle \rangle$$

$$+ \quad \langle \langle male\ brown\ tall\ Irish \rangle \langle female\ brown\ short\ Irish \rangle \rangle$$

Consider a hypothesis space defined over these instances, in which each hypothesis is represented by a pair of 4-tuples, and where each attribute constraint may be a specific value, "?," or "Ø," just as in the *EnjoySport* hypothesis representation. For example, the hypothesis

$$\langle \langle male\ ?\ tall\ ? \rangle \langle female\ ?\ ?\ Japanese \rangle \rangle$$

represents the set of all pairs of people where the first is a tall male (of any nationality and hair color), and the second is a Japanese female (of any hair color and height).

(*a*) Provide a hand trace of the CANDIDATE-ELIMINATION algorithm learning from the above training examples and hypothesis language. In particular, show the specific and general boundaries of the version space after it has processed the first training example, then the second training example, etc.

(*b*) How many distinct hypotheses from the given hypothesis space are consistent with the following single positive training example?

$$+ \quad \langle \langle male\ black\ short\ Portuguese \rangle \langle female\ blonde\ tall\ Indian \rangle \rangle$$

(*c*) Assume the learner has encountered only the positive example from part (*b*), and that it is now allowed to query the trainer by generating any instance and asking the trainer to classify it. Give a specific sequence of queries that assures the learner will converge to the single correct hypothesis, whatever it may be (assuming that the target concept is describable within the given hypothesis language). Give the shortest sequence of queries you can find. How does the length of this sequence relate to your answer to question (*b*)?

(*d*) Note that this hypothesis language cannot express all concepts that can be defined over the instances (i.e., we can define sets of positive and negative examples for which there is no corresponding describable hypothesis). If we were to enrich the language so that it *could* express all concepts that can be defined over the instance language, then how would your answer to (*c*) change?

2.6. Complete the proof of the version space representation theorem (Theorem 2.1).

2.7. Consider a concept learning problem in which each instance is a real number, and in which each hypothesis is an interval over the reals. More precisely, each hypothesis in the hypothesis space H is of the form $a < x < b$, where a and b are any real constants, and x refers to the instance. For example, the hypothesis $4.5 < x < 6.1$ classifies instances between 4.5 and 6.1 as positive, and others as negative. Explain informally why there cannot be a maximally specific consistent hypothesis for any set of positive training examples. Suggest a slight modification to the hypothesis representation so that there will be.

2.8. In this chapter, we commented that given an unbiased hypothesis space (the power set of the instances), the learner would find that each unobserved instance would match exactly half the current members of the version space, regardless of which training examples had been observed. Prove this. In particular, prove that for any instance space X, any set of training examples D, and any instance $x \in X$ not present in D, that if H is the power set of X, then exactly half the hypotheses in $VS_{H,D}$ will classify x as positive and half will classify it as negative.

2.9. Consider a learning problem where each instance is described by a conjunction of n boolean attributes $a_1 \ldots a_n$. Thus, a typical instance would be

$$(a_1 = T) \wedge (a_2 = F) \wedge \ldots \wedge (a_n = T)$$

Now consider a hypothesis space H in which each hypothesis is a *disjunction* of constraints over these attributes. For example, a typical hypothesis would be

$$(a_1 = T) \vee (a_5 = F) \vee (a_7 = T)$$

Propose an algorithm that accepts a sequence of training examples and outputs a consistent hypothesis if one exists. Your algorithm should run in time that is polynomial in n and in the number of training examples.

2.10. Implement the FIND-S algorithm. First verify that it successfully produces the trace in Section 2.4 for the *EnjoySport* example. Now use this program to study the number of random training examples required to exactly learn the target concept. Implement a training example generator that generates random instances, then classifies them according to the target concept:

$$\langle Sunny, Warm, ?, ?, ?, ? \rangle$$

Consider training your FIND-S program on randomly generated examples and measuring the number of examples required before the program's hypothesis is identical to the target concept. Can you predict the average number of examples required? Run the experiment at least 20 times and report the mean number of examples required. How do you expect this number to vary with the number of "?"s in the target concept? How would it vary with the number of attributes used to describe instances and hypotheses?

REFERENCES

Bruner, J. S., Goodnow, J. J., & Austin, G. A. (1957). *A study of thinking.* New York: John Wiley & Sons.

Buchanan, B. G. (1974). Scientific theory formation by computer. In J. C. Simon (Ed.), *Computer Oriented Learning Processes.* Leyden: Noordhoff.

Gunter, C. A., Ngair, T., Panangaden, P., & Subramanian, D. (1991). The common order-theoretic structure of version spaces and ATMS's. *Proceedings of the National Conference on Artificial Intelligence* (pp. 500–505). Anaheim.

Haussler, D. (1988). Quantifying inductive bias: AI learning algorithms and Valiant's learning framework. *Artificial Intelligence,* 36, 177–221.

Hayes-Roth, F. (1974). Schematic classification problems and their solution. *Pattern Recognition,* 6, 105–113.

Hirsh, H. (1990). Incremental version space merging: A general framework for concept learning. Boston: Kluwer.

Hirsh, H. (1991). Theoretical underpinnings of version spaces. *Proceedings of the 12th IJCAI* (pp. 665–670). Sydney.

Hirsh, H. (1994). Generalizing version spaces. *Machine Learning*, 17(1), 5–46.

Hunt, E. G., & Hovland, D. I. (1963). Programming a model of human concept formation. In E. Feigenbaum & J. Feldman (Eds.), *Computers and thought* (pp. 310–325). New York: McGraw Hill.

Michalski, R. S. (1973). AQVAL/1: Computer implementation of a variable valued logic system VL1 and examples of its application to pattern recognition. *Proceedings of the 1st International Joint Conference on Pattern Recognition* (pp. 3–17).

Mitchell, T. M. (1977). Version spaces: A candidate elimination approach to rule learning. *Fifth International Joint Conference on AI* (pp. 305–310). Cambridge, MA: MIT Press.

Mitchell, T. M. (1979). *Version spaces: An approach to concept learning,* (Ph.D. dissertation). Electrical Engineering Dept., Stanford University, Stanford, CA.

Mitchell, T. M. (1982). Generalization as search. *Artificial Intelligence,* 18(2), 203–226.

Mitchell, T. M., Utgoff, P. E., & Banerji, R. (1983). Learning by experimentation: Acquiring and modifying problem-solving heuristics. In Michalski, Carbonell, & Mitchell (Eds.), *Machine Learning* (Vol. 1, pp. 163–190). Tioga Press.

Plotkin, G. D. (1970). A note on inductive generalization. In Meltzer & Michie (Eds.), *Machine Intelligence 5* (pp. 153–163). Edinburgh University Press.

Plotkin, G. D. (1971). A further note on inductive generalization. In Meltzer & Michie (Eds.), *Machine Intelligence 6* (pp. 104–124). Edinburgh University Press.

Popplestone, R. J. (1969). An experiment in automatic induction. In Meltzer & Michie (Eds.), *Machine Intelligence 5* (pp. 204–215). Edinburgh University Press.

Sebag, M. (1994). Using constraints to build version spaces. *Proceedings of the 1994 European Conference on Machine Learning*. Springer-Verlag.

Sebag, M. (1996). Delaying the choice of bias: A disjunctive version space approach. *Proceedings of the 13th International Conference on Machine Learning* (pp. 444–452). San Francisco: Morgan Kaufmann.

Simon, H. A., & Lea, G. (1973). Problem solving and rule induction: A unified view. In Gregg (Ed.), *Knowledge and Cognition* (pp. 105–127). New Jersey: Lawrence Erlbaum Associates.

Smith, B. D., & Rosenbloom, P. (1990). Incremental non-backtracking focusing: A polynomially bounded generalization algorithm for version spaces. *Proceedings of the 1990 National Conference on Artificial Intelligence* (pp. 848–853). Boston.

Subramanian, D., & Feigenbaum, J. (1986). Factorization in experiment generation. *Proceedings of the 1986 National Conference on Artificial Intelligence* (pp. 518–522). Morgan Kaufmann.

Vere, S. A. (1975). Induction of concepts in the predicate calculus. *Fourth International Joint Conference on AI* (pp. 281–287). Tbilisi, USSR.

Winston, P. H. (1970). *Learning structural descriptions from examples,* (Ph.D. dissertation). [MIT Technical Report AI-TR-231].

CHAPTER 3

DECISION TREE LEARNING

Decision tree learning is one of the most widely used and practical methods for inductive inference. It is a method for approximating discrete-valued functions that is robust to noisy data and capable of learning disjunctive expressions. This chapter describes a family of decision tree learning algorithms that includes widely used algorithms such as ID3, ASSISTANT, and C4.5. These decision tree learning methods search a completely expressive hypothesis space and thus avoid the difficulties of restricted hypothesis spaces. Their inductive bias is a preference for small trees over large trees.

3.1 INTRODUCTION

Decision tree learning is a method for approximating discrete-valued target functions, in which the learned function is represented by a decision tree. Learned trees can also be re-represented as sets of if-then rules to improve human readability. These learning methods are among the most popular of inductive inference algorithms and have been successfully applied to a broad range of tasks from learning to diagnose medical cases to learning to assess credit risk of loan applicants.

3.2 DECISION TREE REPRESENTATION

Decision trees classify instances by sorting them down the tree from the root to some leaf node, which provides the classification of the instance. Each node in the tree specifies a test of some *attribute* of the instance, and each branch descending

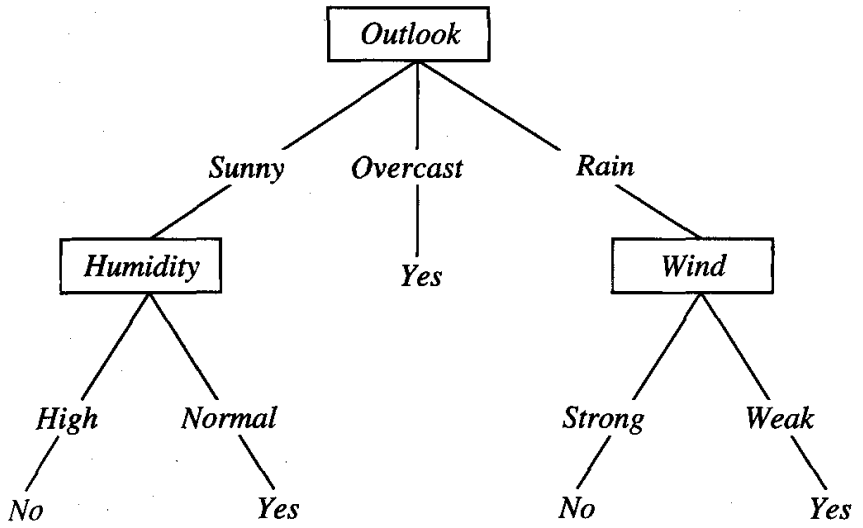


FIGURE 3.1
A decision tree for the concept *PlayTennis*. An example is classified by sorting it through the tree to the appropriate leaf node, then returning the classification associated with this leaf (in this case, *Yes* or *No*). This tree classifies Saturday mornings according to whether or not they are suitable for playing tennis.

from that node corresponds to one of the possible values for this attribute. An instance is classified by starting at the root node of the tree, testing the attribute specified by this node, then moving down the tree branch corresponding to the value of the attribute in the given example. This process is then repeated for the subtree rooted at the new node.

Figure 3.1 illustrates a typical learned decision tree. This decision tree classifies Saturday mornings according to whether they are suitable for playing tennis. For example, the instance

$$\langle Outlook = Sunny,\ Temperature = Hot,\ Humidity = High,\ Wind = Strong \rangle$$

would be sorted down the leftmost branch of this decision tree and would therefore be classified as a negative instance (i.e., the tree predicts that $PlayTennis = no$). This tree and the example used in Table 3.2 to illustrate the ID3 learning algorithm are adapted from (Quinlan 1986).

In general, decision trees represent a disjunction of conjunctions of constraints on the attribute values of instances. Each path from the tree root to a leaf corresponds to a conjunction of attribute tests, and the tree itself to a disjunction of these conjunctions. For example, the decision tree shown in Figure 3.1 corresponds to the expression

$$\begin{aligned} & (Outlook = Sunny \ \wedge \ Humidity = Normal) \\ \vee \quad & (Outlook = Overcast) \\ \vee \quad & (Outlook = Rain \ \wedge \ Wind = Weak) \end{aligned}$$

3.3 APPROPRIATE PROBLEMS FOR DECISION TREE LEARNING

Although a variety of decision tree learning methods have been developed with somewhat differing capabilities and requirements, decision tree learning is generally best suited to problems with the following characteristics:

- *Instances are represented by attribute-value pairs.* Instances are described by a fixed set of attributes (e.g., *Temperature*) and their values (e.g., *Hot*). The easiest situation for decision tree learning is when each attribute takes on a small number of disjoint possible values (e.g., *Hot*, *Mild*, *Cold*). However, extensions to the basic algorithm (discussed in Section 3.7.2) allow handling real-valued attributes as well (e.g., representing *Temperature* numerically).
- *The target function has discrete output values.* The decision tree in Figure 3.1 assigns a boolean classification (e.g., *yes* or *no*) to each example. Decision tree methods easily extend to learning functions with more than two possible output values. A more substantial extension allows learning target functions with real-valued outputs, though the application of decision trees in this setting is less common.
- *Disjunctive descriptions may be required.* As noted above, decision trees naturally represent disjunctive expressions.
- *The training data may contain errors.* Decision tree learning methods are robust to errors, both errors in classifications of the training examples and errors in the attribute values that describe these examples.
- *The training data may contain missing attribute values.* Decision tree methods can be used even when some training examples have unknown values (e.g., if the *Humidity* of the day is known for only some of the training examples). This issue is discussed in Section 3.7.4.

Many practical problems have been found to fit these characteristics. Decision tree learning has therefore been applied to problems such as learning to classify medical patients by their disease, equipment malfunctions by their cause, and loan applicants by their likelihood of defaulting on payments. Such problems, in which the task is to classify examples into one of a discrete set of possible categories, are often referred to as *classification problems*.

The remainder of this chapter is organized as follows. Section 3.4 presents the basic ID3 algorithm for learning decision trees and illustrates its operation in detail. Section 3.5 examines the hypothesis space search performed by this learning algorithm, contrasting it with algorithms from Chapter 2. Section 3.6 characterizes the inductive bias of this decision tree learning algorithm and explores more generally an inductive bias called Occam's razor, which corresponds to a preference for the most simple hypothesis. Section 3.7 discusses the issue of overfitting the training data, as well as strategies such as rule post-pruning to deal with this problem. This section also discusses a number of more advanced topics such as extending the algorithm to accommodate real-valued attributes, training data with unobserved attributes, and attributes with differing costs.

3.4 THE BASIC DECISION TREE LEARNING ALGORITHM

Most algorithms that have been developed for learning decision trees are variations on a core algorithm that employs a top-down, greedy search through the space of possible decision trees. This approach is exemplified by the ID3 algorithm (Quinlan 1986) and its successor C4.5 (Quinlan 1993), which form the primary focus of our discussion here. In this section we present the basic algorithm for decision tree learning, corresponding approximately to the ID3 algorithm. In Section 3.7 we consider a number of extensions to this basic algorithm, including extensions incorporated into C4.5 and other more recent algorithms for decision tree learning.

Our basic algorithm, ID3, learns decision trees by constructing them top-down, beginning with the question "which attribute should be tested at the root of the tree?" To answer this question, each instance attribute is evaluated using a statistical test to determine how well it alone classifies the training examples. The best attribute is selected and used as the test at the root node of the tree. A descendant of the root node is then created for each possible value of this attribute, and the training examples are sorted to the appropriate descendant node (i.e., down the branch corresponding to the example's value for this attribute). The entire process is then repeated using the training examples associated with each descendant node to select the best attribute to test at that point in the tree. This forms a greedy search for an acceptable decision tree, in which the algorithm never backtracks to reconsider earlier choices. A simplified version of the algorithm, specialized to learning boolean-valued functions (i.e., concept learning), is described in Table 3.1.

3.4.1 Which Attribute Is the Best Classifier?

The central choice in the ID3 algorithm is selecting which attribute to test at each node in the tree. We would like to select the attribute that is most useful for classifying examples. What is a good quantitative measure of the worth of an attribute? We will define a statistical property, called *information gain*, that measures how well a given attribute separates the training examples according to their target classification. ID3 uses this information gain measure to select among the candidate attributes at each step while growing the tree.

3.4.1.1 ENTROPY MEASURES HOMOGENEITY OF EXAMPLES

In order to define information gain precisely, we begin by defining a measure commonly used in information theory, called *entropy*, that characterizes the (im)purity of an arbitrary collection of examples. Given a collection S, containing positive and negative examples of some target concept, the entropy of S relative to this boolean classification is

$$Entropy(S) \equiv -p_{\oplus} \log_2 p_{\oplus} - p_{\ominus} \log_2 p_{\ominus} \quad (3.1)$$

ID3(*Examples*, *Target_attribute*, *Attributes*)

Examples are the training examples. Target_attribute is the attribute whose value is to be predicted by the tree. Attributes is a list of other attributes that may be tested by the learned decision tree. Returns a decision tree that correctly classifies the given Examples.

- Create a *Root* node for the tree
- If all *Examples* are positive, Return the single-node tree *Root*, with label = +
- If all *Examples* are negative, Return the single-node tree *Root*, with label = −
- If *Attributes* is empty, Return the single-node tree *Root*, with label = most common value of *Target_attribute* in *Examples*
- Otherwise Begin
  - $A \leftarrow$ the attribute from *Attributes* that best* classifies *Examples*
  - The decision attribute for *Root* $\leftarrow A$
  - For each possible value, v_i, of A,
    - Add a new tree branch below *Root*, corresponding to the test $A = v_i$
    - Let $Examples_{v_i}$ be the subset of *Examples* that have value v_i for A
    - If $Examples_{v_i}$ is empty
      - Then below this new branch add a leaf node with label = most common value of *Target_attribute* in *Examples*
      - Else below this new branch add the subtree ID3($Examples_{v_i}$, *Target_attribute*, *Attributes* − {A}))
- End
- Return *Root*

* The best attribute is the one with highest *information gain*, as defined in Equation (3.4).

TABLE 3.1
Summary of the ID3 algorithm specialized to learning boolean-valued functions. ID3 is a greedy algorithm that grows the tree top-down, at each node selecting the attribute that best classifies the local training examples. This process continues until the tree perfectly classifies the training examples, or until all attributes have been used.

where $p_\oplus$ is the proportion of positive examples in S and $p_\ominus$ is the proportion of negative examples in S. In all calculations involving entropy we define $0 \log 0$ to be 0.

To illustrate, suppose S is a collection of 14 examples of some boolean concept, including 9 positive and 5 negative examples (we adopt the notation [9+, 5−] to summarize such a sample of data). Then the entropy of S relative to this boolean classification is

$$\begin{aligned} Entropy([9+, 5-]) &= -(9/14)\log_2(9/14) - (5/14)\log_2(5/14) \\ &= 0.940 \end{aligned} \tag{3.2}$$

Notice that the entropy is 0 if all members of S belong to the same class. For example, if all members are positive ($p_\oplus = 1$), then $p_\ominus$ is 0, and $Entropy(S) = -1 \cdot \log_2(1) - 0 \cdot \log_2 0 = -1 \cdot 0 - 0 \cdot \log_2 0 = 0$. Note the entropy is 1 when the collection contains an equal number of positive and negative examples. If the collection contains unequal numbers of positive and negative examples, the

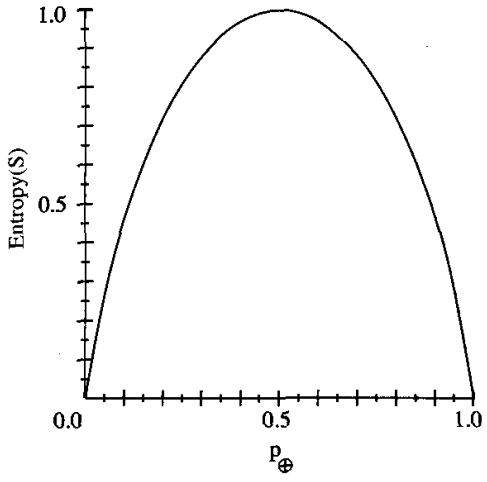


FIGURE 3.2
The entropy function relative to a boolean classification, as the proportion, $p_\oplus$, of positive examples varies between 0 and 1.

entropy is between 0 and 1. Figure 3.2 shows the form of the entropy function relative to a boolean classification, as $p_\oplus$ varies between 0 and 1.

One interpretation of entropy from information theory is that it specifies the minimum number of bits of information needed to encode the classification of an arbitrary member of S (i.e., a member of S drawn at random with uniform probability). For example, if $p_\oplus$ is 1, the receiver knows the drawn example will be positive, so no message need be sent, and the entropy is zero. On the other hand, if $p_\oplus$ is 0.5, one bit is required to indicate whether the drawn example is positive or negative. If $p_\oplus$ is 0.8, then a collection of messages can be encoded using on average less than 1 bit per message by assigning shorter codes to collections of positive examples and longer codes to less likely negative examples.

Thus far we have discussed entropy in the special case where the target classification is boolean. More generally, if the target attribute can take on c different values, then the entropy of S relative to this c-wise classification is defined as

$$Entropy(S) \equiv \sum_{i=1}^{c} -p_i \log_2 p_i \tag{3.3}$$

where p_i is the proportion of S belonging to class i. Note the logarithm is still base 2 because entropy is a measure of the expected encoding length measured in *bits*. Note also that if the target attribute can take on c possible values, the entropy can be as large as $\log_2 c$.

3.4.1.2 INFORMATION GAIN MEASURES THE EXPECTED REDUCTION IN ENTROPY

Given entropy as a measure of the impurity in a collection of training examples, we can now define a measure of the effectiveness of an attribute in classifying the training data. The measure we will use, called *information gain*, is simply the expected reduction in entropy caused by partitioning the examples according to this attribute. More precisely, the information gain, $Gain(S, A)$ of an attribute A,

relative to a collection of examples S, is defined as

$$Gain(S, A) \equiv Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v) \tag{3.4}$$

where $Values(A)$ is the set of all possible values for attribute A, and S_v is the subset of S for which attribute A has value v (i.e., $S_v = \{s \in S | A(s) = v\}$). Note the first term in Equation (3.4) is just the entropy of the original collection S, and the second term is the expected value of the entropy after S is partitioned using attribute A. The expected entropy described by this second term is simply the sum of the entropies of each subset S_v, weighted by the fraction of examples $\frac{|S_v|}{|S|}$ that belong to S_v. $Gain(S, A)$ is therefore the expected reduction in entropy caused by knowing the value of attribute A. Put another way, $Gain(S, A)$ is the information provided about the *target function value*, given the value of some other attribute A. The value of $Gain(S, A)$ is the number of bits saved when encoding the target value of an arbitrary member of S, by knowing the value of attribute A.

For example, suppose S is a collection of training-example days described by attributes including *Wind*, which can have the values *Weak* or *Strong*. As before, assume S is a collection containing 14 examples, [9+, 5−]. Of these 14 examples, suppose 6 of the positive and 2 of the negative examples have $Wind = Weak$, and the remainder have $Wind = Strong$. The information gain due to sorting the original 14 examples by the attribute *Wind* may then be calculated as

$$\begin{aligned}
Values(Wind) &= Weak, Strong \\
S &= [9+, 5-] \\
S_{Weak} &\leftarrow [6+, 2-] \\
S_{Strong} &\leftarrow [3+, 3-] \\
Gain(S, Wind) &= Entropy(S) - \sum_{v \in \{Weak, Strong\}} \frac{|S_v|}{|S|} Entropy(S_v) \\
&= Entropy(S) - (8/14) Entropy(S_{Weak}) \\
&\quad - (6/14) Entropy(S_{Strong}) \\
&= 0.940 - (8/14)0.811 - (6/14)1.00 \\
&= 0.048
\end{aligned}$$

Information gain is precisely the measure used by ID3 to select the best attribute at each step in growing the tree. The use of information gain to evaluate the relevance of attributes is summarized in Figure 3.3. In this figure the information gain of two different attributes, *Humidity* and *Wind*, is computed in order to determine which is the better attribute for classifying the training examples shown in Table 3.2.

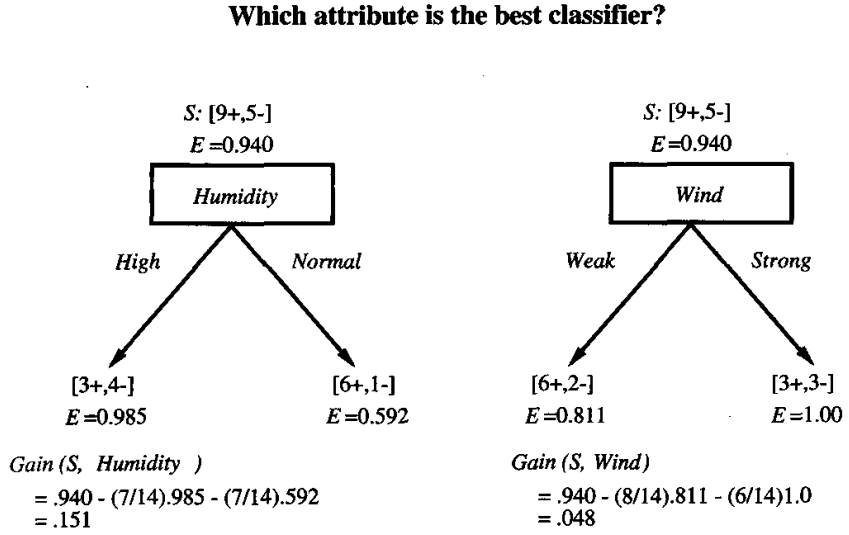


FIGURE 3.3
Humidity provides greater information gain than *Wind*, relative to the target classification. Here, E stands for entropy and S for the original collection of examples. Given an initial collection S of 9 positive and 5 negative examples, [9+, 5−], sorting these by their *Humidity* produces collections of [3+, 4−] ($Humidity = High$) and [6+, 1−] ($Humidity = Normal$). The information gained by this partitioning is .151, compared to a gain of only .048 for the attribute *Wind*.

3.4.2 An Illustrative Example

To illustrate the operation of ID3, consider the learning task represented by the training examples of Table 3.2. Here the target attribute *PlayTennis*, which can have values *yes* or *no* for different Saturday mornings, is to be predicted based on other attributes of the morning in question. Consider the first step through

Day	*Outlook*	*Temperature*	*Humidity*	*Wind*	*PlayTennis*
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

TABLE 3.2
Training examples for the target concept *PlayTennis*.

the algorithm, in which the topmost node of the decision tree is created. Which attribute should be tested first in the tree? ID3 determines the information gain for each candidate attribute (i.e., *Outlook, Temperature, Humidity*, and *Wind*), then selects the one with highest information gain. The computation of information gain for two of these attributes is shown in Figure 3.3. The information gain values for all four attributes are

$$Gain(S, Outlook) = 0.246$$
$$Gain(S, Humidity) = 0.151$$
$$Gain(S, Wind) = 0.048$$
$$Gain(S, Temperature) = 0.029$$

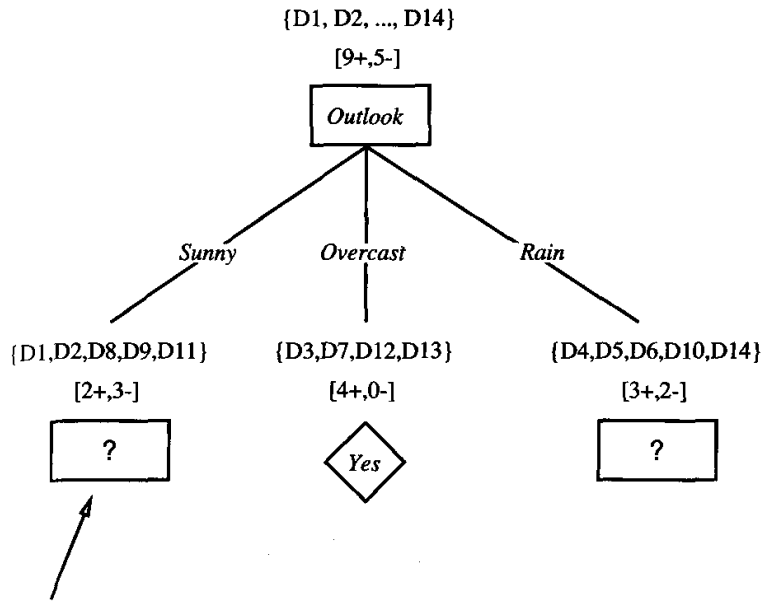
where S denotes the collection of training examples from Table 3.2.

According to the information gain measure, the *Outlook* attribute provides the best prediction of the target attribute, *PlayTennis*, over the training examples. Therefore, *Outlook* is selected as the decision attribute for the root node, and branches are created below the root for each of its possible values (i.e., *Sunny*, *Overcast*, and *Rain*). The resulting partial decision tree is shown in Figure 3.4, along with the training examples sorted to each new descendant node. Note that every example for which *Outlook* = *Overcast* is also a positive example of *PlayTennis*. Therefore, this node of the tree becomes a leaf node with the classification *PlayTennis* = *Yes*. In contrast, the descendants corresponding to *Outlook* = *Sunny* and *Outlook* = *Rain* still have nonzero entropy, and the decision tree will be further elaborated below these nodes.

The process of selecting a new attribute and partitioning the training examples is now repeated for each nonterminal descendant node, this time using only the training examples associated with that node. Attributes that have been incorporated higher in the tree are excluded, so that any given attribute can appear at most once along any path through the tree. This process continues for each new leaf node until either of two conditions is met: (1) every attribute has already been included along this path through the tree, or (2) the training examples associated with this leaf node all have the same target attribute value (i.e., their entropy is zero). Figure 3.4 illustrates the computations of information gain for the next step in growing the decision tree. The final decision tree learned by ID3 from the 14 training examples of Table 3.2 is shown in Figure 3.1.

3.5 HYPOTHESIS SPACE SEARCH IN DECISION TREE LEARNING

As with other inductive learning methods, ID3 can be characterized as searching a space of hypotheses for one that fits the training examples. The hypothesis space searched by ID3 is the set of possible decision trees. ID3 performs a simple-to-complex, hill-climbing search through this hypothesis space, beginning with the empty tree, then considering progressively more elaborate hypotheses in search of a decision tree that correctly classifies the training data. The evaluation function



S_{sunny} = {D1,D2,D8,D9,D11}

$Gain(S_{sunny}, Humidity)$ = .970 - (3/5) 0.0 - (2/5) 0.0 = .970

$Gain(S_{sunny}, Temperature)$ = .970 - (2/5) 0.0 - (2/5) 1.0 - (1/5) 0.0 = .570

$Gain(S_{sunny}, Wind)$ = .970 - (2/5) 1.0 - (3/5) .918 = .019

FIGURE 3.4
The partially learned decision tree resulting from the first step of ID3. The training examples are sorted to the corresponding descendant nodes. The *Overcast* descendant has only positive examples and therefore becomes a leaf node with classification *Yes*. The other two nodes will be further expanded, by selecting the attribute with highest information gain relative to the new subsets of examples.

that guides this hill-climbing search is the information gain measure. This search is depicted in Figure 3.5.

By viewing ID3 in terms of its search space and search strategy, we can get some insight into its capabilities and limitations.

ID3's hypothesis space of all decision trees is a *complete* space of finite discrete-valued functions, relative to the available attributes. Because every finite discrete-valued function can be represented by some decision tree, ID3 avoids one of the major risks of methods that search incomplete hypothesis spaces (such as methods that consider only conjunctive hypotheses): that the hypothesis space might not contain the target function.

ID3 maintains only a single current hypothesis as it searches through the space of decision trees. This contrasts, for example, with the earlier version space Candidate-Elimination method, which maintains the set of *all* hypotheses consistent with the available training examples. By determining only a single hypothesis, ID3 loses the capabilities that follow from

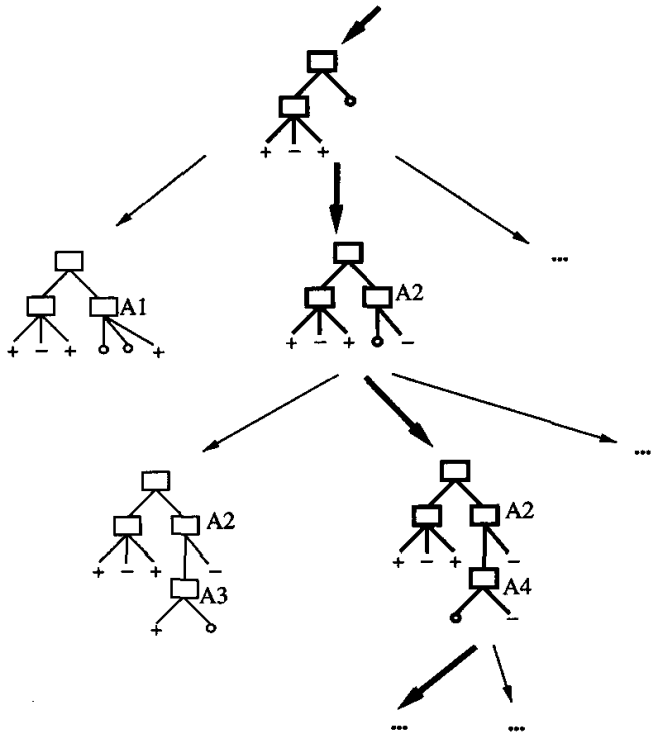


FIGURE 3.5
Hypothesis space search by ID3. ID3 searches through the space of possible decision trees from simplest to increasingly complex, guided by the information gain heuristic.

explicitly representing all consistent hypotheses. For example, it does not have the ability to determine how many alternative decision trees are consistent with the available training data, or to pose new instance queries that optimally resolve among these competing hypotheses.

- ID3 in its pure form performs no backtracking in its search. Once it selects an attribute to test at a particular level in the tree, it never backtracks to reconsider this choice. Therefore, it is susceptible to the usual risks of hill-climbing search without backtracking: converging to locally optimal solutions that are not globally optimal. In the case of ID3, a locally optimal solution corresponds to the decision tree it selects along the single search path it explores. However, this locally optimal solution may be less desirable than trees that would have been encountered along a different branch of the search. Below we discuss an extension that adds a form of backtracking (post-pruning the decision tree).

- ID3 uses all training examples at each step in the search to make statistically based decisions regarding how to refine its current hypothesis. This contrasts with methods that make decisions incrementally, based on individual training examples (e.g., FIND-S or CANDIDATE-ELIMINATION). One advantage of using statistical properties of all the examples (e.g., information gain) is that the resulting search is much less sensitive to errors in individual training examples. ID3 can be easily extended to handle noisy training data by modifying its termination criterion to accept hypotheses that imperfectly fit the training data.

3.6 INDUCTIVE BIAS IN DECISION TREE LEARNING

What is the policy by which ID3 generalizes from observed training examples to classify unseen instances? In other words, what is its inductive bias? Recall from Chapter 2 that inductive bias is the set of assumptions that, together with the training data, deductively justify the classifications assigned by the learner to future instances.

Given a collection of training examples, there are typically many decision trees consistent with these examples. Describing the inductive bias of ID3 therefore consists of describing the basis by which it chooses one of these consistent hypotheses over the others. Which of these decision trees does ID3 choose? It chooses the first acceptable tree it encounters in its simple-to-complex, hill-climbing search through the space of possible trees. Roughly speaking, then, the ID3 search strategy (a) selects in favor of shorter trees over longer ones, and (b) selects trees that place the attributes with highest information gain closest to the root. Because of the subtle interaction between the attribute selection heuristic used by ID3 and the particular training examples it encounters, it is difficult to characterize precisely the inductive bias exhibited by ID3. However, we can approximately characterize its bias as a preference for short decision trees over complex trees.

Approximate inductive bias of ID3: Shorter trees are preferred over larger trees.

In fact, one could imagine an algorithm similar to ID3 that exhibits precisely this inductive bias. Consider an algorithm that begins with the empty tree and searches *breadth first* through progressively more complex trees, first considering all trees of depth 1, then all trees of depth 2, etc. Once it finds a decision tree consistent with the training data, it returns the smallest consistent tree at that search depth (e.g., the tree with the fewest nodes). Let us call this breadth-first search algorithm BFS-ID3. BFS-ID3 finds a shortest decision tree and thus exhibits precisely the bias "shorter trees are preferred over longer trees." ID3 can be viewed as an efficient approximation to BFS-ID3, using a greedy heuristic search to attempt to find the shortest tree without conducting the entire breadth-first search through the hypothesis space.

Because ID3 uses the information gain heuristic and a hill climbing strategy, it exhibits a more complex bias than BFS-ID3. In particular, it does not always find the shortest consistent tree, and it is biased to favor trees that place attributes with high information gain closest to the root.

A closer approximation to the inductive bias of ID3: Shorter trees are preferred over longer trees. Trees that place high information gain attributes close to the root are preferred over those that do not.

3.6.1 Restriction Biases and Preference Biases

There is an interesting difference between the types of inductive bias exhibited by ID3 and by the CANDIDATE-ELIMINATION algorithm discussed in Chapter 2.

Consider the difference between the hypothesis space search in these two approaches:

- ID3 searches a *complete* hypothesis space (i.e., one capable of expressing any finite discrete-valued function). It searches *incompletely* through this space, from simple to complex hypotheses, until its termination condition is met (e.g., until it finds a hypothesis consistent with the data). Its inductive bias is solely a consequence of the ordering of hypotheses by its search strategy. Its hypothesis space introduces no additional bias.
- The version space CANDIDATE-ELIMINATION algorithm searches an *incomplete* hypothesis space (i.e., one that can express only a subset of the potentially teachable concepts). It searches this space *completely*, finding every hypothesis consistent with the training data. Its inductive bias is solely a consequence of the expressive power of its hypothesis representation. Its search strategy introduces no additional bias.

In brief, the inductive bias of ID3 follows from its *search strategy*, whereas the inductive bias of the CANDIDATE-ELIMINATION algorithm follows from the definition of its *search space*.

The inductive bias of ID3 is thus a *preference* for certain hypotheses over others (e.g., for shorter hypotheses), with no hard restriction on the hypotheses that can be eventually enumerated. This form of bias is typically called a *preference bias* (or, alternatively, a *search bias*). In contrast, the bias of the CANDIDATE-ELIMINATION algorithm is in the form of a categorical *restriction* on the set of hypotheses considered. This form of bias is typically called a *restriction bias* (or, alternatively, a *language bias*).

Given that some form of inductive bias is required in order to generalize beyond the training data (see Chapter 2), which type of inductive bias shall we prefer; a preference bias or restriction bias?

Typically, a preference bias is more desirable than a restriction bias, because it allows the learner to work within a complete hypothesis space that is assured to contain the unknown target function. In contrast, a restriction bias that strictly limits the set of potential hypotheses is generally less desirable, because it introduces the possibility of excluding the unknown target function altogether.

Whereas ID3 exhibits a purely preference bias and CANDIDATE-ELIMINATION a purely restriction bias, some learning systems combine both. Consider, for example, the program described in Chapter 1 for learning a numerical evaluation function for game playing. In this case, the learned evaluation function is represented by a linear combination of a fixed set of board features, and the learning algorithm adjusts the parameters of this linear combination to best fit the available training data. In this case, the decision to use a linear function to represent the evaluation function introduces a restriction bias (nonlinear evaluation functions cannot be represented in this form). At the same time, the choice of a particular parameter tuning method (the LMS algorithm in this case) introduces a preference bias stemming from the ordered search through the space of all possible parameter values.

3.6.2 Why Prefer Short Hypotheses?

Is ID3's inductive bias favoring shorter decision trees a sound basis for generalizing beyond the training data? Philosophers and others have debated this question for centuries, and the debate remains unresolved to this day. William of Occam was one of the first to discuss[†] the question, around the year 1320, so this bias often goes by the name of Occam's razor.

Occam's razor: Prefer the simplest hypothesis that fits the data.

Of course giving an inductive bias a name does not justify it. Why should one prefer simpler hypotheses? Notice that scientists sometimes appear to follow this inductive bias. Physicists, for example, prefer simple explanations for the motions of the planets, over more complex explanations. Why? One argument is that because there are fewer short hypotheses than long ones (based on straightforward combinatorial arguments), it is less likely that one will find a short hypothesis that coincidentally fits the training data. In contrast there are often many very complex hypotheses that fit the current training data but fail to generalize correctly to subsequent data. Consider decision tree hypotheses, for example. There are many more 500-node decision trees than 5-node decision trees. Given a small set of 20 training examples, we might expect to be able to find many 500-node decision trees consistent with these, whereas we would be more surprised if a 5-node decision tree could perfectly fit this data. We might therefore believe the 5-node tree is less likely to be a statistical coincidence and prefer this hypothesis over the 500-node hypothesis.

Upon closer examination, it turns out there is a major difficulty with the above argument. By the same reasoning we could have argued that one should prefer decision trees containing exactly 17 leaf nodes with 11 nonleaf nodes, that use the decision attribute A_1 at the root, and test attributes A_2 through A_{11}, in numerical order. There are relatively few such trees, and we might argue (by the same reasoning as above) that our a priori chance of finding one consistent with an arbitrary set of data is therefore small. The difficulty here is that there are very many small sets of hypotheses that one can define—most of them rather arcane. Why should we believe that the small set of hypotheses consisting of decision trees with *short descriptions* should be any more relevant than the multitude of other small sets of hypotheses that we might define?

A second problem with the above argument for Occam's razor is that the size of a hypothesis is determined by the particular representation used *internally* by the learner. Two learners using different internal representations could therefore arrive at different hypotheses, both justifying their contradictory conclusions by Occam's razor! For example, the function represented by the learned decision tree in Figure 3.1 could be represented as a tree with just one decision node, by a learner that uses the boolean attribute XYZ, where we define the attribute XYZ to

[†] Apparently while shaving.

be true for instances that are classified positive by the decision tree in Figure 3.1 and false otherwise. Thus, two learners, both applying Occam's razor, would generalize in different ways if one used the XYZ attribute to describe its examples and the other used only the attributes *Outlook, Temperature, Humidity*, and *Wind.*

This last argument shows that Occam's razor will produce two different hypotheses from the same training examples when it is applied by two learners that perceive these examples in terms of different internal representations. On this basis we might be tempted to reject Occam's razor altogether. However, consider the following scenario that examines the question of which internal representations might arise from a process of evolution and natural selection. Imagine a population of artificial learning agents created by a simulated evolutionary process involving reproduction, mutation, and natural selection of these agents. Let us assume that this evolutionary process can alter the perceptual systems of these agents from generation to generation, thereby changing the internal attributes by which they perceive their world. For the sake of argument, let us also assume that the learning agents employ a fixed learning algorithm (say ID3) that cannot be altered by evolution. It is reasonable to assume that over time evolution will produce internal representation that make these agents increasingly successful within their environment. Assuming that the success of an agent depends highly on its ability to generalize accurately, we would therefore expect evolution to develop internal representations that work well with whatever learning algorithm and inductive bias is present. If the species of agents employs a learning algorithm whose inductive bias is Occam's razor, then we expect evolution to produce internal representations for which Occam's razor is a successful strategy. The essence of the argument here is that evolution will create internal representations that make the learning algorithm's inductive bias a self-fulfilling prophecy, simply because it can alter the representation easier than it can alter the learning algorithm.

For now, we leave the debate regarding Occam's razor. We will revisit it in Chapter 6, where we discuss the Minimum Description Length principle, a version of Occam's razor that can be interpreted within a Bayesian framework.

3.7 ISSUES IN DECISION TREE LEARNING

Practical issues in learning decision trees include determining how deeply to grow the decision tree, handling continuous attributes, choosing an appropriate attribute selection measure, handling training data with missing attribute values, handling attributes with differing costs, and improving computational efficiency. Below we discuss each of these issues and extensions to the basic ID3 algorithm that address them. ID3 has itself been extended to address most of these issues, with the resulting system renamed C4.5 (Quinlan 1993).

3.7.1 Avoiding Overfitting the Data

The algorithm described in Table 3.1 grows each branch of the tree just deeply enough to perfectly classify the training examples. While this is sometimes a

reasonable strategy, in fact it can lead to difficulties when there is noise in the data, or when the number of training examples is too small to produce a representative sample of the true target function. In either of these cases, this simple algorithm can produce trees that *overfit* the training examples.

We will say that a hypothesis overfits the training examples if some other hypothesis that fits the training examples less well actually performs better over the entire distribution of instances (i.e., including instances beyond the training set).

> ***Definition***: Given a hypothesis space H, a hypothesis $h \in H$ is said to **overfit** the training data if there exists some alternative hypothesis $h' \in H$, such that h has smaller error than h' over the training examples, but h' has a smaller error than h over the entire distribution of instances.

Figure 3.6 illustrates the impact of overfitting in a typical application of decision tree learning. In this case, the ID3 algorithm is applied to the task of learning which medical patients have a form of diabetes. The horizontal axis of this plot indicates the total number of nodes in the decision tree, as the tree is being constructed. The vertical axis indicates the accuracy of predictions made by the tree. The solid line shows the accuracy of the decision tree over the training examples, whereas the broken line shows accuracy measured over an independent set of test examples (not included in the training set). Predictably, the accuracy of the tree over the training examples increases monotonically as the tree is grown. However, the accuracy measured over the independent test examples first increases, then decreases. As can be seen, once the tree size exceeds approximately 25 nodes,

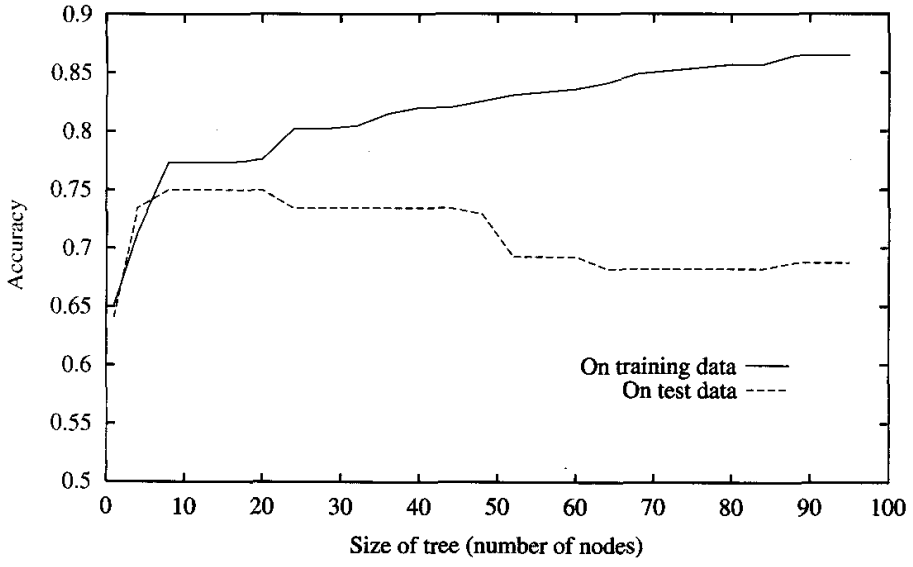


FIGURE 3.6
Overfitting in decision tree learning. As ID3 adds new nodes to grow the decision tree, the accuracy of the tree measured over the training examples increases monotonically. However, when measured over a set of test examples independent of the training examples, accuracy first increases, then decreases. Software and data for experimenting with variations on this plot are available on the World Wide Web at http://www.cs.cmu.edu/~tom/mlbook.html.

further elaboration of the tree decreases its accuracy over the test examples despite increasing its accuracy on the training examples.

How can it be possible for tree h to fit the training examples better than h', but for it to perform more poorly over subsequent examples? One way this can occur is when the training examples contain random errors or noise. To illustrate, consider the effect of adding the following positive training example, incorrectly labeled as negative, to the (otherwise correct) examples in Table 3.2.

$$\langle Outlook = Sunny, Temperature = Hot, Humidity = Normal,$$
$$Wind = Strong, PlayTennis = No \rangle$$

Given the original error-free data, ID3 produces the decision tree shown in Figure 3.1. However, the addition of this incorrect example will now cause ID3 to construct a more complex tree. In particular, the new example will be sorted into the second leaf node from the left in the learned tree of Figure 3.1, along with the previous positive examples D9 and D11. Because the new example is labeled as a negative example, ID3 will search for further refinements to the tree below this node. Of course as long as the new erroneous example differs in some arbitrary way from the other examples affiliated with this node, ID3 will succeed in finding a new decision attribute to separate out this new example from the two previous positive examples at this tree node. The result is that ID3 will output a decision tree (h) that is more complex than the original tree from Figure 3.1 (h'). Of course h will fit the collection of training examples perfectly, whereas the simpler h' will not. However, given that the new decision node is simply a consequence of fitting the noisy training example, we expect h to outperform h' over subsequent data drawn from the same instance distribution.

The above example illustrates how random noise in the training examples can lead to overfitting. In fact, overfitting is possible even when the training data are noise-free, especially when small numbers of examples are associated with leaf nodes. In this case, it is quite possible for coincidental regularities to occur, in which some attribute happens to partition the examples very well, despite being unrelated to the actual target function. Whenever such coincidental regularities exist, there is a risk of overfitting.

Overfitting is a significant practical difficulty for decision tree learning and many other learning methods. For example, in one experimental study of ID3 involving five different learning tasks with noisy, nondeterministic data (Mingers 1989b), overfitting was found to decrease the accuracy of learned decision trees by 10–25% on most problems.

There are several approaches to avoiding overfitting in decision tree learning. These can be grouped into two classes:

- approaches that stop growing the tree earlier, before it reaches the point where it perfectly classifies the training data,
- approaches that allow the tree to overfit the data, and then post-prune the tree.

Although the first of these approaches might seem more direct, the second approach of post-pruning overfit trees has been found to be more successful in practice. This is due to the difficulty in the first approach of estimating precisely when to stop growing the tree.

Regardless of whether the correct tree size is found by stopping early or by post-pruning, a key question is what criterion is to be used to determine the correct final tree size. Approaches include:

- Use a separate set of examples, distinct from the training examples, to evaluate the utility of post-pruning nodes from the tree.
- Use all the available data for training, but apply a statistical test to estimate whether expanding (or pruning) a particular node is likely to produce an improvement beyond the training set. For example, Quinlan (1986) uses a chi-square test to estimate whether further expanding a node is likely to improve performance over the entire instance distribution, or only on the current sample of training data.
- Use an explicit measure of the complexity for encoding the training examples and the decision tree, halting growth of the tree when this encoding size is minimized. This approach, based on a heuristic called the Minimum Description Length principle, is discussed further in Chapter 6, as well as in Quinlan and Rivest (1989) and Mehta et al. (1995).

The first of the above approaches is the most common and is often referred to as a *training and validation set* approach. We discuss the two main variants of this approach below. In this approach, the available data are separated into two sets of examples: a *training set*, which is used to form the learned hypothesis, and a separate *validation set*, which is used to evaluate the accuracy of this hypothesis over subsequent data and, in particular, to evaluate the impact of pruning this hypothesis. The motivation is this: Even though the learner may be misled by random errors and coincidental regularities within the training set, the validation set is unlikely to exhibit the same random fluctuations. Therefore, the validation set can be expected to provide a safety check against overfitting the spurious characteristics of the training set. Of course, it is important that the validation set be large enough to itself provide a statistically significant sample of the instances. One common heuristic is to withhold one-third of the available examples for the validation set, using the other two-thirds for training.

3.7.1.1 REDUCED ERROR PRUNING

How exactly might we use a validation set to prevent overfitting? One approach, called *reduced-error pruning* (Quinlan 1987), is to consider each of the decision nodes in the tree to be candidates for pruning. Pruning a decision node consists of removing the subtree rooted at that node, making it a leaf node, and assigning it the most common classification of the training examples affiliated with that node. Nodes are removed only if the resulting pruned tree performs no worse than the

original over the validation set. This has the effect that any leaf node added due to coincidental regularities in the training set is likely to be pruned because these same coincidences are unlikely to occur in the validation set. Nodes are pruned iteratively, always choosing the node whose removal most increases the decision tree accuracy over the validation set. Pruning of nodes continues until further pruning is harmful (i.e., decreases accuracy of the tree over the validation set).

The impact of reduced-error pruning on the accuracy of the decision tree is illustrated in Figure 3.7. As in Figure 3.6, the accuracy of the tree is shown measured over both training examples and test examples. The additional line in Figure 3.7 shows accuracy over the test examples as the tree is pruned. When pruning begins, the tree is at its maximum size and lowest accuracy over the test set. As pruning proceeds, the number of nodes is reduced and accuracy over the test set increases. Here, the available data has been split into three subsets: the training examples, the validation examples used for pruning the tree, and a set of test examples used to provide an unbiased estimate of accuracy over future unseen examples. The plot shows accuracy over the training and test sets. Accuracy over the validation set used for pruning is not shown.

Using a separate set of data to guide pruning is an effective approach provided a large amount of data is available. The major drawback of this approach is that when data is limited, withholding part of it for the validation set reduces even further the number of examples available for training. The following section presents an alternative approach to pruning that has been found useful in many practical situations where data is limited. Many additional techniques have been proposed as well, involving partitioning the available data several different times in

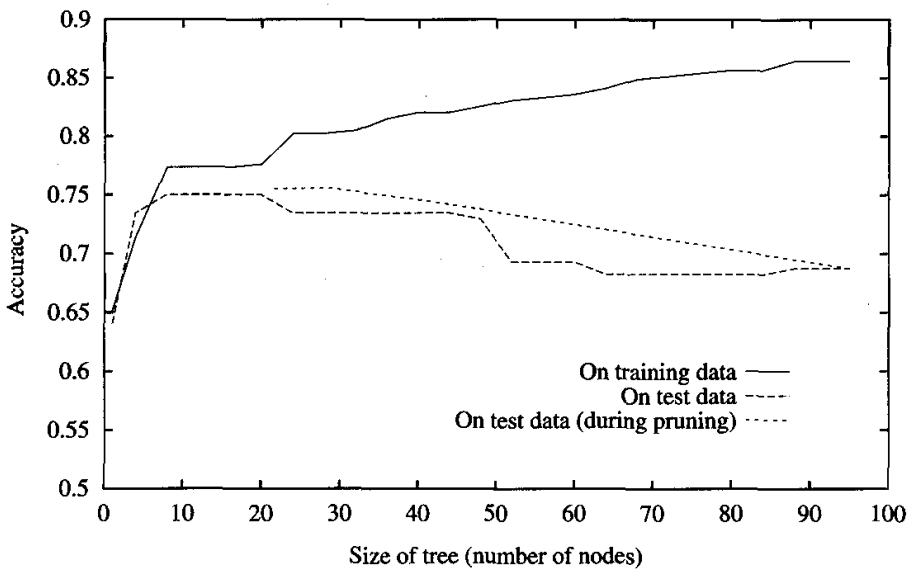


FIGURE 3.7
Effect of reduced-error pruning in decision tree learning. This plot shows the same curves of training and test set accuracy as in Figure 3.6. In addition, it shows the impact of reduced error pruning of the tree produced by ID3. Notice the increase in accuracy over the test set as nodes are pruned from the tree. Here, the validation set used for pruning is distinct from both the training and test sets.

multiple ways, then averaging the results. Empirical evaluations of alternative tree pruning methods are reported by Mingers (1989b) and by Malerba et al. (1995).

3.7.1.2 RULE POST-PRUNING

In practice, one quite successful method for finding high accuracy hypotheses is a technique we shall call *rule post-pruning*. A variant of this pruning method is used by C4.5 (Quinlan 1993), which is an outgrowth of the original ID3 algorithm. Rule post-pruning involves the following steps:

1. Infer the decision tree from the training set, growing the tree until the training data is fit as well as possible and allowing overfitting to occur.
2. Convert the learned tree into an equivalent set of rules by creating one rule for each path from the root node to a leaf node.
3. Prune (generalize) each rule by removing any preconditions that result in improving its estimated accuracy.
4. Sort the pruned rules by their estimated accuracy, and consider them in this sequence when classifying subsequent instances.

To illustrate, consider again the decision tree in Figure 3.1. In rule post-pruning, one rule is generated for each leaf node in the tree. Each attribute test along the path from the root to the leaf becomes a rule antecedent (precondition) and the classification at the leaf node becomes the rule consequent (postcondition). For example, the leftmost path of the tree in Figure 3.1 is translated into the rule

$$\text{IF} \quad (Outlook = Sunny) \wedge (Humidity = High)$$

$$\text{THEN} \quad PlayTennis = No$$

Next, each such rule is pruned by removing any antecedent, or precondition, whose removal does not worsen its estimated accuracy. Given the above rule, for example, rule post-pruning would consider removing the preconditions $(Outlook = Sunny)$ and $(Humidity = High)$. It would select whichever of these pruning steps produced the greatest improvement in estimated rule accuracy, then consider pruning the second precondition as a further pruning step. No pruning step is performed if it reduces the estimated rule accuracy.

As noted above, one method to estimate rule accuracy is to use a validation set of examples disjoint from the training set. Another method, used by C4.5, is to evaluate performance based on the training set itself, using a pessimistic estimate to make up for the fact that the training data gives an estimate biased in favor of the rules. More precisely, C4.5 calculates its pessimistic estimate by calculating the rule accuracy over the training examples to which it applies, then calculating the standard deviation in this estimated accuracy assuming a binomial distribution. For a given confidence level, the lower-bound estimate is then taken as the measure of rule performance (e.g., for a 95% confidence interval, rule accuracy is pessimistically estimated by the observed accuracy over the training

set, minus 1.96 times the estimated standard deviation). The net effect is that for large data sets, the pessimistic estimate is very close to the observed accuracy (e.g., the standard deviation is very small), whereas it grows further from the observed accuracy as the size of the data set decreases. Although this heuristic method is not statistically valid, it has nevertheless been found useful in practice. See Chapter 5 for a discussion of statistically valid approaches to estimating means and confidence intervals.

Why convert the decision tree to rules before pruning? There are three main advantages.

- Converting to rules allows distinguishing among the different contexts in which a decision node is used. Because each distinct path through the decision tree node produces a distinct rule, the pruning decision regarding that attribute test can be made differently for each path. In contrast, if the tree itself were pruned, the only two choices would be to remove the decision node completely, or to retain it in its original form.
- Converting to rules removes the distinction between attribute tests that occur near the root of the tree and those that occur near the leaves. Thus, we avoid messy bookkeeping issues such as how to reorganize the tree if the root node is pruned while retaining part of the subtree below this test.
- Converting to rules improves readability. Rules are often easier for people to understand.

3.7.2 Incorporating Continuous-Valued Attributes

Our initial definition of ID3 is restricted to attributes that take on a discrete set of values. First, the target attribute whose value is predicted by the learned tree must be discrete valued. Second, the attributes tested in the decision nodes of the tree must also be discrete valued. This second restriction can easily be removed so that continuous-valued decision attributes can be incorporated into the learned tree. This can be accomplished by dynamically defining new discrete-valued attributes that partition the continuous attribute value into a discrete set of intervals. In particular, for an attribute A that is continuous-valued, the algorithm can dynamically create a new boolean attribute A_c that is true if $A < c$ and false otherwise. The only question is how to select the best value for the threshold c.

As an example, suppose we wish to include the continuous-valued attribute *Temperature* in describing the training example days in the learning task of Table 3.2. Suppose further that the training examples associated with a particular node in the decision tree have the following values for *Temperature* and the target attribute *PlayTennis*.

Temperature:	40	48	60	72	80	90
PlayTennis:	No	No	Yes	Yes	Yes	No

What threshold-based boolean attribute should be defined based on *Temperature*? Clearly, we would like to pick a threshold, c, that produces the greatest information gain. By sorting the examples according to the continuous attribute A, then identifying adjacent examples that differ in their target classification, we can generate a set of candidate thresholds midway between the corresponding values of A. It can be shown that the value of c that maximizes information gain must always lie at such a boundary (Fayyad 1991). These candidate thresholds can then be evaluated by computing the information gain associated with each. In the current example, there are two candidate thresholds, corresponding to the values of *Temperature* at which the value of *PlayTennis* changes: (48 + 60)/2, and (80 + 90)/2. The information gain can then be computed for each of the candidate attributes, $Temperature_{>54}$ and $Temperature_{>85}$, and the best can be selected ($Temperature_{>54}$). This dynamically created boolean attribute can then compete with the other discrete-valued candidate attributes available for growing the decision tree. Fayyad and Irani (1993) discuss an extension to this approach that splits the continuous attribute into multiple intervals rather than just two intervals based on a single threshold. Utgoff and Brodley (1991) and Murthy et al. (1994) discuss approaches that define features by thresholding linear combinations of several continuous-valued attributes.

3.7.3 Alternative Measures for Selecting Attributes

There is a natural bias in the information gain measure that favors attributes with many values over those with few values. As an extreme example, consider the attribute *Date*, which has a very large number of possible values (e.g., March 4, 1979). If we were to add this attribute to the data in Table 3.2, it would have the highest information gain of any of the attributes. This is because *Date* alone perfectly predicts the target attribute over the training data. Thus, it would be selected as the decision attribute for the root node of the tree and lead to a (quite broad) tree of depth one, which perfectly classifies the training data. Of course, this decision tree would fare poorly on subsequent examples, because it is not a useful predictor despite the fact that it perfectly separates the training data.

What is wrong with the attribute *Date*? Simply put, it has so many possible values that it is bound to separate the training examples into very small subsets. Because of this, it will have a very high information gain relative to the training examples, despite being a very poor predictor of the target function over unseen instances.

One way to avoid this difficulty is to select decision attributes based on some measure other than information gain. One alternative measure that has been used successfully is the *gain ratio* (Quinlan 1986). The gain ratio measure penalizes attributes such as *Date* by incorporating a term, called *split information*, that is sensitive to how broadly and uniformly the attribute splits the data:

$$SplitInformation(S, A) \equiv -\sum_{i=1}^{c} \frac{|S_i|}{|S|} \log_2 \frac{|S_i|}{|S|} \tag{3.5}$$

where S_1 through S_c are the c subsets of examples resulting from partitioning S by the c-valued attribute A. Note that *SplitInformation* is actually the entropy of S with respect to the values of attribute A. This is in contrast to our previous uses of entropy, in which we considered only the entropy of S with respect to the target attribute whose value is to be predicted by the learned tree.

The *GainRatio* measure is defined in terms of the earlier *Gain* measure, as well as this *SplitInformation*, as follows

$$GainRatio(S, A) \equiv \frac{Gain(S, A)}{SplitInformation(S, A)} \tag{3.6}$$

Notice that the *SplitInformation* term discourages the selection of attributes with many uniformly distributed values. For example, consider a collection of n examples that are completely separated by attribute A (e.g., *Date*). In this case, the *SplitInformation* value will be $\log_2 n$. In contrast, a boolean attribute B that splits the same n examples exactly in half will have *SplitInformation* of 1. If attributes A and B produce the same information gain, then clearly B will score higher according to the *GainRatio* measure.

One practical issue that arises in using *GainRatio* in place of *Gain* to select attributes is that the denominator can be zero or very small when $|S_i| \approx |S|$ for one of the S_i. This either makes the *GainRatio* undefined or very large for attributes that happen to have the same value for nearly all members of S. To avoid selecting attributes purely on this basis, we can adopt some heuristic such as first calculating the *Gain* of each attribute, then applying the *GainRatio* test only considering those attributes with above average *Gain* (Quinlan 1986).

An alternative to the *GainRatio*, designed to directly address the above difficulty, is a distance-based measure introduced by Lopez de Mantaras (1991). This measure is based on defining a distance metric between partitions of the data. Each attribute is evaluated based on the distance between the data partition it creates and the perfect partition (i.e., the partition that perfectly classifies the training data). The attribute whose partition is closest to the perfect partition is chosen. Lopez de Mantaras (1991) defines this distance measure, proves that it is not biased toward attributes with large numbers of values, and reports experimental studies indicating that the predictive accuracy of the induced trees is not significantly different from that obtained with the *Gain* and *GainRatio* measures. However, this distance measure avoids the practical difficulties associated with the *GainRatio* measure, and in his experiments it produces significantly smaller trees in the case of data sets whose attributes have very different numbers of values.

A variety of other selection measures have been proposed as well (e.g., see Breiman et al. 1984; Mingers 1989a; Kearns and Mansour 1996; Dietterich et al. 1996). Mingers (1989a) provides an experimental analysis of the relative effectiveness of several selection measures over a variety of problems. He reports significant differences in the sizes of the unpruned trees produced by the different selection measures. However, in his experimental domains the choice of attribute selection measure appears to have a smaller impact on final accuracy than does the extent and method of post-pruning.

3.7.4 Handling Training Examples with Missing Attribute Values

In certain cases, the available data may be missing values for some attributes. For example, in a medical domain in which we wish to predict patient outcome based on various laboratory tests, it may be that the lab test *Blood-Test-Result* is available only for a subset of the patients. In such cases, it is common to estimate the missing attribute value based on other examples for which this attribute has a known value.

Consider the situation in which $Gain(S, A)$ is to be calculated at node n in the decision tree to evaluate whether the attribute A is the best attribute to test at this decision node. Suppose that $\langle x, c(x)\rangle$ is one of the training examples in S and that the value $A(x)$ is unknown.

One strategy for dealing with the missing attribute value is to assign it the value that is most common among training examples at node n. Alternatively, we might assign it the most common value among examples at node n that have the classification $c(x)$. The elaborated training example using this estimated value for $A(x)$ can then be used directly by the existing decision tree learning algorithm. This strategy is examined by Mingers (1989a).

A second, more complex procedure is to assign a probability to each of the possible values of A rather than simply assigning the most common value to $A(x)$. These probabilities can be estimated again based on the observed frequencies of the various values for A among the examples at node n. For example, given a boolean attribute A, if node n contains six known examples with $A = 1$ and four with $A = 0$, then we would say the probability that $A(x) = 1$ is 0.6, and the probability that $A(x) = 0$ is 0.4. A fractional 0.6 of instance x is now distributed down the branch for $A = 1$, and a fractional 0.4 of x down the other tree branch. These fractional examples are used for the purpose of computing information $Gain$ and can be further subdivided at subsequent branches of the tree if a second missing attribute value must be tested. This same fractioning of examples can also be applied after learning, to classify new instances whose attribute values are unknown. In this case, the classification of the new instance is simply the most probable classification, computed by summing the weights of the instance fragments classified in different ways at the leaf nodes of the tree. This method for handling missing attribute values is used in C4.5 (Quinlan 1993).

3.7.5 Handling Attributes with Differing Costs

In some learning tasks the instance attributes may have associated costs. For example, in learning to classify medical diseases we might describe patients in terms of attributes such as *Temperature*, *BiopsyResult*, *Pulse*, *BloodTestResults*, etc. These attributes vary significantly in their costs, both in terms of monetary cost and cost to patient comfort. In such tasks, we would prefer decision trees that use low-cost attributes where possible, relying on high-cost attributes only when needed to produce reliable classifications.

ID3 can be modified to take into account attribute costs by introducing a cost term into the attribute selection measure. For example, we might divide the $Gain$

by the cost of the attribute, so that lower-cost attributes would be preferred. While such cost-sensitive measures do not guarantee finding an optimal cost-sensitive decision tree, they do bias the search in favor of low-cost attributes.

Tan and Schlimmer (1990) and Tan (1993) describe one such approach and apply it to a robot perception task in which the robot must learn to classify different objects according to how they can be grasped by the robot's manipulator. In this case the attributes correspond to different sensor readings obtained by a movable sonar on the robot. Attribute cost is measured by the number of seconds required to obtain the attribute value by positioning and operating the sonar. They demonstrate that more efficient recognition strategies are learned, without sacrificing classification accuracy, by replacing the information gain attribute selection measure by the following measure

$$\frac{Gain^2(S, A)}{Cost(A)}$$

Nunez (1988) describes a related approach and its application to learning medical diagnosis rules. Here the attributes are different symptoms and laboratory tests with differing costs. His system uses a somewhat different attribute selection measure

$$\frac{2^{Gain(S,A)} - 1}{(Cost(A) + 1)^w}$$

where $w \in [0, 1]$ is a constant that determines the relative importance of cost versus information gain. Nunez (1991) presents an empirical comparison of these two approaches over a range of tasks.

3.8 SUMMARY AND FURTHER READING

The main points of this chapter include:

- Decision tree learning provides a practical method for concept learning and for learning other discrete-valued functions. The ID3 family of algorithms infers decision trees by growing them from the root downward, greedily selecting the next best attribute for each new decision branch added to the tree.
- ID3 searches a complete hypothesis space (i.e., the space of decision trees can represent any discrete-valued function defined over discrete-valued instances). It thereby avoids the major difficulty associated with approaches that consider only restricted sets of hypotheses: that the target function might not be present in the hypothesis space.
- The inductive bias implicit in ID3 includes a *preference* for smaller trees; that is, its search through the hypothesis space grows the tree only as large as needed in order to classify the available training examples.
- Overfitting the training data is an important issue in decision tree learning. Because the training examples are only a sample of all possible instances,

it is possible to add branches to the tree that improve performance on the training examples while decreasing performance on other instances outside this set. Methods for post-pruning the decision tree are therefore important to avoid overfitting in decision tree learning (and other inductive inference methods that employ a preference bias).

- A large variety of extensions to the basic ID3 algorithm has been developed by different researchers. These include methods for post-pruning trees, handling real-valued attributes, accommodating training examples with missing attribute values, incrementally refining decision trees as new training examples become available, using attribute selection measures other than information gain, and considering costs associated with instance attributes.

Among the earliest work on decision tree learning is Hunt's Concept Learning System (CLS) (Hunt et al. 1966) and Friedman and Breiman's work resulting in the CART system (Friedman 1977; Breiman et al. 1984). Quinlan's ID3 system (Quinlan 1979, 1983) forms the basis for the discussion in this chapter. Other early work on decision tree learning includes ASSISTANT (Kononenko et al. 1984; Cestnik et al. 1987). Implementations of decision tree induction algorithms are now commercially available on many computer platforms.

For further details on decision tree induction, an excellent book by Quinlan (1993) discusses many practical issues and provides executable code for C4.5. Mingers (1989a) and Buntine and Niblett (1992) provide two experimental studies comparing different attribute-selection measures. Mingers (1989b) and Malerba et al. (1995) provide studies of different pruning strategies. Experiments comparing decision tree learning and other learning methods can be found in numerous papers, including (Dietterich et al. 1995; Fisher and McKusick 1989; Quinlan 1988a; Shavlik et al. 1991; Thrun et al. 1991; Weiss and Kapouleas 1989).

EXERCISES

3.1. Give decision trees to represent the following boolean functions:

(*a*) $A \wedge \neg B$

(*b*) $A \vee [B \wedge C]$

(*c*) $A \; XOR \; B$

(*d*) $[A \wedge B] \vee [C \wedge D]$

3.2. Consider the following set of training examples:

Instance	Classification	a_1	a_2
1	+	T	T
2	+	T	T
3	−	T	F
4	+	F	F
5	−	F	T
6	−	F	T

(*a*) What is the entropy of this collection of training examples with respect to the target function classification?

(*b*) What is the information gain of a_2 relative to these training examples?

3.3. True or false: If decision tree D2 is an elaboration of tree D1, then D1 is *more-general-than* D2. Assume D1 and D2 are decision trees representing arbitrary boolean functions, and that D2 is an elaboration of D1 if ID3 could extend D1 into D2. If true, give a proof; if false, a counterexample. (*More-general-than* is defined in Chapter 2.)

3.4. ID3 searches for just one consistent hypothesis, whereas the CANDIDATE-ELIMINATION algorithm finds all consistent hypotheses. Consider the correspondence between these two learning algorithms.

(*a*) Show the decision tree that would be learned by ID3 assuming it is given the four training examples for the *EnjoySport*? target concept shown in Table 2.1 of Chapter 2.

(*b*) What is the relationship between the learned decision tree and the version space (shown in Figure 2.3 of Chapter 2) that is learned from these same examples? Is the learned tree equivalent to one of the members of the version space?

(*c*) Add the following training example, and compute the new decision tree. This time, show the value of the information gain for each candidate attribute at each step in growing the tree.

Sky	*Air-Temp*	*Humidity*	*Wind*	*Water*	*Forecast*	*Enjoy-Sport?*
Sunny	*Warm*	*Normal*	*Weak*	*Warm*	*Same*	*No*

(*d*) Suppose we wish to design a learner that (like ID3) searches a space of decision tree hypotheses and (like CANDIDATE-ELIMINATION) finds all hypotheses consistent with the data. In short, we wish to apply the CANDIDATE-ELIMINATION algorithm to searching the space of decision tree hypotheses. Show the S and G sets that result from the first training example from Table 2.1. Note S must contain the most specific decision trees consistent with the data, whereas G must contain the most general. Show how the S and G sets are refined by the second training example (you may omit syntactically distinct trees that describe the same concept). What difficulties do you foresee in applying CANDIDATE-ELIMINATION to a decision tree hypothesis space?

REFERENCES

Breiman, L., Friedman, J. H., Olshen, R. A., & Stone, P. J. (1984). *Classification and regression trees.* Belmont, CA: Wadsworth International Group.

Brodley, C. E., & Utgoff, P. E. (1995). Multivariate decision trees. *Machine Learning*, 19, 45–77.

Buntine, W., & Niblett, T. (1992). A further comparison of splitting rules for decision-tree induction. *Machine Learning*, 8, 75–86.

Cestnik, B., Kononenko, I., & Bratko, I. (1987). ASSISTANT-86: A knowledge-elicitation tool for sophisticated users. In I. Bratko & N. Lavrač (Eds.), *Progress in machine learning.* Bled, Yugoslavia: Sigma Press.

Dietterich, T. G., Hild, H., & Bakiri, G. (1995). A comparison of ID3 and BACKPROPAGATION for English text-to-speech mapping. *Machine Learning*, 18(1), 51–80.

Dietterich, T. G., Kearns, M., & Mansour, Y. (1996). Applying the weak learning framework to understand and improve C4.5. *Proceedings of the 13th International Conference on Machine Learning* (pp. 96–104). San Francisco: Morgan Kaufmann.

Fayyad, U. M. (1991). *On the induction of decision trees for multiple concept learning,* (Ph.D. dissertation). EECS Department, University of Michigan.

Fayyad, U. M., & Irani, K. B. (1992). On the handling of continuous-valued attributes in decision tree generation. *Machine Learning*, 8, 87–102.

Fayyad, U. M., & Irani, K. B. (1993). Multi-interval discretization of continuous-valued attributes for classification learning. In R. Bajcsy (Ed.), *Proceedings of the 13th International Joint Conference on Artificial Intelligence* (pp. 1022–1027). Morgan-Kaufmann.

Fayyad, U. M., Weir, N., & Djorgovski, S. (1993). SKICAT: A machine learning system for automated cataloging of large scale sky surveys. *Proceedings of the Tenth International Conference on Machine Learning* (pp. 112–119). Amherst, MA: Morgan Kaufmann.

Fisher, D. H., and McKusick, K. B. (1989). An empirical comparison of ID3 and back-propagation. *Proceedings of the Eleventh International Joint Conference on AI* (pp. 788–793). Morgan Kaufmann.

Friedman, J. H. (1977). A recursive partitioning decision rule for non-parametric classification. *IEEE Transactions on Computers* (pp. 404–408).

Hunt, E. B. (1975). *Artificial Intelligence*. New York: Academic Press.

Hunt, E. B., Marin, J., & Stone, P. J. (1966). *Experiments in Induction*. New York: Academic Press.

Kearns, M., & Mansour, Y. (1996). On the boosting ability of top-down decision tree learning algorithms. *Proceedings of the 28th ACM Symposium on the Theory of Computing*. New York: ACM Press.

Kononenko, I., Bratko, I., & Roskar, E. (1984). *Experiments in automatic learning of medical diagnostic rules* (Technical report). Jozef Stefan Institute, Ljubljana, Yugoslavia.

Lopez de Mantaras, R. (1991). A distance-based attribute selection measure for decision tree induction. *Machine Learning*, 6(1), 81–92.

Malerba, D., Floriana, E., & Semeraro, G. (1995). A further comparison of simplification methods for decision tree induction. In D. Fisher & H. Lenz (Eds.), *Learning from data: AI and statistics*. Springer-Verlag.

Mehta, M., Rissanen, J., & Agrawal, R. (1995). MDL-based decision tree pruning. *Proceedings of the First International Conference on Knowledge Discovery and Data Mining* (pp. 216–221). Menlo Park, CA: AAAI Press.

Mingers, J. (1989a). An empirical comparison of selection measures for decision-tree induction. *Machine Learning*, 3(4), 319–342.

Mingers, J. (1989b). An empirical comparison of pruning methods for decision-tree induction. *Machine Learning*, 4(2), 227–243.

Murphy, P. M., & Pazzani, M. J. (1994). Exploring the decision forest: An empirical investigation of Occam's razor in decision tree induction. *Journal of Artificial Intelligence Research*, 1, 257–275.

Murthy, S. K., Kasif, S., & Salzberg, S. (1994). A system for induction of oblique decision trees. *Journal of Artificial Intelligence Research*, 2, 1–33.

Nunez, M. (1991). The use of background knowledge in decision tree induction. *Machine Learning*, 6(3), 231–250.

Pagallo, G., & Haussler, D. (1990). Boolean feature discovery in empirical learning. *Machine Learning*, 5, 71–100.

Quinlan, J. R. (1979). Discovering rules by induction from large collections of examples. In D. Michie (Ed.), *Expert systems in the micro electronic age*. Edinburgh Univ. Press.

Quinlan, J. R. (1983). Learning efficient classification procedures and their application to chess end games. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach*. San Mateo, CA: Morgan Kaufmann.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, 1(1), 81–106.

Quinlan, J. R. (1987). Rule induction with statistical data—a comparison with multiple regression. *Journal of the Operational Research Society*, 38, 347–352.

Quinlan, J.R. (1988). An empirical comparison of genetic and decision-tree classifiers. *Proceedings of the Fifth International Machine Learning Conference* (135–141). San Mateo, CA: Morgan Kaufmann.

Quinlan, J.R. (1988b). Decision trees and multi-valued attributes. In Hayes, Michie, & Richards (Eds.), *Machine Intelligence 11*, (pp. 305–318). Oxford, England: Oxford University Press.

Quinlan, J. R., & Rivest, R. (1989). *Information and Computation*, (80), 227–248.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. San Mateo, CA: Morgan Kaufmann.

Rissanen, J. (1983). A universal prior for integers and estimation by minimum description length. *Annals of Statistics 11* (2), 416–431.

Rivest, R. L. (1987). Learning decision lists. *Machine Learning*, 2(3), 229–246.

Schaffer, C. (1993). Overfitting avoidance as bias. *Machine Learning*, *10*, 113–152.

Shavlik, J. W., Mooney, R. J., & Towell, G. G. (1991). Symbolic and neural learning algorithms: an experimental comparison. *Machine Learning*, 6(2), 111–144.

Tan, M. (1993). Cost-sensitive learning of classification knowledge and its applications in robotics. *Machine Learning*, 13(1), 1–33.

Tan, M., & Schlimmer, J. C. (1990). Two case studies in cost-sensitive concept acquisition. *Proceedings of the AAAI-90.*

Thrun, S. B. et al. (1991). *The Monk's problems: A performance comparison of different learning algorithms,* (Technical report CMU-CS-91-197). Computer Science Department, Carnegie Mellon Univ., Pittsburgh, PA.

Turney, P. D. (1995). Cost-sensitive classification: empirical evaluation of a hybrid genetic decision tree induction algorithm. *Journal of AI Research*, 2, 369–409.

Utgoff, P. E. (1989). Incremental induction of decision trees. *Machine Learning*, 4(2), 161–186.

Utgoff, P. E., & Brodley, C. E. (1991). *Linear machine decision trees*, (COINS Technical Report 91-10). University of Massachusetts, Amherst, MA.

Weiss, S., & Kapouleas, I. (1989). An empirical comparison of pattern recognition, neural nets, and machine learning classification methods. *Proceedings of the Eleventh IJCAI*, (781–787), Morgan Kaufmann.

CHAPTER 4

ARTIFICIAL NEURAL NETWORKS

Artificial neural networks (ANNs) provide a general, practical method for learning real-valued, discrete-valued, and vector-valued functions from examples. Algorithms such as BACKPROPAGATION use gradient descent to tune network parameters to best fit a training set of input-output pairs. ANN learning is robust to errors in the training data and has been successfully applied to problems such as interpreting visual scenes, speech recognition, and learning robot control strategies.

4.1 INTRODUCTION

Neural network learning methods provide a robust approach to approximating real-valued, discrete-valued, and vector-valued target functions. For certain types of problems, such as learning to interpret complex real-world sensor data, artificial neural networks are among the most effective learning methods currently known. For example, the BACKPROPAGATION algorithm described in this chapter has proven surprisingly successful in many practical problems such as learning to recognize handwritten characters (LeCun et al. 1989), learning to recognize spoken words (Lang et al. 1990), and learning to recognize faces (Cottrell 1990). One survey of practical applications is provided by Rumelhart et al. (1994).

4.1.1 Biological Motivation

The study of artificial neural networks (ANNs) has been inspired in part by the observation that biological learning systems are built of very complex webs of interconnected neurons. In rough analogy, artificial neural networks are built out of a densely interconnected set of simple units, where each unit takes a number of real-valued inputs (possibly the outputs of other units) and produces a single real-valued output (which may become the input to many other units).

To develop a feel for this analogy, let us consider a few facts from neurobiology. The human brain, for example, is estimated to contain a densely interconnected network of approximately 10^{11} neurons, each connected, on average, to 10^4 others. Neuron activity is typically excited or inhibited through connections to other neurons. The fastest neuron switching times are known to be on the order of 10^{-3} seconds—quite slow compared to computer switching speeds of 10^{-10} seconds. Yet humans are able to make surprisingly complex decisions, surprisingly quickly. For example, it requires approximately 10^{-1} seconds to visually recognize your mother. Notice the sequence of neuron firings that can take place during this 10^{-1}-second interval cannot possibly be longer than a few hundred steps, given the switching speed of single neurons. This observation has led many to speculate that the information-processing abilities of biological neural systems must follow from highly parallel processes operating on representations that are distributed over many neurons. One motivation for ANN systems is to capture this kind of highly parallel computation based on distributed representations. Most ANN software runs on sequential machines emulating distributed processes, although faster versions of the algorithms have also been implemented on highly parallel machines and on specialized hardware designed specifically for ANN applications.

While ANNs are loosely motivated by biological neural systems, there are many complexities to biological neural systems that are not modeled by ANNs, and many features of the ANNs we discuss here are known to be inconsistent with biological systems. For example, we consider here ANNs whose individual units output a single constant value, whereas biological neurons output a complex time series of spikes.

Historically, two groups of researchers have worked with artificial neural networks. One group has been motivated by the goal of using ANNs to study and model biological learning processes. A second group has been motivated by the goal of obtaining highly effective machine learning algorithms, independent of whether these algorithms mirror biological processes. Within this book our interest fits the latter group, and therefore we will not dwell further on biological modeling. For more information on attempts to model biological systems using ANNs, see, for example, Churchland and Sejnowski (1992); Zornetzer et al. (1994); Gabriel and Moore (1990).

4.2 NEURAL NETWORK REPRESENTATIONS

A prototypical example of ANN learning is provided by Pomerleau's (1993) system ALVINN, which uses a learned ANN to steer an autonomous vehicle driving

at normal speeds on public highways. The input to the neural network is a 30×32 grid of pixel intensities obtained from a forward-pointed camera mounted on the vehicle. The network output is the direction in which the vehicle is steered. The ANN is trained to mimic the observed steering commands of a human driving the vehicle for approximately 5 minutes. ALVINN has used its learned networks to successfully drive at speeds up to 70 miles per hour and for distances of 90 miles on public highways (driving in the left lane of a divided public highway, with other vehicles present).

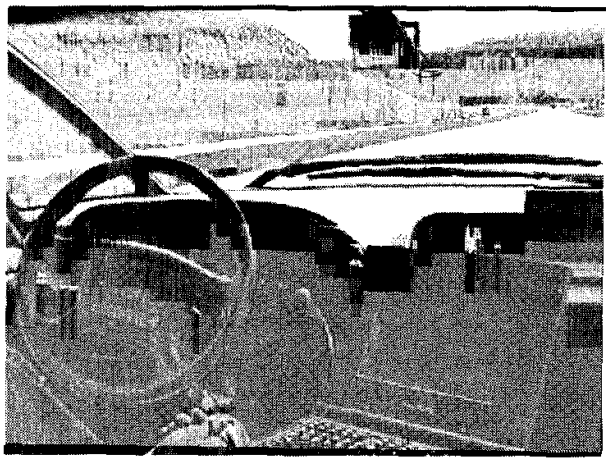
Figure 4.1 illustrates the neural network representation used in one version of the ALVINN system, and illustrates the kind of representation typical of many ANN systems. The network is shown on the left side of the figure, with the input camera image depicted below it. Each node (i.e., circle) in the network diagram corresponds to the output of a single network *unit*, and the lines entering the node from below are its inputs. As can be seen, there are four units that receive inputs directly from all of the 30×32 pixels in the image. These are called "hidden" units because their output is available only within the network and is not available as part of the global network output. Each of these four hidden units computes a single real-valued output based on a weighted combination of its 960 inputs. These hidden unit outputs are then used as inputs to a second layer of 30 "output" units. Each output unit corresponds to a particular steering direction, and the output values of these units determine which steering direction is recommended most strongly.

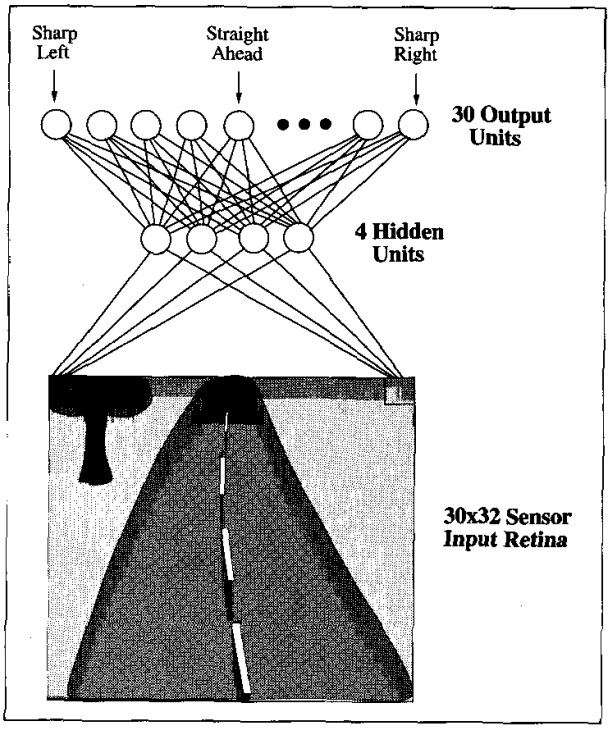
The diagrams on the right side of the figure depict the learned weight values associated with one of the four hidden units in this ANN. The large matrix of black and white boxes on the lower right depicts the weights from the 30×32 pixel inputs into the hidden unit. Here, a white box indicates a positive weight, a black box a negative weight, and the size of the box indicates the weight magnitude. The smaller rectangular diagram directly above the large matrix shows the weights from this hidden unit to each of the 30 output units.

The network structure of ALVINN is typical of many ANNs. Here the individual units are interconnected in layers that form a directed acyclic graph. In general, ANNs can be graphs with many types of structures—acyclic or cyclic, directed or undirected. This chapter will focus on the most common and practical ANN approaches, which are based on the BACKPROPAGATION algorithm. The BACKPROPAGATION algorithm assumes the network is a fixed structure that corresponds to a directed graph, possibly containing cycles. Learning corresponds to choosing a weight value for each edge in the graph. Although certain types of cycles are allowed, the vast majority of practical applications involve acyclic feed-forward networks, similar to the network structure used by ALVINN.

4.3 APPROPRIATE PROBLEMS FOR NEURAL NETWORK LEARNING

ANN learning is well-suited to problems in which the training data corresponds to noisy, complex sensor data, such as inputs from cameras and microphones.





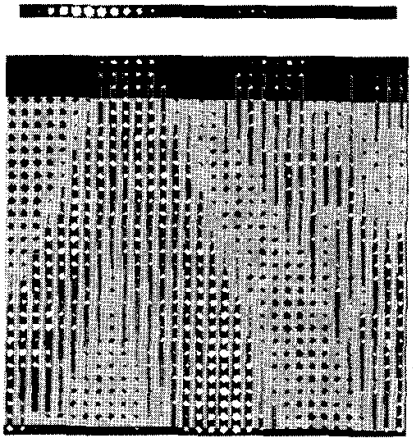


FIGURE 4.1
Neural network learning to steer an autonomous vehicle. The ALVINN system uses BACKPROPAGATION to learn to steer an autonomous vehicle (photo at top) driving at speeds up to 70 miles per hour. The diagram on the left shows how the image of a forward-mounted camera is mapped to 960 neural network inputs, which are fed forward to 4 hidden units, connected to 30 output units. Network outputs encode the commanded steering direction. The figure on the right shows weight values for one of the hidden units in this network. The 30 x 32 weights into the hidden unit are displayed in the large matrix, with white blocks indicating positive and black indicating negative weights. The weights from this hidden unit to the 30 output units are depicted by the smaller rectangular block directly above the large block. As can be seen from these output weights, activation of this particular hidden unit encourages a turn toward the left.

It is also applicable to problems for which more symbolic representations are often used, such as the decision tree learning tasks discussed in Chapter 3. In these cases ANN and decision tree learning often produce results of comparable accuracy. See Shavlik et al. (1991) and Weiss and Kapouleas (1989) for experimental comparisons of decision tree and ANN learning. The BACKPROPAGATION algorithm is the most commonly used ANN learning technique. It is appropriate for problems with the following characteristics:

- *Instances are represented by many attribute-value pairs.* The target function to be learned is defined over instances that can be described by a vector of predefined features, such as the pixel values in the ALVINN example. These input attributes may be highly correlated or independent of one another. Input values can be any real values.
- *The target function output may be discrete-valued, real-valued, or a vector of several real- or discrete-valued attributes.* For example, in the ALVINN system the output is a vector of 30 attributes, each corresponding to a recommendation regarding the steering direction. The value of each output is some real number between 0 and 1, which in this case corresponds to the confidence in predicting the corresponding steering direction. We can also train a single network to output both the steering command and suggested acceleration, simply by concatenating the vectors that encode these two output predictions.
- *The training examples may contain errors.* ANN learning methods are quite robust to noise in the training data.
- *Long training times are acceptable.* Network training algorithms typically require longer training times than, say, decision tree learning algorithms. Training times can range from a few seconds to many hours, depending on factors such as the number of weights in the network, the number of training examples considered, and the settings of various learning algorithm parameters.
- *Fast evaluation of the learned target function may be required.* Although ANN learning times are relatively long, evaluating the learned network, in order to apply it to a subsequent instance, is typically very fast. For example, ALVINN applies its neural network several times per second to continually update its steering command as the vehicle drives forward.
- *The ability of humans to understand the learned target function is not important.* The weights learned by neural networks are often difficult for humans to interpret. Learned neural networks are less easily communicated to humans than learned rules.

The rest of this chapter is organized as follows: We first consider several alternative designs for the primitive units that make up artificial neural networks (perceptrons, linear units, and sigmoid units), along with learning algorithms for training single units. We then present the BACKPROPAGATION algorithm for training

multilayer networks of such units and consider several general issues such as the representational capabilities of ANNs, nature of the hypothesis space search, overfitting problems, and alternatives to the BACKPROPAGATION algorithm. A detailed example is also presented applying BACKPROPAGATION to face recognition, and directions are provided for the reader to obtain the data and code to experiment further with this application.

4.4 PERCEPTRONS

One type of ANN system is based on a unit called a *perceptron*, illustrated in Figure 4.2. A perceptron takes a vector of real-valued inputs, calculates a linear combination of these inputs, then outputs a 1 if the result is greater than some threshold and -1 otherwise. More precisely, given inputs x_1 through x_n, the output $o(x_1, \ldots, x_n)$ computed by the perceptron is

$$o(x_1, \ldots, x_n) = \begin{cases} 1 \text{ if } w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n > 0 \\ -1 \text{ otherwise} \end{cases}$$

where each w_i is a real-valued constant, or *weight*, that determines the contribution of input x_i to the perceptron output. Notice the quantity $(-w_0)$ is a threshold that the weighted combination of inputs $w_1 x_1 + \cdots + w_n x_n$ must surpass in order for the perceptron to output a 1.

To simplify notation, we imagine an additional constant input $x_0 = 1$, allowing us to write the above inequality as $\sum_{i=0}^{n} w_i x_i > 0$, or in vector form as $\vec{w} \cdot \vec{x} > 0$. For brevity, we will sometimes write the perceptron function as

$$o(\vec{x}) = sgn(\vec{w} \cdot \vec{x})$$

where

$$sgn(y) = \begin{cases} 1 \text{ if } y > 0 \\ -1 \text{ otherwise} \end{cases}$$

Learning a perceptron involves choosing values for the weights $w_0, \ldots, w_n$. Therefore, the space H of candidate hypotheses considered in perceptron learning is the set of all possible real-valued weight vectors.

$$H = \{\vec{w} \mid \vec{w} \in \Re^{(n+1)}\}$$

4.4.1 Representational Power of Perceptrons

We can view the perceptron as representing a hyperplane decision surface in the n-dimensional space of instances (i.e., points). The perceptron outputs a 1 for instances lying on one side of the hyperplane and outputs a -1 for instances lying on the other side, as illustrated in Figure 4.3. The equation for this decision hyperplane is $\vec{w} \cdot \vec{x} = 0$. Of course, some sets of positive and negative examples cannot be separated by any hyperplane. Those that can be separated are called *linearly separable* sets of examples.

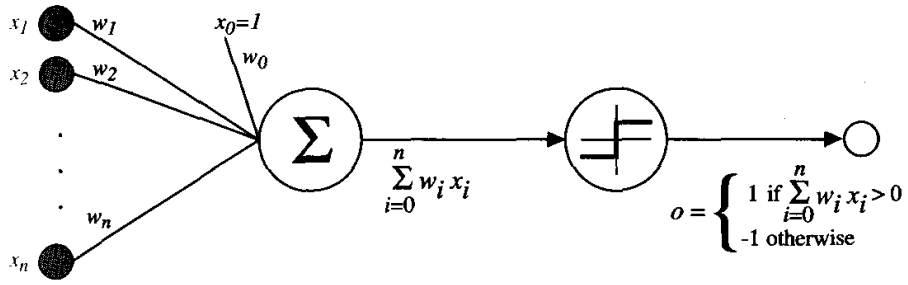


FIGURE 4.2
A perceptron.

A single perceptron can be used to represent many boolean functions. For example, if we assume boolean values of 1 (true) and -1 (false), then one way to use a two-input perceptron to implement the AND function is to set the weights $w_0 = -.8$, and $w_1 = w_2 = .5$. This perceptron can be made to represent the OR function instead by altering the threshold to $w_0 = -.3$. In fact, AND and OR can be viewed as special cases of m-of-n functions: that is, functions where at least m of the n inputs to the perceptron must be true. The OR function corresponds to $m = 1$ and the AND function to $m = n$. Any m-of-n function is easily represented using a perceptron by setting all input weights to the same value (e.g., 0.5) and then setting the threshold w_0 accordingly.

Perceptrons can represent all of the primitive boolean functions AND, OR, NAND ($\neg$ AND), and NOR ($\neg$ OR). Unfortunately, however, some boolean functions cannot be represented by a single perceptron, such as the XOR function whose value is 1 if and only if $x_1 \neq x_2$. Note the set of linearly nonseparable training examples shown in Figure 4.3(b) corresponds to this XOR function.

The ability of perceptrons to represent AND, OR, NAND, and NOR is important because *every* boolean function can be represented by some network of interconnected units based on these primitives. In fact, every boolean function can be represented by some network of perceptrons only two levels deep, in which

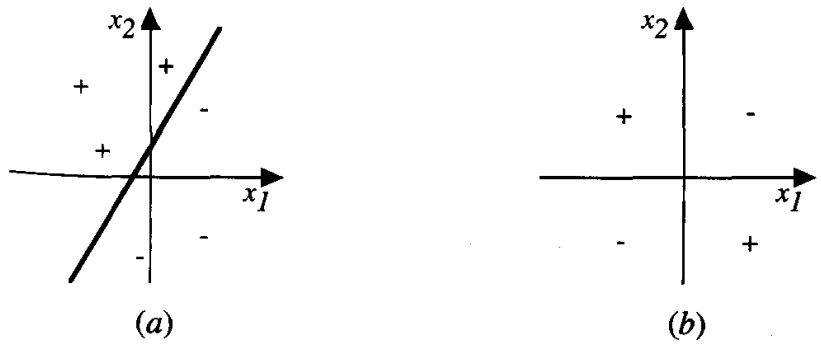


FIGURE 4.3
The decision surface represented by a two-input perceptron. (a) A set of training examples and the decision surface of a perceptron that classifies them correctly. (b) A set of training examples that is not linearly separable (i.e., that cannot be correctly classified by any straight line). x_1 and x_2 are the perceptron inputs. Positive examples are indicated by "+", negative by "−".

the inputs are fed to multiple units, and the outputs of these units are then input to a second, final stage. One way is to represent the boolean function in disjunctive normal form (i.e., as the disjunction (OR) of a set of conjunctions (ANDs) of the inputs and their negations). Note that the input to an AND perceptron can be negated simply by changing the sign of the corresponding input weight.

Because networks of threshold units can represent a rich variety of functions and because single units alone cannot, we will generally be interested in learning multilayer networks of threshold units.

4.4.2 The Perceptron Training Rule

Although we are interested in learning networks of many interconnected units, let us begin by understanding how to learn the weights for a single perceptron. Here the precise learning problem is to determine a weight vector that causes the perceptron to produce the correct ± 1 output for each of the given training examples.

Several algorithms are known to solve this learning problem. Here we consider two: the perceptron rule and the delta rule (a variant of the LMS rule used in Chapter 1 for learning evaluation functions). These two algorithms are guaranteed to converge to somewhat different acceptable hypotheses, under somewhat different conditions. They are important to ANNs because they provide the basis for learning networks of many units.

One way to learn an acceptable weight vector is to begin with random weights, then iteratively apply the perceptron to each training example, modifying the perceptron weights whenever it misclassifies an example. This process is repeated, iterating through the training examples as many times as needed until the perceptron classifies all training examples correctly. Weights are modified at each step according to the *perceptron training rule*, which revises the weight w_i associated with input x_i according to the rule

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = \eta(t - o)x_i$$

Here t is the target output for the current training example, o is the output generated by the perceptron, and η is a positive constant called the *learning rate*. The role of the learning rate is to moderate the degree to which weights are changed at each step. It is usually set to some small value (e.g., 0.1) and is sometimes made to decay as the number of weight-tuning iterations increases.

Why should this update rule converge toward successful weight values? To get an intuitive feel, consider some specific cases. Suppose the training example is correctly classified already by the perceptron. In this case, $(t - o)$ is zero, making Δw_i zero, so that no weights are updated. Suppose the perceptron outputs a -1, when the target output is $+1$. To make the perceptron output a $+1$ instead of -1 in this case, the weights must be altered to increase the value of $\vec{w} \cdot \vec{x}$. For example, if $x_i > 0$, then increasing w_i will bring the perceptron closer to correctly classifying

this example. Notice the training rule will increase w_i in this case, because $(t-o)$, η, and x_i are all positive. For example, if $x_i = .8$, $\eta = 0.1$, $t = 1$, and $o = -1$, then the weight update will be $\Delta w_i = \eta(t-o)x_i = 0.1(1-(-1))0.8 = 0.16$. On the other hand, if $t = -1$ and $o = 1$, then weights associated with positive x_i will be decreased rather than increased.

In fact, the above learning procedure can be proven to converge within a finite number of applications of the perceptron training rule to a weight vector that correctly classifies all training examples, *provided the training examples are linearly separable* and provided a sufficiently small η is used (see Minsky and Papert 1969). If the data are not linearly separable, convergence is not assured.

4.4.3 Gradient Descent and the Delta Rule

Although the perceptron rule finds a successful weight vector when the training examples are linearly separable, it can fail to converge if the examples are not linearly separable. A second training rule, called the *delta rule*, is designed to overcome this difficulty. If the training examples are not linearly separable, the delta rule converges toward a best-fit approximation to the target concept.

The key idea behind the delta rule is to use *gradient descent* to search the hypothesis space of possible weight vectors to find the weights that best fit the training examples. This rule is important because gradient descent provides the basis for the BACKPROPAGATION algorithm, which can learn networks with many interconnected units. It is also important because gradient descent can serve as the basis for learning algorithms that must search through hypothesis spaces containing many different types of continuously parameterized hypotheses.

The delta training rule is best understood by considering the task of training an *unthresholded* perceptron; that is, a *linear unit* for which the output o is given by

$$o(\vec{x}) = \vec{w} \cdot \vec{x} \tag{4.1}$$

Thus, a linear unit corresponds to the first stage of a perceptron, without the threshold.

In order to derive a weight learning rule for linear units, let us begin by specifying a measure for the *training error* of a hypothesis (weight vector), relative to the training examples. Although there are many ways to define this error, one common measure that will turn out to be especially convenient is

$$E(\vec{w}) \equiv \frac{1}{2}\sum_{d \in D}(t_d - o_d)^2 \tag{4.2}$$

where D is the set of training examples, t_d is the target output for training example d, and o_d is the output of the linear unit for training example d. By this definition, $E(\vec{w})$ is simply half the squared difference between the target output t_d and the linear unit output o_d, summed over all training examples. Here we characterize E as a function of $\vec{w}$ because the linear unit output o depends on this weight vector. Of course E also depends on the particular set of training examples, but

we assume these are fixed during training, so we do not bother to write E as an explicit function of these. Chapter 6 provides a Bayesian justification for choosing this particular definition of E. In particular, there we show that under certain conditions the hypothesis that minimizes E is also the most probable hypothesis in H given the training data.

4.4.3.1 VISUALIZING THE HYPOTHESIS SPACE

To understand the gradient descent algorithm, it is helpful to visualize the entire hypothesis space of possible weight vectors and their associated E values, as illustrated in Figure 4.4. Here the axes w_0 and w_1 represent possible values for the two weights of a simple linear unit. The w_0, w_1 plane therefore represents the entire hypothesis space. The vertical axis indicates the error E relative to some fixed set of training examples. The error surface shown in the figure thus summarizes the desirability of every weight vector in the hypothesis space (we desire a hypothesis with minimum error). Given the way in which we chose to define E, for linear units this error surface must always be parabolic with a single global minimum. The specific parabola will depend, of course, on the particular set of training examples.

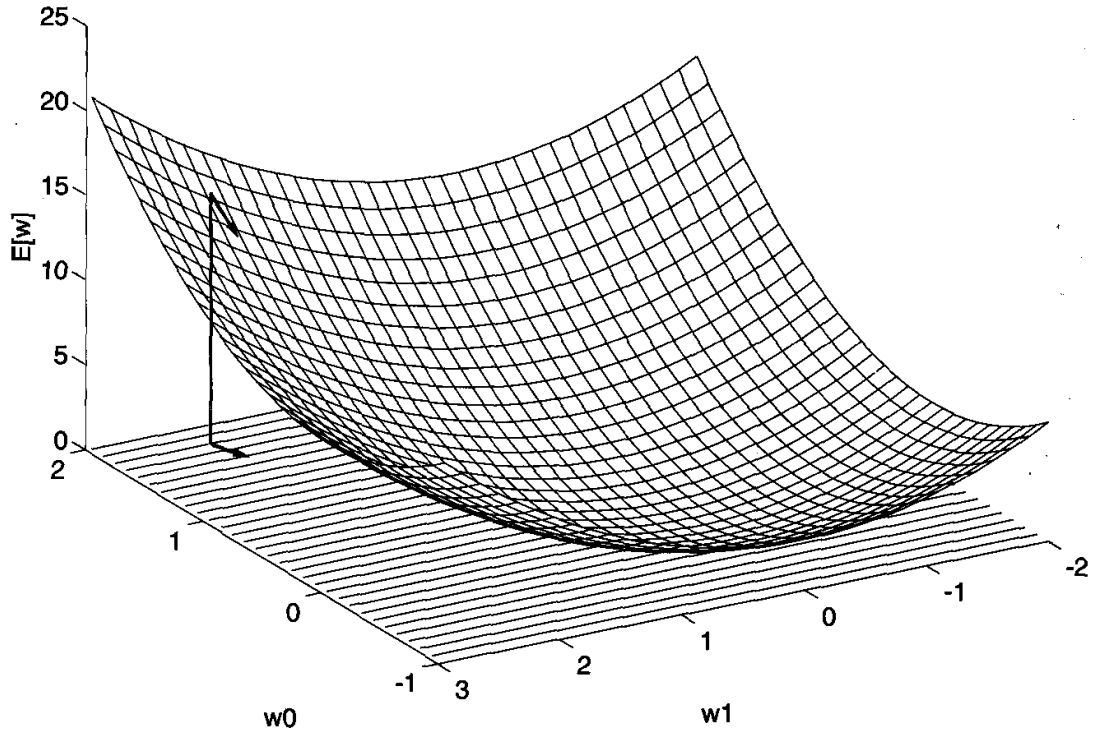


FIGURE 4.4
Error of different hypotheses. For a linear unit with two weights, the hypothesis space H is the w_0, w_1 plane. The vertical axis indicates the error of the corresponding weight vector hypothesis, relative to a fixed set of training examples. The arrow shows the negated gradient at one particular point, indicating the direction in the w_0, w_1 plane producing steepest descent along the error surface.

Gradient descent search determines a weight vector that minimizes E by starting with an arbitrary initial weight vector, then repeatedly modifying it in small steps. At each step, the weight vector is altered in the direction that produces the steepest descent along the error surface depicted in Figure 4.4. This process continues until the global minimum error is reached.

4.4.3.2 DERIVATION OF THE GRADIENT DESCENT RULE

How can we calculate the direction of steepest descent along the error surface? This direction can be found by computing the derivative of E with respect to each component of the vector $\vec{w}$. This vector derivative is called the *gradient* of E with respect to $\vec{w}$, written $\nabla E(\vec{w})$.

$$\nabla E(\vec{w}) \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n}\right] \tag{4.3}$$

Notice $\nabla E(\vec{w})$ is itself a vector, whose components are the partial derivatives of E with respect to each of the w_i. *When interpreted as a vector in weight space, the gradient specifies the direction that produces the steepest increase in* E. The negative of this vector therefore gives the direction of steepest decrease. For example, the arrow in Figure 4.4 shows the negated gradient $-\nabla E(\vec{w})$ for a particular point in the w_0, w_1 plane.

Since the gradient specifies the direction of steepest increase of E, the training rule for gradient descent is

$$\vec{w} \leftarrow \vec{w} + \Delta\vec{w}$$

where

$$\Delta\vec{w} = -\eta\nabla E(\vec{w}) \tag{4.4}$$

Here η is a positive constant called the learning rate, which determines the step size in the gradient descent search. The negative sign is present because we want to move the weight vector in the direction that *decreases* E. This training rule can also be written in its component form

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = -\eta\frac{\partial E}{\partial w_i} \tag{4.5}$$

which makes it clear that steepest descent is achieved by altering each component w_i of $\vec{w}$ in proportion to $\frac{\partial E}{\partial w_i}$.

To construct a practical algorithm for iteratively updating weights according to Equation (4.5), we need an efficient way of calculating the gradient at each step. Fortunately, this is not difficult. The vector of $\frac{\partial E}{\partial w_i}$ derivatives that form the

gradient can be obtained by differentiating E from Equation (4.2), as

$$\begin{aligned}\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2 \\ &= \frac{1}{2}\sum_{d\in D}\frac{\partial}{\partial w_i}(t_d - o_d)^2 \\ &= \frac{1}{2}\sum_{d\in D}2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\ &= \sum_{d\in D}(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x}_d) \\ \frac{\partial E}{\partial w_i} &= \sum_{d\in D}(t_d - o_d)(-x_{id}) \end{aligned} \tag{4.6}$$

where x_{id} denotes the single input component x_i for training example d. We now have an equation that gives $\frac{\partial E}{\partial w_i}$ in terms of the linear unit inputs x_{id}, outputs O_d, and target values t_d associated with the training examples. Substituting Equation (4.6) into Equation (4.5) yields the weight update rule for gradient descent

$$\Delta w_i = \eta \sum_{d\in D}(t_d - o_d)\, x_{id} \tag{4.7}$$

To summarize, the gradient descent algorithm for training linear units is as follows: Pick an initial random weight vector. Apply the linear unit to all training examples, then compute Δw_i for each weight according to Equation (4.7). Update each weight w_i by adding Δw_i, then repeat this process. This algorithm is given in Table 4.1. Because the error surface contains only a single global minimum, this algorithm will converge to a weight vector with minimum error, regardless of whether the training examples are linearly separable, given a sufficiently small learning rate η is used. If η is too large, the gradient descent search runs the risk of overstepping the minimum in the error surface rather than settling into it. For this reason, one common modification to the algorithm is to gradually reduce the value of η as the number of gradient descent steps grows.

4.4.3.3 STOCHASTIC APPROXIMATION TO GRADIENT DESCENT

Gradient descent is an important general paradigm for learning. It is a strategy for searching through a large or infinite hypothesis space that can be applied whenever (1) the hypothesis space contains continuously parameterized hypotheses (e.g., the weights in a linear unit), and (2) the error can be differentiated with respect to these hypothesis parameters. The key practical difficulties in applying gradient descent are (1) converging to a local minimum can sometimes be quite slow (i.e., it can require many thousands of gradient descent steps), and (2) if there are multiple local minima in the error surface, then there is no guarantee that the procedure will find the global minimum.

GRADIENT-DESCENT(*training_examples*, η)

Each training example is a pair of the form $\langle \vec{x}, t \rangle$, *where* $\vec{x}$ *is the vector of input values, and* t *is the target output value.* η *is the learning rate (e.g., .05).*

- Initialize each w_i to some small random value
- Until the termination condition is met, Do
  - Initialize each Δw_i to zero.
  - For each $\langle \vec{x}, t \rangle$ in *training_examples*, Do
    - Input the instance $\vec{x}$ to the unit and compute the output o
    - For each linear unit weight w_i, Do

$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i \tag{T4.1}$$

  - For each linear unit weight w_i, Do

$$w_i \leftarrow w_i + \Delta w_i \tag{T4.2}$$

TABLE 4.1
GRADIENT DESCENT algorithm for training a linear unit. To implement the stochastic approximation to gradient descent, Equation (T4.2) is deleted, and Equation (T4.1) replaced by $w_i \leftarrow w_i + \eta(t - o)x_i$.

One common variation on gradient descent intended to alleviate these difficulties is called *incremental gradient descent*, or alternatively *stochastic gradient descent*. Whereas the gradient descent training rule presented in Equation (4.7) computes weight updates after summing over *all* the training examples in D, the idea behind stochastic gradient descent is to approximate this gradient descent search by updating weights incrementally, following the calculation of the error for *each* individual example. The modified training rule is like the training rule given by Equation (4.7) except that as we iterate through each training example we update the weight according to

$$\Delta w_i = \eta(t - o)\, x_i \tag{4.10}$$

where t, o, and x_i are the target value, unit output, and ith input for the training example in question. To modify the gradient descent algorithm of Table 4.1 to implement this stochastic approximation, Equation (T4.2) is simply deleted and Equation (T4.1) replaced by $w_i \leftarrow w_i + \eta(t - o)\, x_i$. One way to view this stochastic gradient descent is to consider a distinct error function $E_d(\vec{w})$ defined for each individual training example d as follows

$$E_d(\vec{w}) = \frac{1}{2}(t_d - o_d)^2 \tag{4.11}$$

where t_d and o_d are the target value and the unit output value for training example d. Stochastic gradient descent iterates over the training examples d in D, at each iteration altering the weights according to the gradient with respect to $E_d(\vec{w})$. The sequence of these weight updates, when iterated over all training examples, provides a reasonable approximation to descending the gradient with respect to our original error function $E(\vec{w})$. By making the value of η (the gradient

descent step size) sufficiently small, stochastic gradient descent can be made to approximate true gradient descent arbitrarily closely. The key differences between standard gradient descent and stochastic gradient descent are:

- In standard gradient descent, the error is summed over all examples before updating weights, whereas in stochastic gradient descent weights are updated upon examining each training example.
- Summing over multiple examples in standard gradient descent requires more computation per weight update step. On the other hand, because it uses the true gradient, standard gradient descent is often used with a larger step size per weight update than stochastic gradient descent.
- In cases where there are multiple local minima with respect to $E(\vec{w})$, stochastic gradient descent can sometimes avoid falling into these local minima because it uses the various $\nabla E_d(\vec{w})$ rather than $\nabla E(\vec{w})$ to guide its search.

Both stochastic and standard gradient descent methods are commonly used in practice.

The training rule in Equation (4.10) is known as the *delta rule*, or sometimes the LMS (least-mean-square) rule, Adaline rule, or Widrow-Hoff rule (after its inventors). In Chapter 1 we referred to it as the LMS weight-update rule when describing its use for learning an evaluation function for game playing. Notice the delta rule in Equation (4.10) is similar to the perceptron training rule in Equation (4.4.2). In fact, the two expressions appear to be identical. However, the rules are different because in the delta rule o refers to the linear unit output $o(\vec{x}) = \vec{w} \cdot \vec{x}$, whereas for the perceptron rule o refers to the thresholded output $o(\vec{x}) = sgn(\vec{w} \cdot \vec{x})$.

Although we have presented the delta rule as a method for learning weights for unthresholded linear units, it can easily be used to train thresholded perceptron units, as well. Suppose that $o = \vec{w} \cdot \vec{x}$ is the unthresholded linear unit output as above, and $o' = sgn(\vec{w} \cdot \vec{x})$ is the result of thresholding o as in the perceptron. Now if we wish to train a perceptron to fit training examples with target values of ± 1 for o', we can use these same target values and examples to train o instead, using the delta rule. Clearly, if the unthresholded output o can be trained to fit these values perfectly, then the threshold output o' will fit them as well (because $sgn(1) = 1$, and $sgn(-1) = -1$). Even when the target values cannot be fit perfectly, the thresholded o' value will correctly fit the ± 1 target value whenever the linear unit output o has the correct sign. Notice, however, that while this procedure will learn weights that minimize the error in the linear unit output o, these weights will not necessarily minimize the number of training examples misclassified by the thresholded output o'.

4.4.4 Remarks

We have considered two similar algorithms for iteratively learning perceptron weights. The key difference between these algorithms is that the perceptron train-

ing rule updates weights based on the error in the *thresholded* perceptron output, whereas the delta rule updates weights based on the error in the *unthresholded* linear combination of inputs.

The difference between these two training rules is reflected in different convergence properties. The perceptron training rule converges after a finite number of iterations to a hypothesis that perfectly classifies the training data, *provided the training examples are linearly separable.* The delta rule converges only asymptotically toward the minimum error hypothesis, possibly requiring unbounded time, but converges *regardless of whether the training data are linearly separable.* A detailed presentation of the convergence proofs can be found in Hertz et al. (1991).

A third possible algorithm for learning the weight vector is linear programming. Linear programming is a general, efficient method for solving sets of linear inequalities. Notice each training example corresponds to an inequality of the form $\vec{w} \cdot \vec{x} > 0$ or $\vec{w} \cdot \vec{x} \leq 0$, and their solution is the desired weight vector. Unfortunately, this approach yields a solution only when the training examples are linearly separable; however, Duda and Hart (1973, p. 168) suggest a more subtle formulation that accommodates the nonseparable case. In any case, the approach of linear programming does not scale to training multilayer networks, which is our primary concern. In contrast, the gradient descent approach, on which the delta rule is based, can be easily extended to multilayer networks, as shown in the following section.

4.5 MULTILAYER NETWORKS AND THE BACKPROPAGATION ALGORITHM

As noted in Section 4.4.1, single perceptrons can only express linear decision surfaces. In contrast, the kind of multilayer networks learned by the BACKPROPAGATION algorithm are capable of expressing a rich variety of nonlinear decision surfaces. For example, a typical multilayer network and decision surface is depicted in Figure 4.5. Here the speech recognition task involves distinguishing among 10 possible vowels, all spoken in the context of "h_d" (i.e., "hid," "had," "head," "hood," etc.). The input speech signal is represented by two numerical parameters obtained from a spectral analysis of the sound, allowing us to easily visualize the decision surface over the two-dimensional instance space. As shown in the figure, it is possible for the multilayer network to represent highly nonlinear decision surfaces that are much more expressive than the linear decision surfaces of single units shown earlier in Figure 4.3.

This section discusses how to learn such multilayer networks using a gradient descent algorithm similar to that discussed in the previous section.

4.5.1 A Differentiable Threshold Unit

What type of unit shall we use as the basis for constructing multilayer networks? At first we might be tempted to choose the linear units discussed in the previous

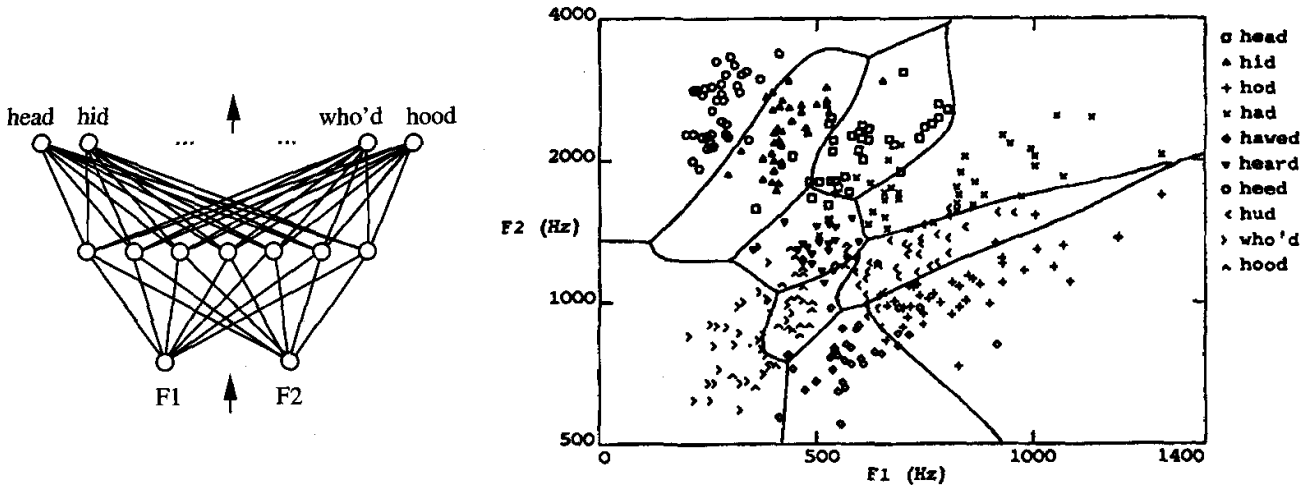


FIGURE 4.5
Decision regions of a multilayer feedforward network. The network shown here was trained to recognize 1 of 10 vowel sounds occurring in the context "h_d" (e.g., "had," "hid"). The network input consists of two parameters, F1 and F2, obtained from a spectral analysis of the sound. The 10 network outputs correspond to the 10 possible vowel sounds. The network prediction is the output whose value is highest. The plot on the right illustrates the highly nonlinear decision surface represented by the learned network. Points shown on the plot are test examples distinct from the examples used to train the network. (Reprinted by permission from Haung and Lippmann (1988).)

section, for which we have already derived a gradient descent learning rule. However, multiple layers of cascaded linear units still produce only linear functions, and we prefer networks capable of representing highly nonlinear functions. The perceptron unit is another possible choice, but its discontinuous threshold makes it undifferentiable and hence unsuitable for gradient descent. What we need is a unit whose output is a nonlinear function of its inputs, but whose output is also a differentiable function of its inputs. One solution is the *sigmoid unit*—a unit very much like a perceptron, but based on a smoothed, differentiable threshold function.

The sigmoid unit is illustrated in Figure 4.6. Like the perceptron, the sigmoid unit first computes a linear combination of its inputs, then applies a threshold to the result. In the case of the sigmoid unit, however, the threshold output is a

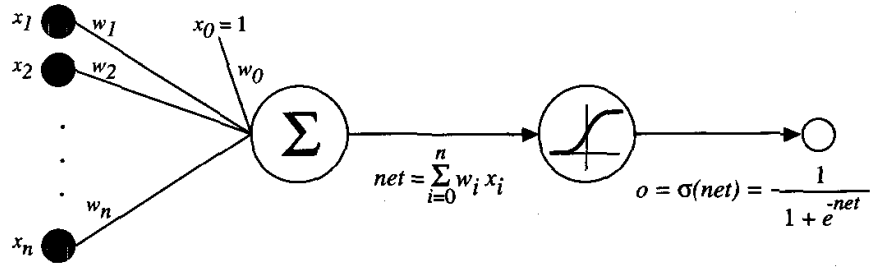


FIGURE 4.6
The sigmoid threshold unit.

continuous function of its input. More precisely, the sigmoid unit computes its output o as

$$o = \sigma(\vec{w} \cdot \vec{x})$$

where

$$\sigma(y) = \frac{1}{1 + e^{-y}} \tag{4.12}$$

σ is often called the sigmoid function or, alternatively, the logistic function. Note its output ranges between 0 and 1, increasing monotonically with its input (see the threshold function plot in Figure 4.6.). Because it maps a very large input domain to a small range of outputs, it is often referred to as the *squashing function* of the unit. The sigmoid function has the useful property that its derivative is easily expressed in terms of its output [in particular, $\frac{d\sigma(y)}{dy} = \sigma(y) \cdot (1 - \sigma(y))$]. As we shall see, the gradient descent learning rule makes use of this derivative. Other differentiable functions with easily calculated derivatives are sometimes used in place of σ. For example, the term e^{-y} in the sigmoid function definition is sometimes replaced by $e^{-k \cdot y}$ where k is some positive constant that determines the steepness of the threshold. The function *tanh* is also sometimes used in place of the sigmoid function (see Exercise 4.8).

4.5.2 The BACKPROPAGATION Algorithm

The BACKPROPAGATION algorithm learns the weights for a multilayer network, given a network with a fixed set of units and interconnections. It employs gradient descent to attempt to minimize the squared error between the network output values and the target values for these outputs. This section presents the BACKPROPAGATION algorithm, and the following section gives the derivation for the gradient descent weight update rule used by BACKPROPAGATION.

Because we are considering networks with multiple output units rather than single units as before, we begin by redefining E to sum the errors over all of the network output units

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 \tag{4.13}$$

where *outputs* is the set of output units in the network, and t_{kd} and o_{kd} are the target and output values associated with the kth output unit and training example d.

The learning problem faced by BACKPROPAGATION is to search a large hypothesis space defined by all possible weight values for all the units in the network. The situation can be visualized in terms of an error surface similar to that shown for linear units in Figure 4.4. The error in that diagram is replaced by our new definition of E, and the other dimensions of the space correspond now to all of the weights associated with all of the units in the network. As in the case of training a single unit, gradient descent can be used to attempt to find a hypothesis to minimize E.

BACKPROPAGATION($training\_examples, \eta, n_{in}, n_{out}, n_{hidden}$)

Each training example is a pair of the form $\langle \vec{x}, \vec{t} \rangle$, *where* $\vec{x}$ *is the vector of network input values, and* $\vec{t}$ *is the vector of target network output values.*

η *is the learning rate (e.g., .05).* n_{in} *is the number of network inputs,* n_{hidden} *the number of units in the hidden layer, and* n_{out} *the number of output units.*

The input from unit i *into unit* j *is denoted* x_{ji}, *and the weight from unit* i *to unit* j *is denoted* w_{ji}.

- Create a feed-forward network with n_{in} inputs, n_{hidden} hidden units, and n_{out} output units.
- Initialize all network weights to small random numbers (e.g., between $-.05$ and .05).
- Until the termination condition is met, Do
  - For each $\langle \vec{x}, \vec{t} \rangle$ in $training\_examples$, Do

Propagate the input forward through the network:

1. Input the instance $\vec{x}$ to the network and compute the output o_u of every unit u in the network.

Propagate the errors backward through the network:

2. For each network output unit k, calculate its error term δ_k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k) \tag{T4.3}$$

3. For each hidden unit h, calculate its error term δ_h

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k \tag{T4.4}$$

4. Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

where

$$\Delta w_{ji} = \eta \delta_j x_{ji} \tag{T4.5}$$

TABLE 4.2
The stochastic gradient descent version of the BACKPROPAGATION algorithm for feedforward networks containing two layers of sigmoid units.

One major difference in the case of multilayer networks is that the error surface can have multiple local minima, in contrast to the single-minimum parabolic error surface shown in Figure 4.4. Unfortunately, this means that gradient descent is guaranteed only to converge toward some local minimum, and not necessarily the global minimum error. Despite this obstacle, in practice BACKPROPAGATION has been found to produce excellent results in many real-world applications.

The BACKPROPAGATION algorithm is presented in Table 4.2. The algorithm as described here applies to layered feedforward networks containing two layers of sigmoid units, with units at each layer connected to all units from the preceding layer. This is the incremental, or stochastic, gradient descent version of BACKPROPAGATION. The notation used here is the same as that used in earlier sections, with the following extensions:

- An index (e.g., an integer) is assigned to each node in the network, where a "node" is either an input to the network or the output of some unit in the network.
- x_{ji} denotes the input from node i to unit j, and w_{ji} denotes the corresponding weight.
- δ_n denotes the error term associated with unit n. It plays a role analogous to the quantity $(t - o)$ in our earlier discussion of the delta training rule. As we shall see later, $\delta_n = -\frac{\partial E}{\partial net_n}$.

Notice the algorithm in Table 4.2 begins by constructing a network with the desired number of hidden and output units and initializing all network weights to small random values. Given this fixed network structure, the main loop of the algorithm then repeatedly iterates over the training examples. For each training example, it applies the network to the example, calculates the error of the network output for this example, computes the gradient with respect to the error on this example, then updates all weights in the network. This gradient descent step is iterated (often thousands of times, using the same training examples multiple times) until the network performs acceptably well.

The gradient descent weight-update rule (Equation [T4.5] in Table 4.2) is similar to the delta training rule (Equation [4.10]). Like the delta rule, it updates each weight in proportion to the learning rate η, the input value x_{ji} to which the weight is applied, and the error in the output of the unit. The only difference is that the error $(t - o)$ in the delta rule is replaced by a more complex error term, δ_j. The exact form of δ_j follows from the derivation of the weight-tuning rule given in Section 4.5.3. To understand it intuitively, first consider how δ_k is computed for each network *output* unit k (Equation [T4.3] in the algorithm). δ_k is simply the familiar $(t_k - o_k)$ from the delta rule, multiplied by the factor $o_k(1 - o_k)$, which is the derivative of the sigmoid squashing function. The δ_h value for each *hidden* unit h has a similar form (Equation [T4.4] in the algorithm). However, since training examples provide target values t_k *only* for network outputs, no target values are directly available to indicate the error of hidden units' values. Instead, the error term for hidden unit h is calculated by summing the error terms δ_k for each output unit influenced by h, weighting each of the δ_k's by w_{kh}, the weight from hidden unit h to output unit k. This weight characterizes the degree to which hidden unit h is "responsible for" the error in output unit k.

The algorithm in Table 4.2 updates weights incrementally, following the presentation of each training example. This corresponds to a stochastic approximation to gradient descent. To obtain the true gradient of E one would sum the $\delta_j x_{ji}$ values over all training examples before altering weight values.

The weight-update loop in BACKPROPAGATION may be iterated thousands of times in a typical application. A variety of termination conditions can be used to halt the procedure. One may choose to halt after a fixed number of iterations through the loop, or once the error on the training examples falls below some threshold, or once the error on a separate validation set of examples meets some

criterion. The choice of termination criterion is an important one, because too few iterations can fail to reduce error sufficiently, and too many can lead to overfitting the training data. This issue is discussed in greater detail in Section 4.6.5.

4.5.2.1 ADDING MOMENTUM

Because BACKPROPAGATION is such a widely used algorithm, many variations have been developed. Perhaps the most common is to alter the weight-update rule in Equation (T4.5) in the algorithm by making the weight update on the nth iteration depend partially on the update that occurred during the $(n-1)$th iteration, as follows:

$$\Delta w_{ji}(n) = \eta \, \delta_j \, x_{ji} + \alpha \Delta w_{ji}(n-1) \tag{4.18}$$

Here $\Delta w_{ji}(n)$ is the weight update performed during the nth iteration through the main loop of the algorithm, and $0 \leq \alpha < 1$ is a constant called the *momentum*. Notice the first term on the right of this equation is just the weight-update rule of Equation (T4.5) in the BACKPROPAGATION algorithm. The second term on the right is new and is called the momentum term. To see the effect of this momentum term, consider that the gradient descent search trajectory is analogous to that of a (momentumless) ball rolling down the error surface. The effect of α is to add momentum that tends to keep the ball rolling in the same direction from one iteration to the next. This can sometimes have the effect of keeping the ball rolling through small local minima in the error surface, or along flat regions in the surface where the ball would stop if there were no momentum. It also has the effect of gradually increasing the step size of the search in regions where the gradient is unchanging, thereby speeding convergence.

4.5.2.2 LEARNING IN ARBITRARY ACYCLIC NETWORKS

The definition of BACKPROPAGATION presented in Table 4.2 applies only to two-layer networks. However, the algorithm given there easily generalizes to feedforward networks of arbitrary depth. The weight update rule seen in Equation (T4.5) is retained, and the only change is to the procedure for computing δ values. In general, the δ_r value for a unit r in layer m is computed from the δ values at the next deeper layer $m+1$ according to

$$\delta_r = o_r (1 - o_r) \sum_{s \in layer\, m+1} w_{sr} \, \delta_s \tag{4.19}$$

Notice this is identical to Step 3 in the algorithm of Table 4.2, so all we are really saying here is that this step may be repeated for any number of hidden layers in the network.

It is equally straightforward to generalize the algorithm to any directed acyclic graph, regardless of whether the network units are arranged in uniform layers as we have assumed up to now. In the case that they are not, the rule for calculating δ for any internal unit (i.e., any unit that is not an output) is

$$\delta_r = o_r (1 - o_r) \sum_{s \in Downstream(r)} w_{sr} \, \delta_s \tag{4.20}$$

where $Downstream(r)$ is the set of units immediately downstream from unit r in the network: that is, all units whose inputs include the output of unit r. It is this general form of the weight-update rule that we derive in Section 4.5.3.

4.5.3 Derivation of the BACKPROPAGATION Rule

This section presents the derivation of the BACKPROPAGATION weight-tuning rule. It may be skipped on a first reading, without loss of continuity.

The specific problem we address here is deriving the stochastic gradient descent rule implemented by the algorithm in Table 4.2. Recall from Equation (4.11) that stochastic gradient descent involves iterating through the training examples one at a time, for each training example d descending the gradient of the error E_d with respect to this single example. In other words, for each training example d every weight w_{ji} is updated by adding to it Δw_{ji}

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} \tag{4.21}$$

where E_d is the error on training example d, summed over all output units in the network

$$E_d(\vec{w}) \equiv \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2$$

Here *outputs* is the set of output units in the network, t_k is the target value of unit k for training example d, and o_k is the output of unit k given training example d.

The derivation of the stochastic gradient descent rule is conceptually straightforward, but requires keeping track of a number of subscripts and variables. We will follow the notation shown in Figure 4.6, adding a subscript j to denote to the jth unit of the network as follows:

- x_{ji} = the ith input to unit j
- w_{ji} = the weight associated with the ith input to unit j
- $net_j = \sum_i w_{ji} x_{ji}$ (the weighted sum of inputs for unit j)
- o_j = the output computed by unit j
- t_j = the target output for unit j
- σ = the sigmoid function
- $outputs$ = the set of units in the final layer of the network
- $Downstream(j)$ = the set of units whose immediate inputs include the output of unit j

We now derive an expression for $\frac{\partial E_d}{\partial w_{ji}}$ in order to implement the stochastic gradient descent rule seen in Equation (4.21). To begin, notice that weight w_{ji} can influence the rest of the network only through net_j. Therefore, we can use the

chain rule to write

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j}\frac{\partial net_j}{\partial w_{ji}}$$
$$= \frac{\partial E_d}{\partial net_j} x_{ji} \tag{4.22}$$

Given Equation (4.22), our remaining task is to derive a convenient expression for $\frac{\partial E_d}{\partial net_j}$. We consider two cases in turn: the case where unit j is an output unit for the network, and the case where j is an internal unit.

Case 1: Training Rule for Output Unit Weights. Just as w_{ji} can influence the rest of the network only through net_j, net_j can influence the network only through o_j. Therefore, we can invoke the chain rule again to write

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial net_j} \tag{4.23}$$

To begin, consider just the first term in Equation (4.23)

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j}\frac{1}{2}\sum_{k \in outputs}(t_k - o_k)^2$$

The derivatives $\frac{\partial}{\partial o_j}(t_k - o_k)^2$ will be zero for all output units k except when $k = j$. We therefore drop the summation over output units and simply set $k = j$.

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j}\frac{1}{2}(t_j - o_j)^2$$
$$= \frac{1}{2}2(t_j - o_j)\frac{\partial(t_j - o_j)}{\partial o_j}$$
$$= -(t_j - o_j) \tag{4.24}$$

Next consider the second term in Equation (4.23). Since $o_j = \sigma(net_j)$, the derivative $\frac{\partial o_j}{\partial net_j}$ is just the derivative of the sigmoid function, which we have already noted is equal to $\sigma(net_j)(1 - \sigma(net_j))$. Therefore,

$$\frac{\partial o_j}{\partial net_j} = \frac{\partial \sigma(net_j)}{\partial net_j}$$
$$= o_j(1 - o_j) \tag{4.25}$$

Substituting expressions (4.24) and (4.25) into (4.23), we obtain

$$\frac{\partial E_d}{\partial net_j} = -(t_j - o_j)\, o_j(1 - o_j) \tag{4.26}$$

and combining this with Equations (4.21) and (4.22), we have the stochastic gradient descent rule for output units

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} = \eta\ (t_j - o_j)\ o_j(1 - o_j) x_{ji} \tag{4.27}$$

Note this training rule is exactly the weight update rule implemented by Equations (T4.3) and (T4.5) in the algorithm of Table 4.2. Furthermore, we can see now that δ_k in Equation (T4.3) is equal to the quantity $-\frac{\partial E_d}{\partial net_k}$. In the remainder of this section we will use δ_i to denote the quantity $-\frac{\partial E_d}{\partial net_i}$ for an arbitrary unit i.

Case 2: Training Rule for Hidden Unit Weights. In the case where j is an internal, or hidden unit in the network, the derivation of the training rule for w_{ji} must take into account the indirect ways in which w_{ji} can influence the network outputs and hence E_d. For this reason, we will find it useful to refer to the set of all units immediately downstream of unit j in the network (i.e., all units whose direct inputs include the output of unit j). We denote this set of units by $Downstream(j)$. Notice that net_j can influence the network outputs (and therefore E_d) only through the units in $Downstream(j)$. Therefore, we can write

$$\begin{aligned}
\frac{\partial E_d}{\partial net_j} &= \sum_{k \in Downstream(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k\ w_{kj} \frac{\partial o_j}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k\ w_{kj}\ o_j(1 - o_j)
\end{aligned} \tag{4.28}$$

Rearranging terms and using δ_j to denote $-\frac{\partial E_d}{\partial net_j}$, we have

$$\delta_j = o_j(1 - o_j) \sum_{k \in Downstream(j)} \delta_k\ w_{kj}$$

and

$$\Delta w_{ji} = \eta\ \delta_j\ x_{ji}$$

which is precisely the general rule from Equation (4.20) for updating internal unit weights in arbitrary acyclic directed graphs. Notice Equation (T4.4) from Table 4.2 is just a special case of this rule, in which $Downstream(j) = outputs$.

4.6 REMARKS ON THE BACKPROPAGATION ALGORITHM

4.6.1 Convergence and Local Minima

As shown above, the BACKPROPAGATION algorithm implements a gradient descent search through the space of possible network weights, iteratively reducing the error E between the training example target values and the network outputs. Because the error surface for multilayer networks may contain many different local minima, gradient descent can become trapped in any of these. As a result, BACKPROPAGATION over multilayer networks is only guaranteed to converge toward some local minimum in E and not necessarily to the global minimum error.

Despite the lack of assured convergence to the global minimum error, BACKPROPAGATION is a highly effective function approximation method in practice. In many practical applications the problem of local minima has not been found to be as severe as one might fear. To develop some intuition here, consider that networks with large numbers of weights correspond to error surfaces in very high dimensional spaces (one dimension per weight). When gradient descent falls into a local minimum with respect to one of these weights, it will not necessarily be in a local minimum with respect to the other weights. In fact, the more weights in the network, the more dimensions that might provide "escape routes" for gradient descent to fall away from the local minimum with respect to this single weight.

A second perspective on local minima can be gained by considering the manner in which network weights evolve as the number of training iterations increases. Notice that if network weights are initialized to values near zero, then during early gradient descent steps the network will represent a very smooth function that is approximately linear in its inputs. This is because the sigmoid threshold function itself is approximately linear when the weights are close to zero (see the plot of the sigmoid function in Figure 4.6). Only after the weights have had time to grow will they reach a point where they can represent highly nonlinear network functions. One might expect more local minima to exist in the region of the weight space that represents these more complex functions. One hopes that by the time the weights reach this point they have already moved close enough to the global minimum that even local minima in this region are acceptable.

Despite the above comments, gradient descent over the complex error surfaces represented by ANNs is still poorly understood, and no methods are known to predict with certainty when local minima will cause difficulties. Common heuristics to attempt to alleviate the problem of local minima include:

- Add a momentum term to the weight-update rule as described in Equation (4.18). Momentum can sometimes carry the gradient descent procedure through narrow local minima (though in principle it can also carry it through narrow global minima into other local minima!).
- Use stochastic gradient descent rather than true gradient descent. As discussed in Section 4.4.3.3, the stochastic approximation to gradient descent effectively descends a different error surface for each training example, re-

lying on the average of these to approximate the gradient with respect to the full training set. These different error surfaces typically will have different local minima, making it less likely that the process will get stuck in any one of them.

- Train multiple networks using the same data, but initializing each network with different random weights. If the different training efforts lead to different local minima, then the network with the best performance over a separate validation data set can be selected. Alternatively, all networks can be retained and treated as a "committee" of networks whose output is the (possibly weighted) average of the individual network outputs.

4.6.2 Representational Power of Feedforward Networks

What set of functions can be represented by feedforward networks? Of course the answer depends on the width and depth of the networks. Although much is still unknown about which function classes can be described by which types of networks, three quite general results are known:

- *Boolean functions.* Every boolean function can be represented exactly by some network with two layers of units, although the number of hidden units required grows exponentially in the worst case with the number of network inputs. To see how this can be done, consider the following general scheme for representing an arbitrary boolean function: For each possible input vector, create a distinct hidden unit and set its weights so that it activates if and only if this specific vector is input to the network. This produces a hidden layer that will always have exactly one unit active. Now implement the output unit as an OR gate that activates just for the desired input patterns.
- *Continuous functions.* Every bounded continuous function can be approximated with arbitrarily small error (under a finite norm) by a network with two layers of units (Cybenko 1989; Hornik et al. 1989). The theorem in this case applies to networks that use sigmoid units at the hidden layer and (unthresholded) linear units at the output layer. The number of hidden units required depends on the function to be approximated.
- *Arbitrary functions.* Any function can be approximated to arbitrary accuracy by a network with three layers of units (Cybenko 1988). Again, the output layer uses linear units, the two hidden layers use sigmoid units, and the number of units required at each layer is not known in general. The proof of this involves showing that any function can be approximated by a linear combination of many localized functions that have value 0 everywhere except for some small region, and then showing that two layers of sigmoid units are sufficient to produce good local approximations.

These results show that limited depth feedforward networks provide a very expressive hypothesis space for BACKPROPAGATION. However, it is important to

keep in mind that the network weight vectors reachable by gradient descent from the initial weight values may not include all possible weight vectors. Hertz et al. (1991) provide a more detailed discussion of the above results.

4.6.3 Hypothesis Space Search and Inductive Bias

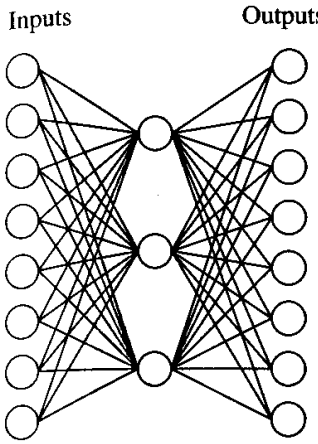
It is interesting to compare the hypothesis space search of BACKPROPAGATION to the search performed by other learning algorithms. For BACKPROPAGATION, every possible assignment of network weights represents a syntactically distinct hypothesis that in principle can be considered by the learner. In other words, the hypothesis space is the n-dimensional Euclidean space of the n network weights. Notice this hypothesis space is *continuous*, in contrast to the hypothesis spaces of decision tree learning and other methods based on discrete representations. The fact that it is continuous, together with the fact that E is differentiable with respect to the continuous parameters of the hypothesis, results in a well-defined error gradient that provides a very useful structure for organizing the search for the best hypothesis. This structure is quite different from the general-to-specific ordering used to organize the search for symbolic concept learning algorithms, or the simple-to-complex ordering over decision trees used by the ID3 and C4.5 algorithms.

What is the inductive bias by which BACKPROPAGATION generalizes beyond the observed data? It is difficult to characterize precisely the inductive bias of BACKPROPAGATION learning, because it depends on the interplay between the gradient descent search and the way in which the weight space spans the space of representable functions. However, one can roughly characterize it as *smooth interpolation between data points*. Given two positive training examples with no negative examples between them, BACKPROPAGATION will tend to label points in between as positive examples as well. This can be seen, for example, in the decision surface illustrated in Figure 4.5, in which the specific sample of training examples gives rise to smoothly varying decision regions.

4.6.4 Hidden Layer Representations

One intriguing property of BACKPROPAGATION is its ability to discover useful intermediate representations at the hidden unit layers inside the network. Because training examples constrain only the network inputs and outputs, the weight-tuning procedure is free to set weights that define whatever hidden unit representation is most effective at minimizing the squared error E. This can lead BACKPROPAGATION to define new hidden layer features that are not explicit in the input representation, but which capture properties of the input instances that are most relevant to learning the target function.

Consider, for example, the network shown in Figure 4.7. Here, the eight network inputs are connected to three hidden units, which are in turn connected to the eight output units. Because of this structure, the three hidden units will be forced to re-represent the eight input values in some way that captures their



Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.15	.99	.99	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.01	.11	.88	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

FIGURE 4.7
Learned Hidden Layer Representation. This $8 \times 3 \times 8$ network was trained to learn the identity function, using the eight training examples shown. After 5000 training epochs, the three hidden unit values encode the eight distinct inputs using the encoding shown on the right. Notice if the encoded values are rounded to zero or one, the result is the standard binary encoding for eight distinct values.

relevant features, so that this hidden layer representation can be used by the output units to compute the correct target values.

Consider training the network shown in Figure 4.7 to learn the simple target function $f(\vec{x}) = \vec{x}$, where $\vec{x}$ is a vector containing seven 0's and a single 1. The network must learn to reproduce the eight inputs at the corresponding eight output units. Although this is a simple function, the network in this case is constrained to use only three hidden units. Therefore, the essential information from all eight input units must be captured by the three learned hidden units.

When BACKPROPAGATION is applied to this task, using each of the eight possible vectors as training examples, it successfully learns the target function. What hidden layer representation is created by the gradient descent BACKPROPAGATION algorithm? By examining the hidden unit values generated by the learned network for each of the eight possible input vectors, it is easy to see that the learned encoding is similar to the familiar standard binary encoding of eight values using three bits (e.g., $000, 001, 010, \ldots, 111$). The exact values of the hidden units for one typical run of BACKPROPAGATION are shown in Figure 4.7.

This ability of multilayer networks to automatically discover useful representations at the hidden layers is a key feature of ANN learning. In contrast to learning methods that are constrained to use only predefined features provided by the human designer, this provides an important degree of flexibility that allows the learner to invent features not explicitly introduced by the human designer. Of course these invented features must still be computable as sigmoid unit functions of the provided network inputs. Note when more layers of units are used in the network, more complex features can be invented. Another example of hidden layer features is provided in the face recognition application discussed in Section 4.7.

In order to develop a better intuition for the operation of BACKPROPAGATION in this example, let us examine the operation of the gradient descent procedure in

greater detail[†]. The network in Figure 4.7 was trained using the algorithm shown in Table 4.2, with initial weights set to random values in the interval $(-0.1, 0.1)$, learning rate $\eta = 0.3$, and no weight momentum (i.e., $\alpha = 0$). Similar results were obtained by using other learning rates and by including nonzero momentum. The hidden unit encoding shown in Figure 4.7 was obtained after 5000 training iterations through the outer loop of the algorithm (i.e., 5000 iterations through each of the eight training examples). Most of the interesting weight changes occurred, however, during the first 2500 iterations.

We can directly observe the effect of BACKPROPAGATION's gradient descent search by plotting the squared output error as a function of the number of gradient descent search steps. This is shown in the top plot of Figure 4.8. Each line in this plot shows the squared output error summed over all training examples, for one of the eight network outputs. The horizontal axis indicates the number of iterations through the outermost loop of the BACKPROPAGATION algorithm. As this plot indicates, the sum of squared errors for each output decreases as the gradient descent procedure proceeds, more quickly for some output units and less quickly for others.

The evolution of the hidden layer representation can be seen in the second plot of Figure 4.8. This plot shows the three hidden unit values computed by the learned network for one of the possible inputs (in particular, 01000000). Again, the horizontal axis indicates the number of training iterations. As this plot indicates, the network passes through a number of different encodings before converging to the final encoding given in Figure 4.7.

Finally, the evolution of individual weights within the network is illustrated in the third plot of Figure 4.8. This plot displays the evolution of weights connecting the eight input units (and the constant 1 bias input) to one of the three hidden units. Notice that significant changes in the weight values for this hidden unit coincide with significant changes in the hidden layer encoding and output squared errors. The weight that converges to a value near zero in this case is the bias weight w_0.

4.6.5 Generalization, Overfitting, and Stopping Criterion

In the description of the BACKPROPAGATION algorithm in Table 4.2, the termination condition for the algorithm has been left unspecified. What is an appropriate condition for terminating the weight update loop? One obvious choice is to continue training until the error E on the training examples falls below some predetermined threshold. In fact, this is a poor strategy because BACKPROPAGATION is susceptible to overfitting the training examples at the cost of decreasing generalization accuracy over other unseen examples.

To see the dangers of minimizing the error over the training data, consider how the error E varies with the number of weight iterations. Figure 4.9 shows

[†]The source code to reproduce this example is available at http://www.cs.cmu.edu/~tom/mlbook.html.

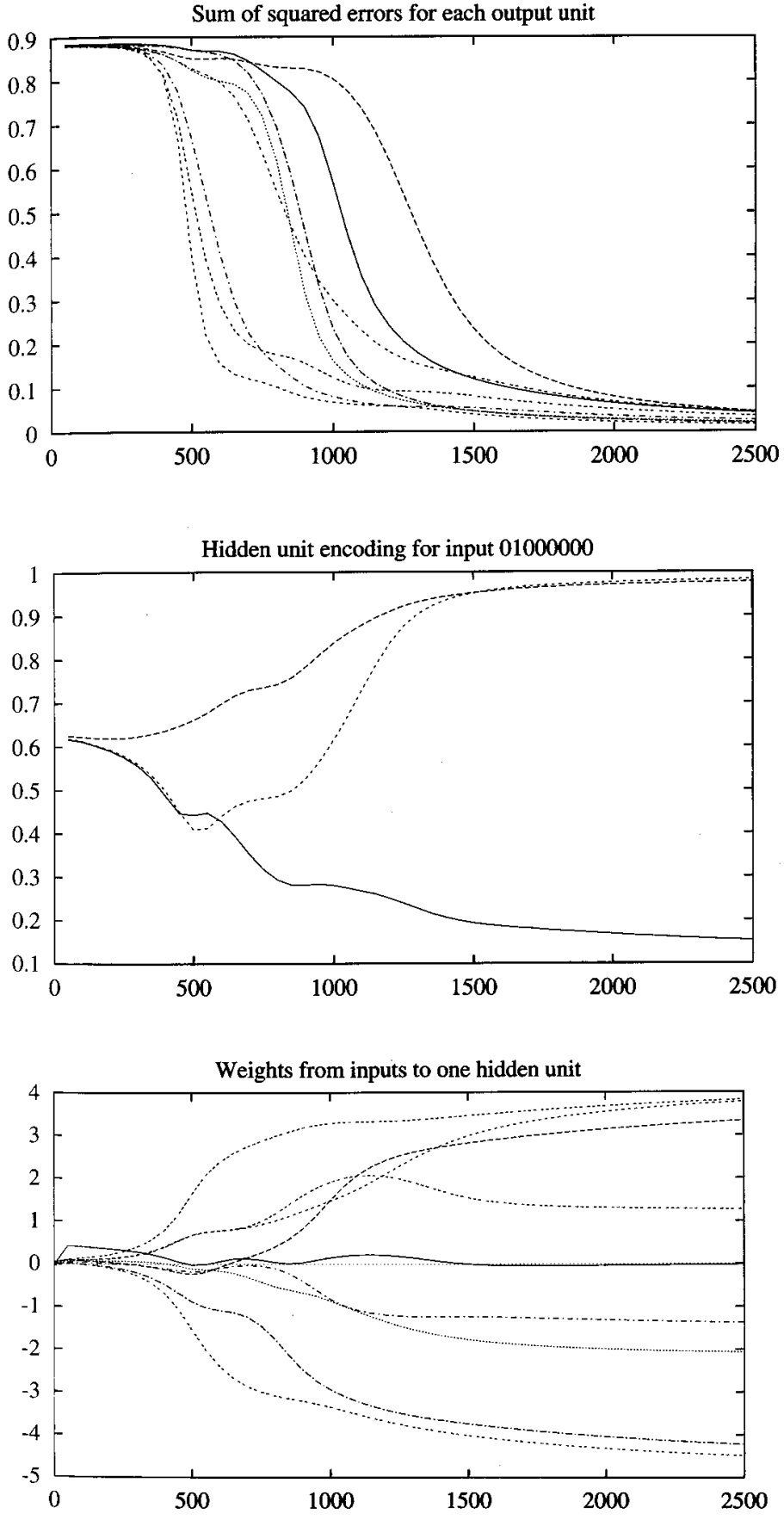


FIGURE 4.8
Learning the $8 \times 3 \times 8$ Network. The top plot shows the evolving sum of squared errors for each of the eight output units, as the number of training iterations (epochs) increases. The middle plot shows the evolving hidden layer representation for the input string "01000000." The bottom plot shows the evolving weights for one of the three hidden units.

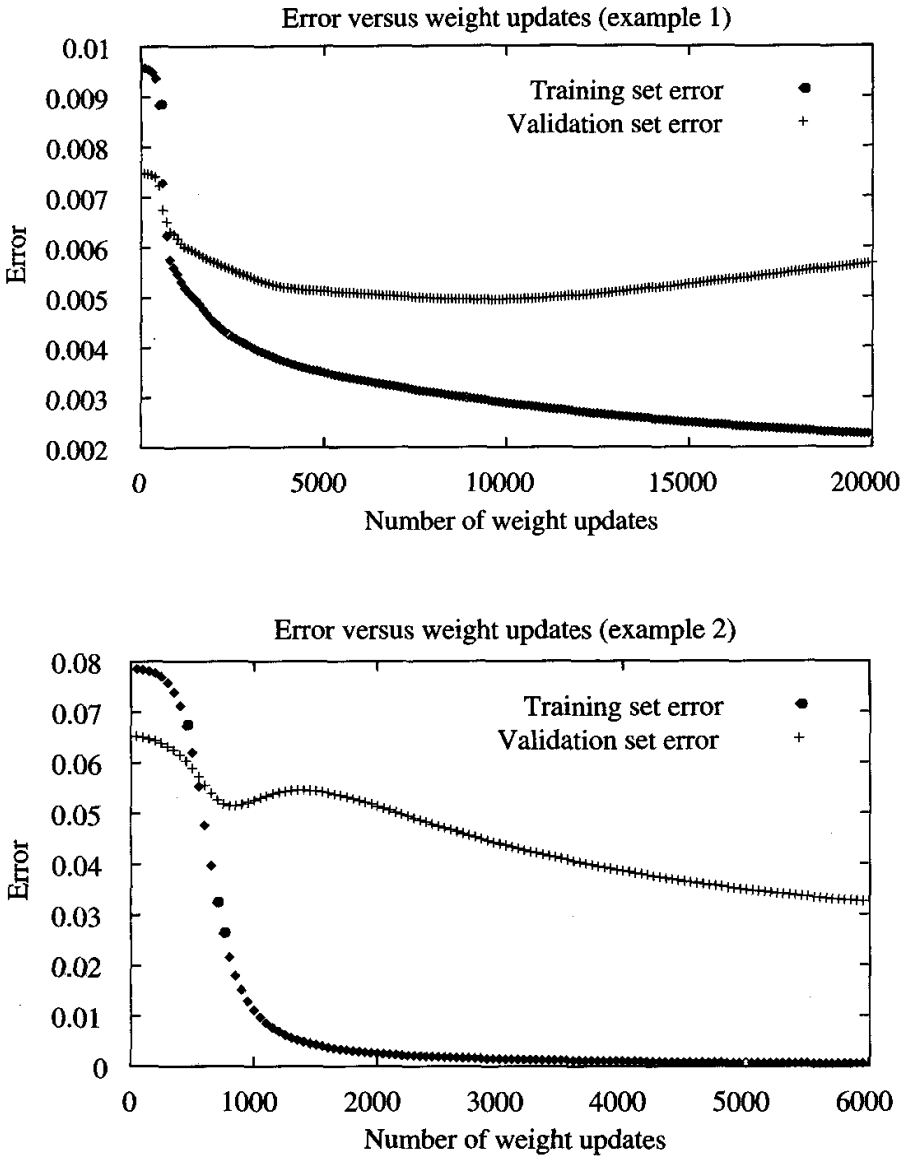


FIGURE 4.9
Plots of error E as a function of the number of weight updates, for two different robot perception tasks. In both learning cases, error E over the training examples decreases monotonically, as gradient descent minimizes this measure of error. Error over the separate "validation" set of examples typically decreases at first, then may later increase due to overfitting the training examples. The network most likely to generalize correctly to unseen data is the network with the lowest error over the validation set. Notice in the second plot, one must be careful to not stop training too soon when the validation set error begins to increase.

this variation for two fairly typical applications of BACKPROPAGATION. Consider first the top plot in this figure. The lower of the two lines shows the monotonically decreasing error E over the training set, as the number of gradient descent iterations grows. The upper line shows the error E measured over a different *validation set* of examples, distinct from the training examples. This line measures the *generalization accuracy* of the network—the accuracy with which it fits examples beyond the training data.

Notice the generalization accuracy measured over the validation examples first decreases, then increases, even as the error over the training examples continues to decrease. How can this occur? This occurs because the weights are being tuned to fit idiosyncrasies of the training examples that are not representative of the general distribution of examples. The large number of weight parameters in ANNs provides many degrees of freedom for fitting such idiosyncrasies.

Why does overfitting tend to occur during later iterations, but not during earlier iterations? Consider that network weights are initialized to small random values. With weights of nearly identical value, only very smooth decision surfaces are describable. As training proceeds, some weights begin to grow in order to reduce the error over the training data, and the complexity of the learned decision surface increases. Thus, the effective complexity of the hypotheses that can be reached by BACKPROPAGATION increases with the number of weight-tuning iterations. Given enough weight-tuning iterations, BACKPROPAGATION will often be able to create overly complex decision surfaces that fit noise in the training data or unrepresentative characteristics of the particular training sample. This overfitting problem is analogous to the overfitting problem in decision tree learning (see Chapter 3).

Several techniques are available to address the overfitting problem for BACKPROPAGATION learning. One approach, known as *weight decay*, is to decrease each weight by some small factor during each iteration. This is equivalent to modifying the definition of E to include a penalty term corresponding to the total magnitude of the network weights. The motivation for this approach is to keep weight values small, to bias learning against complex decision surfaces.

One of the most successful methods for overcoming the overfitting problem is to simply provide a set of validation data to the algorithm in addition to the training data. The algorithm monitors the error with respect to this validation set, while using the training set to drive the gradient descent search. In essence, this allows the algorithm itself to plot the two curves shown in Figure 4.9. How many weight-tuning iterations should the algorithm perform? Clearly, it should use the number of iterations that produces the lowest error *over the validation set,* since this is the best indicator of network performance over unseen examples. In typical implementations of this approach, two copies of the network weights are kept: one copy for training and a separate copy of the best-performing weights thus far, measured by their error over the validation set. Once the trained weights reach a significantly higher error over the validation set than the stored weights, training is terminated and the stored weights are returned as the final hypothesis. When this procedure is applied in the case of the top plot of Figure 4.9, it outputs the network weights obtained after 9100 iterations. The second plot in Figure 4.9 shows that it is not always obvious when the lowest error on the validation set has been reached. In this plot, the validation set error decreases, then increases, then decreases again. Care must be taken to avoid the mistaken conclusion that the network has reached its lowest validation set error at iteration 850.

In general, the issue of overfitting and how to overcome it is a subtle one. The above cross-validation approach works best when extra data are available to provide a validation set. Unfortunately, however, the problem of overfitting is most

severe for small training sets. In these cases, a k-fold cross-validation approach is sometimes used, in which cross validation is performed k different times, each time using a different partitioning of the data into training and validation sets, and the results are then averaged. In one version of this approach, the m available examples are partitioned into k disjoint subsets, each of size m/k. The cross-validation procedure is then run k times, each time using a different one of these subsets as the validation set and combining the other subsets for the training set. Thus, each example is used in the validation set for one of the experiments and in the training set for the other $k - 1$ experiments. On each experiment the above cross-validation approach is used to determine the number of iterations i that yield the best performance on the validation set. The mean $\bar{i}$ of these estimates for i is then calculated, and a final run of BACKPROPAGATION is performed *training on all n examples* for $\bar{i}$ iterations, with no validation set. This procedure is closely related to the procedure for comparing two learning methods based on limited data, described in Chapter 5.

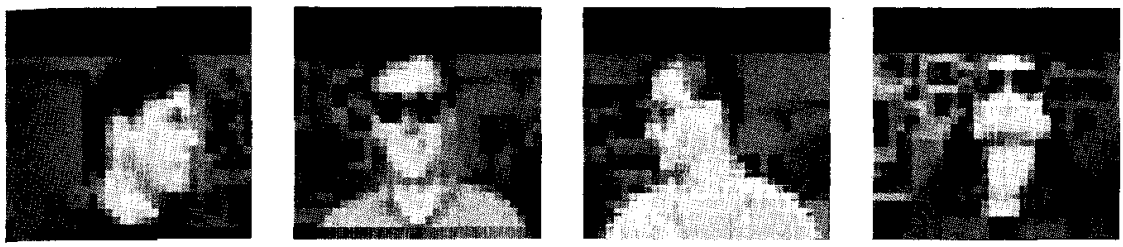
4.7 AN ILLUSTRATIVE EXAMPLE: FACE RECOGNITION

To illustrate some of the practical design choices involved in applying BACKPROPAGATION, this section discusses applying it to a learning task involving face recognition. All image data and code used to produce the examples described in this section are available at World Wide Web site http://www.cs.cmu.edu/~tom/mlbook.html, along with complete documentation on how to use the code. Why not try it yourself?

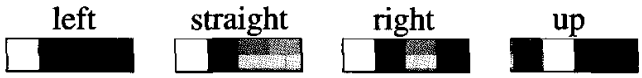
4.7.1 The Task

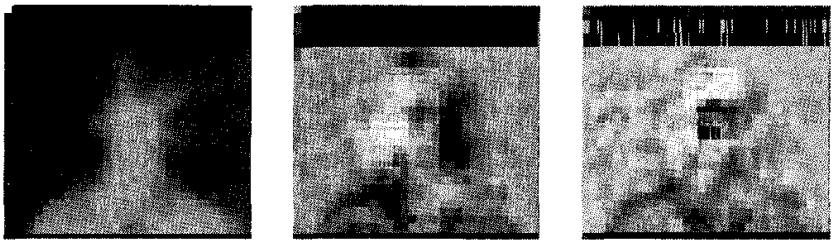
The learning task here involves classifying camera images of faces of various people in various poses. Images of 20 different people were collected, including approximately 32 images per person, varying the person's expression (happy, sad, angry, neutral), the direction in which they were looking (left, right, straight ahead, up), and whether or not they were wearing sunglasses. As can be seen from the example images in Figure 4.10, there is also variation in the background behind the person, the clothing worn by the person, and the position of the person's face within the image. In total, 624 greyscale images were collected, each with a resolution of 120×128, with each image pixel described by a greyscale intensity value between 0 (black) and 255 (white).

A variety of target functions can be learned from this image data. For example, given an image as input we could train an ANN to output the identity of the person, the direction in which the person is facing, the gender of the person, whether or not they are wearing sunglasses, etc. All of these target functions can be learned to high accuracy from this image data, and the reader is encouraged to try out these experiments. In the remainder of this section we consider one particular task: learning the direction in which the person is facing (to their left, right, straight ahead, or upward).

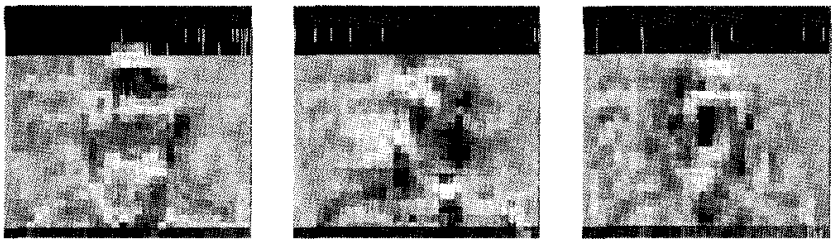


30 × 32 resolution input images





Network weights after 1 iteration through each training example



Network weights after 100 iterations through each training example

FIGURE 4.10
Learning an artificial neural network to recognize face pose. Here a 960 × 3 × 4 network is trained on grey-level images of faces (see top), to predict whether a person is looking to their left, right, ahead, or up. After training on 260 such images, the network achieves an accuracy of 90% over a separate test set. The learned network weights are shown after one weight-tuning iteration through the training examples and after 100 iterations. Each output unit (left, straight, right, up) has four weights, shown by dark (negative) and light (positive) blocks. The leftmost block corresponds to the weight w_0, which determines the unit threshold, and the three blocks to the right correspond to weights on inputs from the three hidden units. The weights from the image pixels into each hidden unit are also shown, with each weight plotted in the position of the corresponding image pixel.

4.7.2 Design Choices

In applying BACKPROPAGATION to any given task, a number of design choices must be made. We summarize these choices below for our task of learning the direction in which a person is facing. Although no attempt was made to determine the precise optimal design choices for this task, the design described here learns

the target function quite well. After training on a set of 260 images, classification accuracy over a separate test set is 90%. In contrast, the default accuracy achieved by randomly guessing one of the four possible face directions is 25%.

Input encoding. Given that the ANN input is to be some representation of the image, one key design choice is how to encode this image. For example, we could preprocess the image to extract edges, regions of uniform intensity, or other local image features, then input these features to the network. One difficulty with this design option is that it would lead to a variable number of features (e.g., edges) per image, whereas the ANN has a fixed number of input units. The design option chosen in this case was instead to encode the image as a fixed set of 30×32 pixel intensity values, with one network input per pixel. The pixel intensity values ranging from 0 to 255 were linearly scaled to range from 0 to 1 so that network inputs would have values in the same interval as the hidden unit and output unit activations. The 30×32 pixel image is, in fact, a coarse resolution summary of the original 120×128 captured image, with each coarse pixel intensity calculated as the mean of the corresponding high-resolution pixel intensities. Using this coarse-resolution image reduces the number of inputs and network weights to a much more manageable size, thereby reducing computational demands, while maintaining sufficient resolution to correctly classify the images. Recall from Figure 4.1 that the ALVINN system uses a similar coarse-resolution image as input to the network. One interesting difference is that in ALVINN, each coarse resolution pixel intensity is obtained by selecting the intensity of a single pixel at random from the appropriate region within the high-resolution image, rather than taking the mean of all pixel intensities within this region. The motivation for this in ALVINN is that it significantly reduces the computation required to produce the coarse-resolution image from the available high-resolution image. This efficiency is especially important when the network must be used to process many images per second while autonomously driving the vehicle.

Output encoding. The ANN must output one of four values indicating the direction in which the person is looking (left, right, up, or straight). Note we could encode this four-way classification using a single output unit, assigning outputs of, say, 0.2, 0.4, 0.6, and 0.8 to encode these four possible values. Instead, we use four distinct output units, each representing one of the four possible face directions, with the highest-valued output taken as the network prediction. This is often called a *1-of-n* output encoding. There are two motivations for choosing the 1-of-n output encoding over the single unit option. First, it provides more degrees of freedom to the network for representing the target function (i.e., there are n times as many weights available in the output layer of units). Second, in the 1-of-n encoding the difference between the highest-valued output and the second-highest can be used as a measure of the confidence in the network prediction (ambiguous classifications may result in near or exact ties). A further design choice here is "what should be the target values for these four output units?" One obvious choice would be to use the four target values $\langle 1, 0, 0, 0 \rangle$ to encode a face looking to the

left, $\langle 0, 1, 0, 0 \rangle$ to encode a face looking straight, etc. Instead of 0 and 1 values, we use values of 0.1 and 0.9, so that $\langle 0.9, 0.1, 0.1, 0.1 \rangle$ is the target output vector for a face looking to the left. The reason for avoiding target values of 0 and 1 is that sigmoid units cannot produce these output values given finite weights. If we attempt to train the network to fit target values of exactly 0 and 1, gradient descent will force the weights to grow without bound. On the other hand, values of 0.1 and 0.9 are achievable using a sigmoid unit with finite weights.

Network graph structure. As described earlier, BACKPROPAGATION can be applied to any acyclic directed graph of sigmoid units. Therefore, another design choice we face is how many units to include in the network and how to interconnect them. The most common network structure is a layered network with feedforward connections from every unit in one layer to every unit in the next. In the current design we chose this standard structure, using two layers of sigmoid units (one hidden layer and one output layer). It is common to use one or two layers of sigmoid units and, occasionally, three layers. It is not common to use more layers than this because training times become very long and because networks with three layers of sigmoid units can already express a rich variety of target functions (see Section 4.6.2). Given our choice of a layered feedforward network with one hidden layer, how many hidden units should we include? In the results reported in Figure 4.10, only three hidden units were used, yielding a test set accuracy of 90%. In other experiments 30 hidden units were used, yielding a test set accuracy one to two percent higher. Although the generalization accuracy varied only a small amount between these two experiments, the second experiment required significantly more training time. Using 260 training images, the training time was approximately 1 hour on a Sun Sparc5 workstation for the 30 hidden unit network, compared to approximately 5 minutes for the 3 hidden unit network. In many applications it has been found that some minimum number of hidden units is required in order to learn the target function accurately and that extra hidden units above this number do not dramatically affect generalization accuracy, provided cross-validation methods are used to determine how many gradient descent iterations should be performed. If such methods are not used, then increasing the number of hidden units often increases the tendency to overfit the training data, thereby reducing generalization accuracy.

Other learning algorithm parameters. In these learning experiments the learning rate η was set to 0.3, and the momentum α was set to 0.3. Lower values for both parameters produced roughly equivalent generalization accuracy, but longer training times. If these values are set too high, training fails to converge to a network with acceptable error over the training set. Full gradient descent was used in all these experiments (in contrast to the stochastic approximation to gradient descent in the algorithm of Table 4.2). Network weights in the output units were initialized to small random values. However, input unit weights were initialized to zero, because this yields much more intelligible visualizations of the learned weights (see Figure 4.10), without any noticeable impact on generalization accuracy. The

number of training iterations was selected by partitioning the available data into a training set and a separate validation set. Gradient descent was used to minimize the error over the training set, and after every 50 gradient descent steps the performance of the network was evaluated over the validation set. The final selected network was the one with the highest accuracy over the validation set. See Section 4.6.5 for an explanation and justification of this procedure. The final reported accuracy (e.g., 90% for the network in Figure 4.10) was measured over yet a third set of test examples that were not used in any way to influence training.

4.7.3 Learned Hidden Representations

It is interesting to examine the learned weight values for the 2899 weights in the network. Figure 4.10 depicts the values of each of these weights after one iteration through the weight update for all training examples, and again after 100 iterations.

To understand this diagram, consider first the four rectangular blocks just below the face images in the figure. Each of these rectangles depicts the weights for one of the four output units in the network (encoding left, straight, right, and up). The four squares within each rectangle indicate the four weights associated with this output unit—the weight w_0, which determines the unit threshold (on the left), followed by the three weights connecting the three hidden units to this output. The brightness of the square indicates the weight value, with bright white indicating a large positive weight, dark black indicating a large negative weight, and intermediate shades of grey indicating intermediate weight values. For example, the output unit labeled "up" has a near zero w_0 threshold weight, a large positive weight from the first hidden unit, and a large negative weight from the second hidden unit.

The weights of the hidden units are shown directly below those for the output units. Recall that each hidden unit receives an input from each of the 30×32 image pixels. The 30×32 weights associated with these inputs are displayed so that each weight is in the position of the corresponding image pixel (with the w_0 threshold weight superimposed in the top left of the array). Interestingly, one can see that the weights have taken on values that are especially sensitive to features in the region of the image in which the face and body typically appear.

The values of the network weights after 100 gradient descent iterations through each training example are shown at the bottom of the figure. Notice the leftmost hidden unit has very different weights than it had after the first iteration, and the other two hidden units have changed as well. It is possible to understand to some degree the encoding in this final set of weights. For example, consider the output unit that indicates a person is looking to his right. This unit has a strong positive weight from the second hidden unit and a strong negative weight from the third hidden unit. Examining the weights of these two hidden units, it is easy to see that if the person's face is turned to his right (i.e., our left), then his bright skin will roughly align with strong positive weights in this hidden unit, and his dark hair will roughly align with negative weights, resulting in this unit outputting a large value. The same image will cause the third hidden unit to output a value

close to zero, as the bright face will tend to align with the large negative weights in this case.

4.8 ADVANCED TOPICS IN ARTIFICIAL NEURAL NETWORKS

4.8.1 Alternative Error Functions

As noted earlier, gradient descent can be performed for any function E that is differentiable with respect to the parameterized hypothesis space. While the basic BACKPROPAGATION algorithm defines E in terms of the sum of squared errors of the network, other definitions have been suggested in order to incorporate other constraints into the weight-tuning rule. For each new definition of E a new weight-tuning rule for gradient descent must be derived. Examples of alternative definitions of E include

- Adding a penalty term for weight magnitude. As discussed above, we can add a term to E that increases with the magnitude of the weight vector. This causes the gradient descent search to seek weight vectors with small magnitudes, thereby reducing the risk of overfitting. One way to do this is to redefine E as

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 + \gamma \sum_{i,j} w_{ji}^2$$

  which yields a weight update rule identical to the BACKPROPAGATION rule, except that each weight is multiplied by the constant $(1 - 2\gamma\eta)$ upon each iteration. Thus, choosing this definition of E is equivalent to using a weight decay strategy (see Exercise 4.10.)
- Adding a term for errors in the *slope*, or derivative of the target function. In some cases, training information may be available regarding desired derivatives of the target function, as well as desired values. For example, Simard et al. (1992) describe an application to character recognition in which certain training derivatives are used to constrain the network to learn character recognition functions that are invariant of translation within the image. Mitchell and Thrun (1993) describe methods for calculating training derivatives based on the learner's prior knowledge. In both of these systems (described in Chapter 12), the error function is modified to add a term measuring the discrepancy between these training derivatives and the actual derivatives of the learned network. One example of such an error function is

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} \left[(t_{kd} - o_{kd})^2 + \mu \sum_{j \in inputs} \left(\frac{\partial t_{kd}}{\partial x_d^j} - \frac{\partial o_{kd}}{\partial x_d^j} \right)^2 \right]$$

  Here x_d^j denotes the value of the jth input unit for training example d. Thus, $\frac{\partial t_{kd}}{\partial x_d^j}$ is the training derivative describing how the target output value

t_{kd} should vary with a change in the input x_d^j. Similarly, $\frac{\partial o_{kd}}{\partial x_d^j}$ denotes the corresponding derivative of the actual learned network. The constant μ determines the relative weight placed on fitting the training values versus the training derivatives.

- Minimizing the *cross entropy* of the network with respect to the target values. Consider learning a probabilistic function, such as predicting whether a loan applicant will pay back a loan based on attributes such as the applicant's age and bank balance. Although the training examples exhibit only boolean target values (either a 1 or 0, depending on whether this applicant paid back the loan), the underlying target function might be best modeled by outputting the *probability* that the given applicant will repay the loan, rather than attempting to output the actual 1 and 0 value for each input instance. Given such situations in which we wish for the network to output probability estimates, it can be shown that the best (i.e., maximum likelihood) probability estimates are given by the network that minimizes the cross entropy, defined as

$$-\sum_{d \in D} t_d \log o_d + (1 - t_d) \log(1 - o_d)$$

Here o_d is the probability estimate output by the network for training example d, and t_d is the 1 or 0 target value for training example d. Chapter 6 discusses when and why the most probable network hypothesis is the one that minimizes this cross entropy and derives the corresponding gradient descent weight-tuning rule for sigmoid units. That chapter also describes other conditions under which the most probable hypothesis is the one that minimizes the sum of squared errors.

- Altering the effective error function can also be accomplished by weight sharing, or "tying together" weights associated with different units or inputs. The idea here is that different network weights are forced to take on identical values, usually to enforce some constraint known in advance to the human designer. For example, Waibel et al. (1989) and Lang et al. (1990) describe an application of neural networks to speech recognition, in which the network inputs are the speech frequency components at different times within a 144 millisecond time window. One assumption that can be made in this application is that the frequency components that identify a specific sound (e.g., "eee") should be independent of the exact time that the sound occurs within the 144 millisecond window. To enforce this constraint, the various units that receive input from different portions of the time window are forced to share weights. The net effect is to constrain the space of potential hypotheses, thereby reducing the risk of overfitting and improving the chances for accurately generalizing to unseen situations. Such weight sharing is typically implemented by first updating each of the shared weights separately within each unit that uses the weight, then replacing each instance of the shared weight by the mean of their values. The result of this procedure is that shared weights effectively adapt to a different error function than do the unshared weights.

4.8.2 Alternative Error Minimization Procedures

While gradient descent is one of the most general search methods for finding a hypothesis to minimize the error function, it is not always the most efficient. It is not uncommon for BACKPROPAGATION to require tens of thousands of iterations through the weight update loop when training complex networks. For this reason, a number of alternative weight optimization algorithms have been proposed and explored. To see some of the other possibilities, it is helpful to think of a weight-update method as involving two decisions: choosing a direction in which to alter the current weight vector and choosing a distance to move. In BACKPROPAGATION, the direction is chosen by taking the negative of the gradient, and the distance is determined by the learning rate constant η.

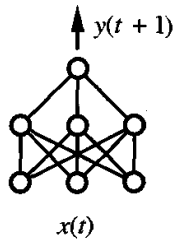
One optimization method, known as *line search*, involves a different approach to choosing the distance for the weight update. In particular, once a line is chosen that specifies the direction of the update, the update distance is chosen by finding the minimum of the error function along this line. Notice this can result in a very large or very small weight update, depending on the position of the point along the line that minimizes error. A second method, that builds on the idea of line search, is called the *conjugate gradient* method. Here, a sequence of line searches is performed to search for a minimum in the error surface. On the first step in this sequence, the direction chosen is the negative of the gradient. On each subsequent step, a new direction is chosen so that the component of the error gradient that has just been made zero, remains zero.

While alternative error-minimization methods sometimes lead to improved efficiency in training the network, methods such as conjugate gradient tend to have no significant impact on the generalization error of the final network. The only likely impact on the final error is that different error-minimization procedures may fall into different local minima. Bishop (1996) contains a general discussion of several parameter optimization methods for training networks.

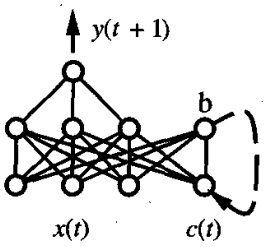
4.8.3 Recurrent Networks

Up to this point we have considered only network topologies that correspond to acyclic directed graphs. Recurrent networks are artificial neural networks that apply to time series data and that use outputs of network units at time t as the input to other units at time $t + 1$. In this way, they support a form of directed cycles in the network. To illustrate, consider the time series prediction task of predicting the next day's stock market average $y(t+1)$ based on the current day's economic indicators $x(t)$. Given a time series of such data, one obvious approach is to train a feedforward network to predict $y(t + 1)$ as its output, based on the input values $x(t)$. Such a network is shown in Figure 4.11(a).

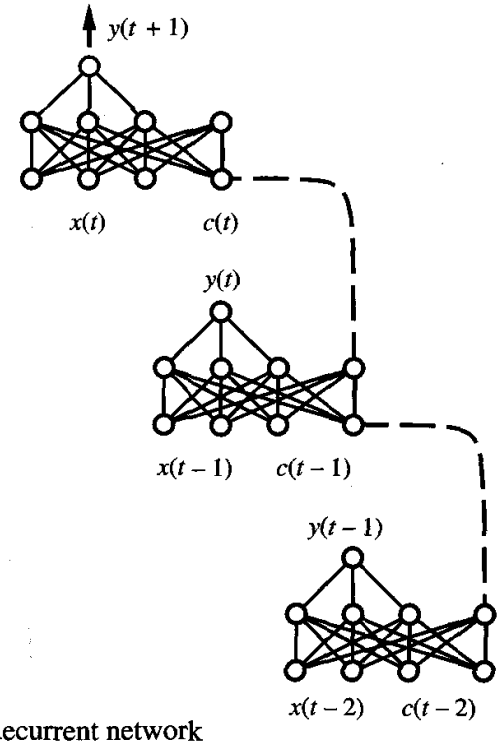
One limitation of such a network is that the prediction of $y(t + 1)$ depends only on $x(t)$ and cannot capture possible dependencies of $y(t+1)$ on earlier values of x. This might be necessary, for example, if tomorrow's stock market average $y(t + 1)$ depends on the difference between today's economic indicator values $x(t)$ and yesterday's values $x(t - 1)$. Of course we could remedy this difficulty



(a) Feedforward network



(b) Recurrent network



(c) Recurrent network unfolded in time

FIGURE 4.11
Recurrent networks.

by making both $x(t)$ and $x(t-1)$ inputs to the feedforward network. However, if we wish the network to consider an arbitrary window of time in the past when predicting $y(t+1)$, then a different solution is required. The recurrent network shown in Figure 4.11(b) provides one such solution. Here, we have added a new unit b to the hidden layer, and new input unit $c(t)$. The value of $c(t)$ is defined as the value of unit b at time $t-1$; that is, the input value $c(t)$ to the network at one time step is simply copied from the value of unit b on the previous time step. Notice this implements a recurrence relation, in which b represents information about the history of network inputs. Because b depends on both $x(t)$ and on $c(t)$, it is possible for b to summarize information from earlier values of x that are arbitrarily distant in time. Many other network topologies also can be used to

represent recurrence relations. For example, we could have inserted several layers of units between the input and unit b, and we could have added several context units in parallel where we added the single units b and c.

How can such recurrent networks be trained? There are several variants of recurrent networks, and several training methods have been proposed (see, for example, Jordan 1986; Elman 1990; Mozer 1995; Williams and Zipser 1995). Interestingly, recurrent networks such as the one shown in Figure 4.11(b) can be trained using a simple variant of BACKPROPAGATION. To understand how, consider Figure 4.11(c), which shows the data flow of the recurrent network "unfolded" in time. Here we have made several copies of the recurrent network, replacing the feedback loop by connections between the various copies. Notice that this large unfolded network contains no cycles. Therefore, the weights in the unfolded network can be trained directly using BACKPROPAGATION. Of course in practice we wish to keep only one copy of the recurrent network and one set of weights. Therefore, after training the unfolded network, the final weight w_{ji} in the recurrent network can be taken to be the mean value of the corresponding w_{ji} weights in the various copies. Mozer (1995) describes this training process in greater detail. In practice, recurrent networks are more difficult to train than networks with no feedback loops and do not generalize as reliably. However, they remain important due to their increased representational power.

4.8.4 Dynamically Modifying Network Structure

Up to this point we have considered neural network learning as a problem of adjusting weights within a fixed graph structure. A variety of methods have been proposed to dynamically grow or shrink the number of network units and interconnections in an attempt to improve generalization accuracy and training efficiency.

One idea is to begin with a network containing no hidden units, then grow the network as needed by adding hidden units until the training error is reduced to some acceptable level. The CASCADE-CORRELATION algorithm (Fahlman and Lebiere 1990) is one such algorithm. CASCADE-CORRELATION begins by constructing a network with no hidden units. In the case of our face-direction learning task, for example, it would construct a network containing only the four output units completely connected to the 30×32 input nodes. After this network is trained for some time, we may well find that there remains a significant residual error due to the fact that the target function cannot be perfectly represented by a network with this single-layer structure. In this case, the algorithm adds a hidden unit, choosing its weight values to maximize the correlation between the hidden unit value and the residual error of the overall network. The new unit is now installed into the network, with its weight values held fixed, and a new connection from this new unit is added to each output unit. The process is now repeated. The original weights are retrained (holding the hidden unit weights fixed), the residual error is checked, and a second hidden unit added if the residual error is still above threshold. Whenever a new hidden unit is added, its inputs include all of the original network inputs plus the outputs of any existing hidden units. The network is

grown in this fashion, accumulating hidden units until the network residual error is reduced to some acceptable level. Fahlman and Lebiere (1990) report cases in which CASCADE-CORRELATION significantly reduces training times, due to the fact that only a single layer of units is trained at each step. One practical difficulty is that because the algorithm can add units indefinitely, it is quite easy for it to overfit the training data, and precautions to avoid overfitting must be taken.

A second idea for dynamically altering network structure is to take the opposite approach. Instead of beginning with the simplest possible network and adding complexity, we begin with a complex network and prune it as we find that certain connections are inessential. One way to decide whether a particular weight is inessential is to see whether its value is close to zero. A second way, which appears to be more successful in practice, is to consider the effect that a small variation in the weight has on the error E. The effect on E of varying w (i.e., $\frac{\partial E}{\partial w}$) can be taken as a measure of the salience of the connection. LeCun et al. (1990) describe a process in which a network is trained, the least salient connections removed, and this process iterated until some termination condition is met. They refer to this as the "optimal brain damage" approach, because at each step the algorithm attempts to remove the least useful connections. They report that in a character recognition application this approach reduced the number of weights in a large network by a factor of 4, with a slight improvement in generalization accuracy and a significant improvement in subsequent training efficiency.

In general, techniques for dynamically modifying network structure have met with mixed success. It remains to be seen whether they can reliably improve on the generalization accuracy of BACKPROPAGATION. However, they have been shown in some cases to provide significant improvements in training times.

4.9 SUMMARY AND FURTHER READING

Main points of this chapter include:

- Artificial neural network learning provides a practical method for learning real-valued and vector-valued functions over continuous and discrete-valued attributes, in a way that is robust to noise in the training data. The BACKPROPAGATION algorithm is the most common network learning method and has been successfully applied to a variety of learning tasks, such as handwriting recognition and robot control.
- The hypothesis space considered by the BACKPROPAGATION algorithm is the space of all functions that can be represented by assigning weights to the given, fixed network of interconnected units. Feedforward networks containing three layers of units are able to approximate *any* function to arbitrary accuracy, given a sufficient (potentially very large) number of units in each layer. Even networks of practical size are capable of representing a rich space of highly nonlinear functions, making feedforward networks a good choice for learning discrete and continuous functions whose general form is unknown in advance.

- BACKPROPAGATION searches the space of possible hypotheses using gradient descent to iteratively reduce the error in the network fit to the training examples. Gradient descent converges to a local minimum in the training error with respect to the network weights. More generally, gradient descent is a potentially useful method for searching many continuously parameterized hypothesis spaces where the training error is a differentiable function of hypothesis parameters.
- One of the most intriguing properties of BACKPROPAGATION is its ability to invent new features that are not explicit in the input to the network. In particular, the internal (hidden) layers of multilayer networks learn to represent intermediate features that are useful for learning the target function and that are only implicit in the network inputs. This capability is illustrated, for example, by the ability of the $8 \times 3 \times 8$ network in Section 4.6.4 to invent the boolean encoding of digits from 1 to 8 and by the image features represented by the hidden layer in the face-recognition application of Section 4.7.
- Overfitting the training data is an important issue in ANN learning. Overfitting results in networks that generalize poorly to new data despite excellent performance over the training data. Cross-validation methods can be used to estimate an appropriate stopping point for gradient descent search and thus to minimize the risk of overfitting.
- Although BACKPROPAGATION is the most common ANN learning algorithm, many others have been proposed, including algorithms for more specialized tasks. For example, recurrent neural network methods train networks containing directed cycles, and algorithms such as CASCADE CORRELATION alter the network structure as well as the network weights.

Additional information on ANN learning can be found in several other chapters in this book. A Bayesian justification for choosing to minimize the sum of squared errors is given in Chapter 6, along with a justification for minimizing the cross-entropy instead of the sum of squared errors in other cases. Theoretical results characterizing the number of training examples needed to reliably learn boolean functions and the Vapnik-Chervonenkis dimension of certain types of networks can be found in Chapter 7. A discussion of overfitting and how to avoid it can be found in Chapter 5. Methods for using prior knowledge to improve the generalization accuracy of ANN learning are discussed in Chapter 12.

Work on artificial neural networks dates back to the very early days of computer science. McCulloch and Pitts (1943) proposed a model of a neuron that corresponds to the perceptron, and a good deal of work through the 1960s explored variations of this model. During the early 1960s Widrow and Hoff (1960) explored perceptron networks (which they called "adelines") and the delta rule, and Rosenblatt (1962) proved the convergence of the perceptron training rule. However, by the late 1960s it became clear that single-layer perceptron networks had limited representational capabilities, and no effective algorithms were known for training multilayer networks. Minsky and Papert (1969) showed that even

simple functions such as XOR could not be represented or learned with single-layer perceptron networks, and work on ANNs receded during the 1970s.

During the mid-1980s work on ANNs experienced a resurgence, caused in large part by the invention of BACKPROPAGATION and related algorithms for training multilayer networks (Rumelhart and McClelland 1986; Parker 1985). These ideas can be traced to related earlier work (e.g., Werbos 1975). Since the 1980s, BACKPROPAGATION has become a widely used learning method, and many other ANN approaches have been actively explored. The advent of inexpensive computers during this same period has allowed experimenting with computationally intensive algorithms that could not be thoroughly explored during the 1960s.

A number of textbooks are devoted to the topic of neural network learning. An early but still useful book on parameter learning methods for pattern recognition is Duda and Hart (1973). The text by Widrow and Stearns (1985) covers perceptrons and related single-layer networks and their applications. Rumelhart and McClelland (1986) produced an edited collection of papers that helped generate the increased interest in these methods beginning in the mid-1980s. Recent books on neural network learning include Bishop (1996); Chauvin and Rumelhart (1995); Freeman and Skapina (1991); Fu (1994); Hecht-Nielsen (1990); and Hertz et al. (1991).

EXERCISES

4.1. What are the values of weights w_0, w_1, and w_2 for the perceptron whose decision surface is illustrated in Figure 4.3? Assume the surface crosses the x_1 axis at -1, and the x_2 axis at 2.

4.2. Design a two-input perceptron that implements the boolean function $A \wedge \neg B$. Design a two-layer network of perceptrons that implements $A\ XOR\ B$.

4.3. Consider two perceptrons defined by the threshold expression $w_0 + w_1 x_1 + w_2 x_2 > 0$. Perceptron A has weight values

$$w_0 = 1,\ w_1 = 2,\ w_2 = 1$$

and perceptron B has the weight values

$$w_0 = 0,\ w_1 = 2,\ w_2 = 1$$

True or false? Perceptron A is *more_general_than* perceptron B. (*more_general_than* is defined in Chapter 2.)

4.4. Implement the delta training rule for a two-input linear unit. Train it to fit the target concept $-2 + x_1 + 2x_2 > 0$. Plot the error E as a function of the number of training iterations. Plot the decision surface after 5, 10, 50, 100, ..., iterations.

(*a*) Try this using various constant values for η and using a decaying learning rate of η_0/i for the ith iteration. Which works better?

(*b*) Try incremental and batch learning. Which converges more quickly? Consider both number of weight updates and total execution time.

4.5. Derive a gradient descent training rule for a single unit with output o, where

$$o = w_0 + w_1 x_1 + w_1 x_1^2 + \ldots + w_n x_n + w_n x_n^2$$

4.6. Explain informally why the delta training rule in Equation (4.10) is only an approximation to the true gradient descent rule of Equation (4.7).

4.7. Consider a two-layer feedforward ANN with two inputs a and b, one hidden unit c, and one output unit d. This network has five weights (w_{ca}, w_{cb}, w_{c0}, w_{dc}, w_{d0}), where w_{x0} represents the threshold weight for unit x. Initialize these weights to the values (.1, .1, .1, .1, .1), then give their values after each of the first two training iterations of the BACKPROPAGATION algorithm. Assume learning rate $\eta = .3$, momentum $\alpha = 0.9$, incremental weight updates, and the following training examples:

a	b	d
1	0	1
0	1	0

4.8. Revise the BACKPROPAGATION algorithm in Table 4.2 so that it operates on units using the squashing function *tanh* in place of the sigmoid function. That is, assume the output of a single unit is $o = tanh(\vec{w} \cdot \vec{x})$. Give the weight update rule for output layer weights and hidden layer weights. Hint: $tanh'(x) = 1 - tanh^2(x)$.

4.9. Recall the $8 \times 3 \times 8$ network described in Figure 4.7. Consider trying to train a $8 \times 1 \times 8$ network for the same task; that is, a network with just one hidden unit. Notice the eight training examples in Figure 4.7 could be represented by eight distinct values for the single hidden unit (e.g., 0.1, 0.2, ..., 0.8). Could a network with just one hidden unit therefore learn the identity function defined over these training examples? Hint: Consider questions such as "do there exist values for the hidden unit weights that can create the hidden unit encoding suggested above?" "do there exist values for the output unit weights that could correctly decode this encoding of the input?" and "is gradient descent likely to find such weights?"

4.10. Consider the alternative error function described in Section 4.8.1

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 + \gamma \sum_{i,j} w_{ji}^2$$

Derive the gradient descent update rule for this definition of E. Show that it can be implemented by multiplying each weight by some constant before performing the standard gradient descent update given in Table 4.2.

4.11. Apply BACKPROPAGATION to the task of face recognition. See World Wide Web URL http://www.cs.cmu.edu/~tom/book.html for details, including face-image data, BACKPROPAGATION code, and specific tasks.

4.12. Consider deriving a gradient descent algorithm to learn target concepts corresponding to rectangles in the x, y plane. Describe each hypothesis by the x and y coordinates of the lower-left and upper-right corners of the rectangle – llx, lly, urx, and ury respectively. An instance $\langle x, y \rangle$ is labeled positive by hypothesis $\langle llx, lly, urx, ury \rangle$ if and only if the point $\langle x, y \rangle$ lies inside the corresponding rectangle. Define error E as in the chapter. Can you devise a gradient descent algorithm to learn such rectangle hypotheses? Notice that E is not a continuous function of llx, lly, urx, and ury, just as in the case of perceptron learning. (Hint: Consider the two solutions used for perceptrons: (1) changing the classification rule to make output predictions *continuous* functions of the inputs, and (2) defining an alternative error—such as distance to the rectangle center—as in using the delta rule to train perceptrons.) Does your algorithm converge to the minimum error hypothesis when the positive and negative examples are separable by a rectangle? When they are not? Do you

have problems with local minima? How does your algorithm compare to symbolic methods for learning conjunctions of feature constraints?

REFERENCES

Bishop, C. M. (1996). *Neural networks for pattern recognition.* Oxford, England: Oxford University Press.

Chauvin, Y., & Rumelhart, D. (1995). BACKPROPAGATION: *Theory, architectures, and applications* (edited collection). Hillsdale, NJ: Lawrence Erlbaum Assoc.

Churchland, P. S., & Sejnowski, T. J. (1992). *The computational brain.* Cambridge, MA: The MIT Press.

Cybenko, G. (1988). Continuous valued neural networks with two hidden layers are sufficient (Technical Report). Department of Computer Science, Tufts University, Medford, MA.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2, 303–314.

Cottrell, G. W. (1990). Extracting features from faces using compression networks: Face, identity, emotion and gender recognition using holons. In D. Touretzky (Ed.), *Connection Models: Proceedings of the 1990 Summer School.* San Mateo, CA: Morgan Kaufmann.

Dietterich, T. G., Hild, H., & Bakiri, G. (1995). A comparison of ID3 and BACKPROPAGATION for English text-to-speech mapping. *Machine Learning*, 18(1), 51–80.

Duda, R., & Hart, P. (1973). *Pattern classification and scene analysis.* New York: John Wiley & Sons.

Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, 14, 179–211.

Fahlman, S., & Lebiere, C. (1990). *The* CASCADE-CORRELATION *learning architecture* (Technical Report CMU-CS-90-100). Computer Science Department, Carnegie Mellon University, Pittsburgh, PA.

Freeman, J. A., & Skapura, D. M. (1991). *Neural networks.* Reading, MA: Addison Wesley.

Fu, L. (1994). *Neural networks in computer intelligence.* New York: McGraw Hill.

Gabriel, M. & Moore, J. (1990). *Learning and computational neuroscience: Foundations of adaptive networks* (edited collection). Cambridge, MA: The MIT Press.

Hecht-Nielsen, R. (1990). *Neurocomputing.* Reading, MA: Addison Wesley.

Hertz, J., Krogh, A., & Palmer, R.G. (1991). *Introduction to the theory of neural computation.* Reading, MA: Addison Wesley.

Hornick, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2, 359–366.

Huang, W. Y., & Lippmann, R. P. (1988). Neural net and traditional classifiers. In Anderson (Ed.), *Neural Information Processing Systems* (pp. 387–396).

Jordan, M. (1986). Attractor dynamics and parallelism in a connectionist sequential machine. *Proceedings of the Eighth Annual Conference of the Cognitive Science Society* (pp. 531–546).

Kohonen, T. (1984). *Self-organization and associative memory.* Berlin: Springer-Verlag.

Lang, K. J., Waibel, A. H., & Hinton, G. E. (1990). A time-delay neural network architecture for isolated word recognition. *Neural Networks*, 3, 33–43.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L.D. (1989). BACKPROPAGATION applied to handwritten zip code recognition. *Neural Computation*, 1(4).

LeCun, Y., Denker, J. S., & Solla, S. A. (1990). Optimal brain damage. In D. Touretzky (Ed.), *Advances in Neural Information Processing Systems* (Vol. 2, pp. 598–605). San Mateo, CA: Morgan Kaufmann.

Manke, S., Finke, M. & Waibel, A. (1995). NPEN++: a writer independent, large vocabulary online cursive handwriting recognition system. *Proceedings of the International Conference on Document Analysis and Recognition.* Montreal, Canada: IEEE Computer Society.

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5, 115–133.

Mitchell, T. M., & Thrun, S. B. (1993). Explanation-based neural network learning for robot control. In Hanson, Cowan, & Giles (Eds.), *Advances in neural information processing systems 5* (pp. 287–294). San Francisco: Morgan Kaufmann.

Mozer, M. (1995). A focused BACKPROPAGATION algorithm for temporal pattern recognition. In Y. Chauvin & D. Rumelhart (Eds.), *Backpropagation: Theory, architectures, and applications* (pp. 137–169). Hillsdale, NJ: Lawrence Erlbaum Associates.

Minsky, M., & Papert, S. (1969). *Perceptrons*. Cambridge, MA: MIT Press.

Nilsson, N. J. (1965). *Learning machines*. New York: McGraw Hill.

Parker, D. (1985). *Learning logic* (MIT Technical Report TR-47). MIT Center for Research in Computational Economics and Management Science.

Pomerleau, D. A. (1993). Knowledge-based training of artificial neural networks for autonomous robot driving. In J. Connell & S. Mahadevan (Eds.), *Robot Learning* (pp. 19–43). Boston: Kluwer Academic Publishers.

Rosenblatt, F. (1959). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65, 386–408.

Rosenblatt, F. (1962). *Principles of neurodynamics*. New York: Spartan Books.

Rumelhart, D. E., & McClelland, J. L. (1986). *Parallel distributed processing: exploration in the microstructure of cognition* (Vols. 1 & 2). Cambridge, MA: MIT Press.

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

Shavlik, J. W., Mooney, R. J., & Towell, G. G. (1991). Symbolic and neural learning algorithms: An experimental comparison. *Machine Learning*, 6(2), 111–144.

Simard, P. S., Victorri, B., LeCun, Y., & Denker, J. (1992). Tangent prop—A formalism for specifying selected invariances in an adaptive network. In Moody, et al. (Eds.), *Advances in Neural Information Processing Systems 4* (pp. 895–903). San Francisco: Morgan Kaufmann.

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.*

Weiss, S., & Kapouleas, I. (1989). An empirical comparison of pattern recognition, neural nets, and machine learning classification methods. *Proceedings of the Eleventh IJCAI* (pp. 781–787). San Francisco: Morgan Kaufmann.

Werbos, P. (1975). Beyond regression: *New tools for prediction and analysis in the behavioral sciences* (Ph.D. dissertation). Harvard University.

Widrow, B., & Hoff, M. E. (1960). Adaptive switching circuits. *IRE WESCON Convention Record*, 4, 96–104.

Widrow, B., & Stearns, S. D. (1985). *Adaptive signal processing*. Signal Processing Series. Englewood Cliffs, NJ: Prentice Hall.

Williams, R., & Zipser, D. (1995). Gradient-based learning algorithms for recurrent networks and their computational complexity. In Y. Chauvin & D. Rumelhart (Eds.), *Backpropagation: Theory, architectures, and applications* (pp. 433–486). Hillsdale, NJ: Lawrence Erlbaum Associates.

Zornetzer, S. F., Davis, J. L., & Lau, C. (1994). *An introduction to neural and electronic networks* (edited collection) (2nd ed.). New York: Academic Press.

CHAPTER 5

EVALUATING HYPOTHESES

Empirically evaluating the accuracy of hypotheses is fundamental to machine learning. This chapter presents an introduction to statistical methods for estimating hypothesis accuracy, focusing on three questions. First, given the observed accuracy of a hypothesis over a limited sample of data, how well does this estimate its accuracy over additional examples? Second, given that one hypothesis outperforms another over some sample of data, how probable is it that this hypothesis is more accurate in general? Third, when data is limited what is the best way to use this data to both learn a hypothesis and estimate its accuracy? Because limited samples of data might misrepresent the general distribution of data, estimating true accuracy from such samples can be misleading. Statistical methods, together with assumptions about the underlying distributions of data, allow one to bound the difference between observed accuracy over the sample of available data and the true accuracy over the entire distribution of data.

5.1 MOTIVATION

In many cases it is important to evaluate the performance of learned hypotheses as precisely as possible. One reason is simply to understand whether to use the hypothesis. For instance, when learning from a limited-size database indicating the effectiveness of different medical treatments, it is important to understand as precisely as possible the accuracy of the learned hypotheses. A second reason is that evaluating hypotheses is an integral component of many learning methods. For example, in post-pruning decision trees to avoid overfitting, we must evaluate

the impact of possible pruning steps on the accuracy of the resulting decision tree. Therefore it is important to understand the likely errors inherent in estimating the accuracy of the pruned and unpruned tree.

Estimating the accuracy of a hypothesis is relatively straightforward when data is plentiful. However, when we must learn a hypothesis and estimate its future accuracy given only a limited set of data, two key difficulties arise:

- *Bias in the estimate.* First, the observed accuracy of the learned hypothesis over the training examples is often a poor estimator of its accuracy over future examples. Because the learned hypothesis was derived from these examples, they will typically provide an optimistically biased estimate of hypothesis accuracy over future examples. This is especially likely when the learner considers a very rich hypothesis space, enabling it to overfit the training examples. To obtain an unbiased estimate of future accuracy, we typically test the hypothesis on some set of test examples chosen independently of the training examples and the hypothesis.
- *Variance in the estimate.* Second, even if the hypothesis accuracy is measured over an unbiased set of test examples independent of the training examples, the measured accuracy can still vary from the true accuracy, depending on the makeup of the particular set of test examples. The smaller the set of test examples, the greater the expected variance.

This chapter discusses methods for evaluating learned hypotheses, methods for comparing the accuracy of two hypotheses, and methods for comparing the accuracy of two learning algorithms when only limited data is available. Much of the discussion centers on basic principles from statistics and sampling theory, though the chapter assumes no special background in statistics on the part of the reader. The literature on statistical tests for hypotheses is very large. This chapter provides an introductory overview that focuses only on the issues most directly relevant to learning, evaluating, and comparing hypotheses.

5.2 ESTIMATING HYPOTHESIS ACCURACY

When evaluating a learned hypothesis we are most often interested in estimating the accuracy with which it will classify future instances. At the same time, we would like to know the probable error in this accuracy estimate (i.e., what error bars to associate with this estimate).

Throughout this chapter we consider the following setting for the learning problem. There is some space of possible instances X (e.g., the set of all people) over which various target functions may be defined (e.g., people who plan to purchase new skis this year). We assume that different instances in X may be encountered with different frequencies. A convenient way to model this is to assume there is some unknown probability distribution $\mathcal{D}$ that defines the probability of encountering each instance in X (e.g., $\mathcal{D}$ might assign a higher probability to encountering 19-year-old people than 109-year-old people). Notice $\mathcal{D}$ says nothing

about whether x is a positive or negative example; it only determines the probability that x will be encountered. The learning task is to learn the target concept or target function f by considering a space H of possible hypotheses. Training examples of the target function f are provided to the learner by a trainer who draws each instance independently, according to the distribution $\mathcal{D}$, and who then forwards the instance x along with its correct target value $f(x)$ to the learner.

To illustrate, consider learning the target function "people who plan to purchase new skis this year," given a sample of training data collected by surveying people as they arrive at a ski resort. In this case the instance space X is the space of all people, who might be described by attributes such as their age, occupation, how many times they skied last year, etc. The distribution $\mathcal{D}$ specifies for each person x the probability that x will be encountered as the next person arriving at the ski resort. The target function $f : X \rightarrow \{0, 1\}$ classifies each person according to whether or not they plan to purchase skis this year.

Within this general setting we are interested in the following two questions:

1. Given a hypothesis h and a data sample containing n examples drawn at random according to the distribution $\mathcal{D}$, what is the best estimate of the accuracy of h over future instances drawn from the same distribution?
2. What is the probable error in this accuracy estimate?

5.2.1 Sample Error and True Error

To answer these questions, we need to distinguish carefully between two notions of accuracy or, equivalently, error. One is the error rate of the hypothesis over the sample of data that is available. The other is the error rate of the hypothesis over the entire unknown distribution $\mathcal{D}$ of examples. We will call these the *sample error* and the *true error* respectively.

The *sample error* of a hypothesis with respect to some sample S of instances drawn from X is the fraction of S that it misclassifies:

> ***Definition:*** The **sample error** (denoted $error_S(h)$) of hypothesis h with respect to target function f and data sample S is
>
> $$error_S(h) \equiv \frac{1}{n} \sum_{x \in S} \delta(f(x), h(x))$$
>
> Where n is the number of examples in S, and the quantity $\delta(f(x), h(x))$ is 1 if $f(x) \neq h(x)$, and 0 otherwise.

The *true error* of a hypothesis is the probability that it will misclassify a single randomly drawn instance from the distribution $\mathcal{D}$.

> ***Definition:*** The **true error** (denoted $error_{\mathcal{D}}(h)$) of hypothesis h with respect to target function f and distribution $\mathcal{D}$, is the probability that h will misclassify an instance drawn at random according to $\mathcal{D}$.
>
> $$error_{\mathcal{D}}(h) \equiv \Pr_{x \in \mathcal{D}}[f(x) \neq h(x)]$$

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

What we usually wish to know is the true error $error_{\mathcal{D}}(h)$ of the hypothesis, because this is the error we can expect when applying the hypothesis to future examples. All we can measure, however, is the sample error $error_S(h)$ of the hypothesis for the data sample S that we happen to have in hand. The main question considered in this section is "How good an estimate of $error_{\mathcal{D}}(h)$ is provided by $error_S(h)$?"

5.2.2 Confidence Intervals for Discrete-Valued Hypotheses

Here we give an answer to the question "How good an estimate of $error_{\mathcal{D}}(h)$ is provided by $error_S(h)$?" for the case in which h is a discrete-valued hypothesis. More specifically, suppose we wish to estimate the true error for some discrete-valued hypothesis h, based on its observed sample error over a sample S, where

- the sample S contains n examples drawn independent of one another, and independent of h, according to the probability distribution $\mathcal{D}$
- $n \geq 30$
- hypothesis h commits r errors over these n examples (i.e., $error_S(h) = r/n$).

Under these conditions, statistical theory allows us to make the following assertions:

1. Given no other information, the most probable value of $error_{\mathcal{D}}(h)$ is $error_S(h)$
2. With approximately 95% probability, the true error $error_{\mathcal{D}}(h)$ lies in the interval

$$error_S(h) \pm 1.96\sqrt{\frac{error_S(h)(1 - error_S(h))}{n}}$$

To illustrate, suppose the data sample S contains $n = 40$ examples and that hypothesis h commits $r = 12$ errors over this data. In this case, the sample error $error_S(h) = 12/40 = .30$. Given no other information, the best estimate of the true error $error_{\mathcal{D}}(h)$ is the observed sample error .30. However, we do not expect this to be a perfect estimate of the true error. If we were to collect a second sample S' containing 40 new randomly drawn examples, we might expect the sample error $error_{S'}(h)$ to vary slightly from the sample error $error_S(h)$. We expect a difference due to the random differences in the makeup of S and S'. In fact, if we repeated this experiment over and over, each time drawing a new sample S_i containing 40 new examples, we would find that for approximately 95% of these experiments, the calculated interval would contain the true error. For this reason, we call this interval the 95% confidence interval estimate for $error_{\mathcal{D}}(h)$. In the current example, where $r = 12$ and $n = 40$, the 95% confidence interval is, according to the above expression, $0.30 \pm (1.96 \cdot .07) = 0.30 \pm .14$.

Confidence level $N\%$:	50%	68%	80%	90%	95%	98%	99%
Constant z_N:	0.67	1.00	1.28	1.64	1.96	2.33	2.58

TABLE 5.1
Values of z_N for two-sided $N\%$ confidence intervals.

The above expression for the 95% confidence interval can be generalized to any desired confidence level. The constant 1.96 is used in case we desire a 95% confidence interval. A different constant, z_N, is used to calculate the $N\%$ confidence interval. The general expression for approximate $N\%$ confidence intervals for $error_{\mathcal{D}}(h)$ is

$$error_S(h) \pm z_N \sqrt{\frac{error_S(h)(1 - error_S(h))}{n}} \tag{5.1}$$

where the constant z_N is chosen depending on the desired confidence level, using the values of z_N given in Table 5.1.

Thus, just as we could calculate the 95% confidence interval for $error_{\mathcal{D}}(h)$ to be $0.30 \pm (1.96 \cdot .07)$ (when $r = 12$, $n = 40$), we can calculate the 68% confidence interval in this case to be $0.30 \pm (1.0 \cdot .07)$. Note it makes intuitive sense that the 68% confidence interval is smaller than the 95% confidence interval, because we have reduced the probability with which we demand that $error_{\mathcal{D}}(h)$ fall into the interval.

Equation (5.1) describes how to calculate the confidence intervals, or error bars, for estimates of $error_{\mathcal{D}}(h)$ that are based on $error_S(h)$. In using this expression, it is important to keep in mind that this applies only to discrete-valued hypotheses, that it assumes the sample S is drawn at random using the same distribution from which future data will be drawn, and that it assumes the data is independent of the hypothesis being tested. We should also keep in mind that the expression provides only an approximate confidence interval, though the approximation is quite good when the sample contains at least 30 examples, and $error_S(h)$ is not too close to 0 or 1. A more accurate rule of thumb is that the above approximation works well when

$$n\, error_S(h)(1 - error_S(h)) \geq 5$$

Above we summarized the procedure for calculating confidence intervals for discrete-valued hypotheses. The following section presents the underlying statistical justification for this procedure.

5.3 BASICS OF SAMPLING THEORY

This section introduces basic notions from statistics and sampling theory, including probability distributions, expected value, variance, Binomial and Normal distributions, and two-sided and one-sided intervals. A basic familiarity with these

- A *random variable* can be viewed as the name of an experiment with a probabilistic outcome. Its value is the outcome of the experiment.
- A *probability distribution* for a random variable Y specifies the probability $\Pr(Y = y_i)$ that Y will take on the value y_i, for each possible value y_i.
- The *expected value*, or *mean*, of a random variable Y is $E[Y] = \sum_i y_i \Pr(Y = y_i)$. The symbol μ_Y is commonly used to represent E[Y].
- The *variance* of a random variable is $Var(Y) = E[(Y - \mu_Y)^2]$. The variance characterizes the width or dispersion of the distribution about its mean.
- The *standard deviation* of Y is $\sqrt{Var(Y)}$. The symbol σ_Y is often used used to represent the standard deviation of Y.
- The *Binomial distribution* gives the probability of observing r heads in a series of n independent coin tosses, if the probability of heads in a single toss is p.
- The *Normal distribution* is a bell-shaped probability distribution that covers many natural phenomena.
- The *Central Limit Theorem* is a theorem stating that the sum of a large number of independent, identically distributed random variables approximately follows a Normal distribution.
- An *estimator* is a random variable Y used to estimate some parameter p of an underlying population.
- The *estimation bias* of Y as an estimator for p is the quantity $(E[Y] - p)$. An unbiased estimator is one for which the bias is zero.
- A *N% confidence interval* estimate for parameter p is an interval that includes p with probability $N\%$.

TABLE 5.2
Basic definitions and facts from statistics.

concepts is important to understanding how to evaluate hypotheses and learning algorithms. Even more important, these same notions provide an important conceptual framework for understanding machine learning issues such as overfitting and the relationship between successful generalization and the number of training examples considered. The reader who is already familiar with these notions may skip or skim this section without loss of continuity. The key concepts introduced in this section are summarized in Table 5.2.

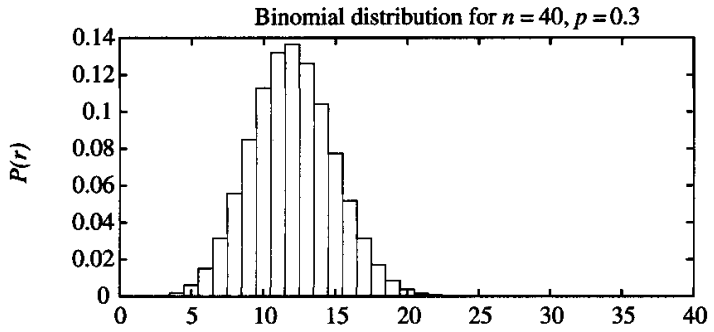
5.3.1 Error Estimation and Estimating Binomial Proportions

Precisely how does the deviation between sample error and true error depend on the size of the data sample? This question is an instance of a well-studied problem in statistics: the problem of estimating the proportion of a population that exhibits some property, given the observed proportion over some random sample of the population. In our case, the property of interest is that h misclassifies the example.

The key to answering this question is to note that when we measure the sample error we are performing an experiment with a random outcome. We first collect a random sample S of n independently drawn instances from the distribution $\mathcal{D}$, and then measure the sample error $error_S(h)$. As noted in the previous

section, if we were to repeat this experiment many times, each time drawing a different random sample S_i of size n, we would expect to observe different values for the various $error_{S_i}(h)$, depending on random differences in the makeup of the various S_i. We say in such cases that $error_{S_i}(h)$, the outcome of the ith such experiment, is a *random variable*. In general, one can think of a random variable as the name of an experiment with a random outcome. The value of the random variable is the observed outcome of the random experiment.

Imagine that we were to run k such random experiments, measuring the random variables $error_{S_1}(h), error_{S_2}(h) \ldots error_{S_k}(h)$. Imagine further that we then plotted a histogram displaying the frequency with which we observed each possible error value. As we allowed k to grow, the histogram would approach the form of the distribution shown in Table 5.3. This table describes a particular probability distribution called the *Binomial distribution*.



A *Binomial distribution* gives the probability of observing r heads in a sample of n independent coin tosses, when the probability of heads on a single coin toss is p. It is defined by the probability function

$$P(r) = \frac{n!}{r!(n-r)!} p^r (1-p)^{n-r}$$

If the random variable X follows a Binomial distribution, then:

- The probability $\Pr(X = r)$ that X will take on the value r is given by $P(r)$
- The expected, or mean value of X, $E[X]$, is

$$E[X] = np$$

- The variance of X, $Var(X)$, is

$$Var(X) = np(1-p)$$

- The standard deviation of X, σ_X, is

$$\sigma_X = \sqrt{np(1-p)}$$

For sufficiently large values of n the Binomial distribution is closely approximated by a Normal distribution (see Table 5.4) with the same mean and variance. Most statisticians recommend using the Normal approximation only when $np(1-p) \geq 5$.

TABLE 5.3
The Binomial distribution.

5.3.2 The Binomial Distribution

A good way to understand the Binomial distribution is to consider the following problem. You are given a worn and bent coin and asked to estimate the probability that the coin will turn up heads when tossed. Let us call this unknown probability of heads p. You toss the coin n times and record the number of times r that it turns up heads. A reasonable estimate of p is r/n. Note that if the experiment were rerun, generating a new set of n coin tosses, we might expect the number of heads r to vary somewhat from the value measured in the first experiment, yielding a somewhat different estimate for p. The Binomial distribution describes for each possible value of r (i.e., from 0 to n), the probability of observing exactly r heads given a sample of n independent tosses of a coin whose true probability of heads is p.

Interestingly, estimating p from a random sample of coin tosses is equivalent to estimating $error_{\mathcal{D}}(h)$ from testing h on a random sample of instances. A single toss of the coin corresponds to drawing a single random instance from $\mathcal{D}$ and determining whether it is misclassified by h. The probability p that a single random coin toss will turn up heads corresponds to the probability that a single instance drawn at random will be misclassified (i.e., p corresponds to $error_{\mathcal{D}}(h)$). The number r of heads observed over a sample of n coin tosses corresponds to the number of misclassifications observed over n randomly drawn instances. Thus r/n corresponds to $error_S(h)$. The problem of estimating p for coins is identical to the problem of estimating $error_{\mathcal{D}}(h)$ for hypotheses. The Binomial distribution gives the general form of the probability distribution for the random variable r, whether it represents the number of heads in n coin tosses or the number of hypothesis errors in a sample of n examples. The detailed form of the Binomial distribution depends on the specific sample size n and the specific probability p or $error_{\mathcal{D}}(h)$.

The general setting to which the Binomial distribution applies is:

1. There is a base, or underlying, experiment (e.g., toss of the coin) whose outcome can be described by a random variable, say Y. The random variable Y can take on two possible values (e.g., $Y = 1$ if heads, $Y = 0$ if tails).
2. The probability that $Y = 1$ on any single trial of the underlying experiment is given by some constant p, independent of the outcome of any other experiment. The probability that $Y = 0$ is therefore $(1 - p)$. Typically, p is not known in advance, and the problem is to estimate it.
3. A series of n independent trials of the underlying experiment is performed (e.g., n independent coin tosses), producing the sequence of independent, identically distributed random variables $Y_1, Y_2, \ldots, Y_n$. Let R denote the number of trials for which $Y_i = 1$ in this series of n experiments

$$R \equiv \sum_{i=1}^{n} Y_i$$

4. The probability that the random variable R will take on a specific value r (e.g., the probability of observing exactly r heads) is given by the Binomial distribution

$$\Pr(R = r) = \frac{n!}{r!(n-r)!} p^r (1-p)^{n-r} \tag{5.2}$$

A plot of this probability distribution is shown in Table 5.3.

The Binomial distribution characterizes the probability of observing r heads from n coin flip experiments, as well as the probability of observing r errors in a data sample containing n randomly drawn instances.

5.3.3 Mean and Variance

Two properties of a random variable that are often of interest are its expected value (also called its mean value) and its variance. The expected value is the average of the values taken on by repeatedly sampling the random variable. More precisely

Definition: Consider a random variable Y that takes on the possible values $y_1, \ldots y_n$. The **expected value** of Y, $E[Y]$, is

$$E[Y] \equiv \sum_{i=1}^{n} y_i \Pr(Y = y_i) \tag{5.3}$$

For example, if Y takes on the value 1 with probability .7 and the value 2 with probability .3, then its expected value is $(1 \cdot 0.7 + 2 \cdot 0.3 = 1.3)$. In case the random variable Y is governed by a Binomial distribution, then it can be shown that

$$E[Y] = np \tag{5.4}$$

where n and p are the parameters of the Binomial distribution defined in Equation (5.2).

A second property, the variance, captures the "width" or "spread" of the probability distribution; that is, it captures how far the random variable is expected to vary from its mean value.

Definition: The **variance** of a random variable Y, $Var[Y]$, is

$$Var[Y] \equiv E[(Y - E[Y])^2] \tag{5.5}$$

The variance describes the expected squared error in using a single observation of Y to estimate its mean $E[Y]$. The square root of the variance is called the *standard deviation* of Y, denoted σ_Y.

Definition: The **standard deviation** of a random variable Y, σ_Y, is

$$\sigma_Y \equiv \sqrt{E[(Y - E[Y])^2]} \tag{5.6}$$

In case the random variable Y is governed by a Binomial distribution, then the variance and standard deviation are given by

$$Var[Y] = np(1-p)$$
$$\sigma_Y = \sqrt{np(1-p)} \tag{5.7}$$

5.3.4 Estimators, Bias, and Variance

Now that we have shown that the random variable $error_S(h)$ obeys a Binomial distribution, we return to our primary question: What is the likely difference between $error_S(h)$ and the true error $error_{\mathcal{D}}(h)$?

Let us describe $error_S(h)$ and $error_{\mathcal{D}}(h)$ using the terms in Equation (5.2) defining the Binomial distribution. We then have

$$error_S(h) = \frac{r}{n}$$
$$error_{\mathcal{D}}(h) = p$$

where n is the number of instances in the sample S, r is the number of instances from S misclassified by h, and p is the probability of misclassifying a single instance drawn from $\mathcal{D}$.

Statisticians call $error_S(h)$ an *estimator* for the true error $error_{\mathcal{D}}(h)$. In general, an estimator is any random variable used to estimate some parameter of the underlying population from which the sample is drawn. An obvious question to ask about any estimator is whether on average it gives the right estimate. We define the *estimation bias* to be the difference between the expected value of the estimator and the true value of the parameter.

Definition: The **estimation bias** of an estimator Y for an arbitrary parameter p is

$$E[Y] - p$$

If the estimation bias is zero, we say that Y is an *unbiased estimator* for p. Notice this will be the case if the average of many random values of Y generated by repeated random experiments (i.e., $E[Y]$) converges toward p.

Is $error_S(h)$ an unbiased estimator for $error_{\mathcal{D}}(h)$? Yes, because for a Binomial distribution the expected value of r is equal to np (Equation [5.4]). It follows, given that n is a constant, that the expected value of r/n is p.

Two quick remarks are in order regarding the estimation bias. First, when we mentioned at the beginning of this chapter that testing the hypothesis on the training examples provides an optimistically biased estimate of hypothesis error, it is exactly this notion of estimation bias to which we were referring. In order for $error_S(h)$ to give an unbiased estimate of $error_{\mathcal{D}}(h)$, the hypothesis h and sample S must be chosen independently. Second, this notion of *estimation bias* should not be confused with the *inductive bias* of a learner introduced in Chapter 2. The

estimation bias is a numerical quantity, whereas the inductive bias is a set of assertions.

A second important property of any estimator is its variance. Given a choice among alternative unbiased estimators, it makes sense to choose the one with least variance. By our definition of variance, this choice will yield the smallest expected squared error between the estimate and the true value of the parameter.

To illustrate these concepts, suppose we test a hypothesis and find that it commits $r = 12$ errors on a sample of $n = 40$ randomly drawn test examples. Then an unbiased estimate for $error_{\mathcal{D}}(h)$ is given by $error_S(h) = r/n = 0.3$. The variance in this estimate arises completely from the variance in r, because n is a constant. Because r is Binomially distributed, its variance is given by Equation (5.7) as $np(1-p)$. Unfortunately p is unknown, but we can substitute our estimate r/n for p. This yields an estimated variance in r of $40 \cdot 0.3(1 - 0.3) = 8.4$, or a corresponding standard deviation of $\sqrt{8.4} \approx 2.9$. This implies that the standard deviation in $error_S(h) = r/n$ is approximately $2.9/40 = .07$. To summarize, $error_S(h)$ in this case is observed to be 0.30, with a standard deviation of approximately 0.07. (See Exercise 5.1.)

In general, given r errors in a sample of n independently drawn test examples, the standard deviation for $error_S(h)$ is given by

$$\sigma_{error_S(h)} = \frac{\sigma_r}{n} = \sqrt{\frac{p(1-p)}{n}} \tag{5.8}$$

which can be approximated by substituting $r/n = error_S(h)$ for p

$$\sigma_{error_S(h)} \approx \sqrt{\frac{error_S(h)(1 - error_S(h))}{n}} \tag{5.9}$$

5.3.5 Confidence Intervals

One common way to describe the uncertainty associated with an estimate is to give an interval within which the true value is expected to fall, along with the probability with which it is expected to fall into this interval. Such estimates are called *confidence interval* estimates.

> ***Definition:*** An $N\%$ **confidence interval** for some parameter p is an interval that is expected with probability $N\%$ to contain p.

For example, if we observe $r = 12$ errors in a sample of $n = 40$ independently drawn examples, we can say with approximately 95% probability that the interval 0.30 ± 0.14 contains the true error $error_{\mathcal{D}}(h)$.

How can we derive confidence intervals for $error_{\mathcal{D}}(h)$? The answer lies in the fact that we know the Binomial probability distribution governing the estimator $error_S(h)$. The mean value of this distribution is $error_{\mathcal{D}}(h)$, and the standard deviation is given by Equation (5.9). Therefore, to derive a 95% confidence interval, we need only find the interval centered around the mean value $error_{\mathcal{D}}(h)$,