Preface

These notes are in the process of becoming a textbook. The process is quite unfinished, and the author solicits corrections, criticisms, and suggestions from students and other readers. Although I have tried to eliminate errors, some undoubtedly remain—*caveat lector*. Many typographical infelicities will no doubt persist until the final version. More material has yet to be added. Please let me have your suggestions about topics that are too important to be left out. I hope that future versions will cover Hopfield nets, Elman nets and other recurrent nets, radial basis functions, grammar and automata learning, genetic algorithms, and Bayes networks .... I am also collecting exercises and project suggestions which will appear in future versions.

Some of my plans for additions and other reminders are mentioned in marginal notes.

My intention is to pursue a middle ground between a theoretical textbook and one that focusses on applications. The book concentrates on the important *ideas* in machine learning. I do not give proofs of many of the theorems that I state, but I do give plausibility arguments and citations to formal proofs. And, I do not treat many matters that would be of practical importance in applications; the book is not a handbook of machine learning practice. Instead, my goal is to give the reader sufficient preparation to make the extensive literature on machine learning accessible.

Students in my Stanford courses on machine learning have already made several useful suggestions, as have my colleague, Pat Langley, and my teaching assistants, Ron Kohavi, Karl Pfleger, Robert Allen, and Lise Getoor.

Chapter 1

Preliminaries

1.1 Introduction

1.1.1 What is Machine Learning?

Learning, like intelligence, covers such a broad range of processes that it is difficult to define precisely. A dictionary definition includes phrases such as "to gain knowledge, or understanding of, or skill in, by study, instruction, or experience," and "modification of a behavioral tendency by experience." Zoologists and psychologists study learning in animals and humans. In this book we focus on learning in machines. There are several parallels between animal and machine learning. Certainly, many techniques in machine learning derive from the efforts of psychologists to make more precise their theories of animal and human learning through computational models. It seems likely also that the concepts and techniques being explored by researchers in machine learning may illuminate certain aspects of biological learning.

As regards machines, we might say, very broadly, that a machine learns whenever it changes its structure, program, or data (based on its inputs or in response to external information) in such a manner that its expected future performance improves. Some of these changes, such as the addition of a record to a data base, fall comfortably within the province of other disciplines and are not necessarily better understood for being called learning. But, for example, when the performance of a speech-recognition machine improves after hearing several samples of a person's speech, we feel quite justified in that case to say that the machine has learned.

Machine learning usually refers to the changes in systems that perform tasks associated with *artificial intelligence (AI)*. Such tasks involve recognition, diagnosis, planning, robot control, prediction, etc. The "changes" might be either enhancements to already performing systems or *ab initio* synthesis of new systems. To be slightly more specific, we show the architecture of a typical AI

"agent" in Fig. 1.1. This agent perceives and models its environment and computes appropriate actions, perhaps by anticipating their effects. Changes made to any of the components shown in the figure might count as learning. Different learning mechanisms might be employed depending on which subsystem is being changed. We will study several different learning methods in this book.

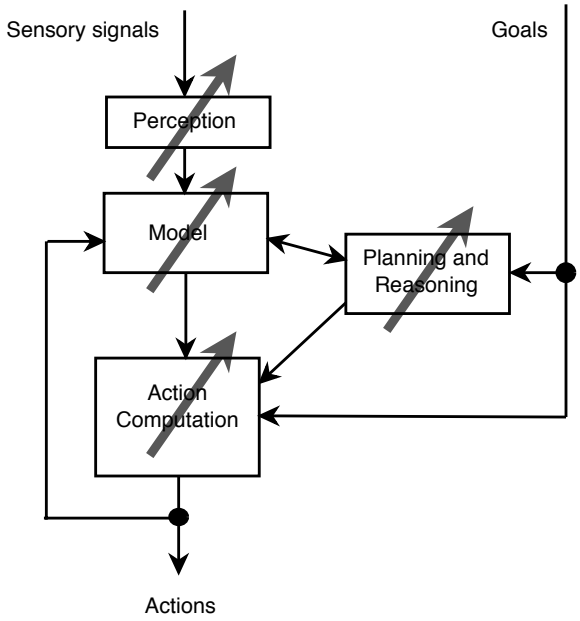

Figure 1.1: An AI System

One might ask "Why should machines have to learn? Why not design machines to perform as desired in the first place?" There are several reasons why machine learning is important. Of course, we have already mentioned that the achievement of learning in machines might help us understand how animals and humans learn. But there are important engineering reasons as well. Some of these are:

- Some tasks cannot be defined well except by example; that is, we might be able to specify input/output pairs but not a concise relationship between inputs and desired outputs. We would like machines to be able to adjust their internal structure to produce correct outputs for a large number of sample inputs and thus suitably constrain their input/output function to approximate the relationship implicit in the examples.
- It is possible that hidden among large piles of data are important relationships and correlations. Machine learning methods can often be used to extract these relationships (*data mining*).

- Human designers often produce machines that do not work as well as desired in the environments in which they are used. In fact, certain characteristics of the working environment might not be completely known at design time. Machine learning methods can be used for on-the-job improvement of existing machine designs.
- The amount of knowledge available about certain tasks might be too large for explicit encoding by humans. Machines that learn this knowledge gradually might be able to capture more of it than humans would want to write down.
- Environments change over time. Machines that can adapt to a changing environment would reduce the need for constant redesign.
- New knowledge about tasks is constantly being discovered by humans. Vocabulary changes. There is a constant stream of new events in the world. Continuing redesign of AI systems to conform to new knowledge is impractical, but machine learning methods might be able to track much of it.

1.1.2 Wellsprings of Machine Learning

Work in machine learning is now converging from several sources. These different traditions each bring different methods and different vocabulary which are now being assimilated into a more unified discipline. Here is a brief listing of some of the separate disciplines that have contributed to machine learning; more details will follow in the the appropriate chapters:

- **Statistics:** A long-standing problem in statistics is how best to use samples drawn from unknown probability distributions to help decide from which distribution some new sample is drawn. A related problem is how to estimate the value of an unknown function at a new point given the values of this function at a set of sample points. Statistical methods for dealing with these problems can be considered instances of machine learning because the decision and estimation rules depend on a corpus of samples drawn from the problem environment. We will explore some of the statistical methods later in the book. Details about the statistical theory underlying these methods can be found in statistical textbooks such as [Anderson, 1958].
- **Brain Models:** Non-linear elements with weighted inputs have been suggested as simple models of biological neurons. Networks of these elements have been studied by several researchers including [McCulloch & Pitts, 1943, Hebb, 1949, Rosenblatt, 1958] and, more recently by [Gluck & Rumelhart, 1989, Sejnowski, Koch, & Churchland, 1988]. Brain modelers are interested in how closely these networks approximate the learning phenomena of

living brains. We shall see that several important machine learning techniques are based on networks of nonlinear elements—often called *neural networks*. Work inspired by this school is sometimes called *connectionism*, *brain-style computation*, or *sub-symbolic processing*.

- **Adaptive Control Theory:** Control theorists study the problem of controlling a process having unknown parameters which must be estimated during operation. Often, the parameters change during operation, and the control process must track these changes. Some aspects of controlling a robot based on sensory inputs represent instances of this sort of problem. For an introduction see [Bollinger & Duffie, 1988].

- **Psychological Models:** Psychologists have studied the performance of humans in various learning tasks. An early example is the EPAM network for storing and retrieving one member of a pair of words when given another [Feigenbaum, 1961]. Related work led to a number of early decision tree [Hunt, Marin, & Stone, 1966] and semantic network [Anderson & Bower, 1973] methods. More recent work of this sort has been influenced by activities in artificial intelligence which we will be presenting.

  Some of the work in reinforcement learning can be traced to efforts to model how reward stimuli influence the learning of goal-seeking behavior in animals [Sutton & Barto, 1987]. Reinforcement learning is an important theme in machine learning research.

- **Artificial Intelligence:** From the beginning, AI research has been concerned with machine learning. Samuel developed a prominent early program that learned parameters of a function for evaluating board positions in the game of checkers [Samuel, 1959]. AI researchers have also explored the role of analogies in learning [Carbonell, 1983] and how future actions and decisions can be based on previous exemplary cases [Kolodner, 1993]. Recent work has been directed at discovering rules for expert systems using decision-tree methods [Quinlan, 1990] and inductive logic programming [Muggleton, 1991, Lavrač & Džeroski, 1994]. Another theme has been saving and generalizing the results of problem solving using explanation-based learning [DeJong & Mooney, 1986, Laird, *et al.*, 1986, Minton, 1988, Etzioni, 1993].

- **Evolutionary Models:**

  In nature, not only do individual animals learn to perform better, but species *evolve* to be better fit in their individual niches. Since the distinction between evolving and learning can be blurred in computer systems, techniques that model certain aspects of biological evolution have been proposed as learning methods to improve the performance of computer programs. Genetic algorithms [Holland, 1975] and genetic programming [Koza, 1992, Koza, 1994] are the most prominent computational techniques for evolution.

1.1.3 Varieties of Machine Learning

Orthogonal to the question of the historical source of any learning technique is the more important question of *what* is to be learned. In this book, we take it that the thing to be learned is a computational structure of some sort. We will consider a variety of different computational structures:

- Functions
- Logic programs and rule sets
- Finite-state machines
- Grammars
- Problem solving systems

We will present methods both for the synthesis of these structures from examples and for changing existing structures. In the latter case, the change to the existing structure might be simply to make it more computationally efficient rather than to increase the coverage of the situations it can handle. Much of the terminology that we shall be using throughout the book is best introduced by discussing the problem of learning functions, and we turn to that matter first.

1.2 Learning Input-Output Functions

We use Fig. 1.2 to help define some of the terminology used in describing the problem of learning a function. Imagine that there is a function, f, and the task of the learner is to guess what it is. Our hypothesis about the function to be learned is denoted by h. Both f and h are functions of a vector-valued input $\mathbf{X} = (x_1, x_2, \ldots, x_i, \ldots, x_n)$ which has n components. We think of h as being implemented by a device that has $\mathbf{X}$ as input and $h(\mathbf{X})$ as output. Both f and h themselves may be vector-valued. We assume *a priori* that the hypothesized function, h, is selected from a class of functions $\mathcal{H}$. Sometimes we know that f also belongs to this class or to a subset of this class. We select h based on a *training set*, Ξ, of m input vector examples. Many important details depend on the nature of the assumptions made about all of these entities.

1.2.1 Types of Learning

There are two major settings in which we wish to learn a function. In one, called *supervised learning*, we know (sometimes only approximately) the values of f for the m samples in the training set, Ξ. We assume that if we can find a hypothesis, h, that closely agrees with f for the members of Ξ, then this hypothesis will be a good guess for f—especially if Ξ is large.

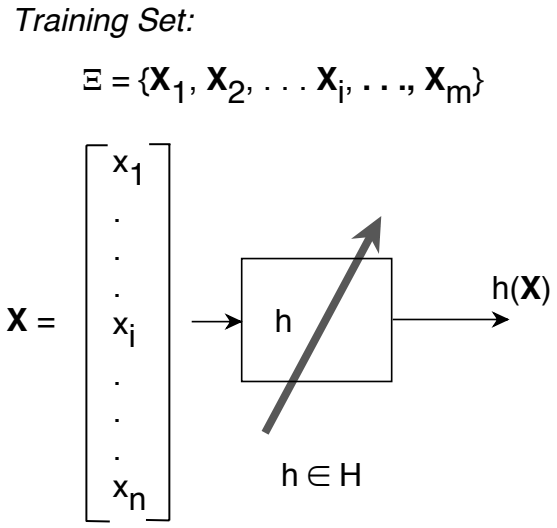

Figure 1.2: An Input-Output Function

Curve-fitting is a simple example of supervised learning of a function. Suppose we are given the values of a two-dimensional function, f, at the four sample points shown by the solid circles in Fig. 1.3. We want to fit these four points with a function, h, drawn from the set, $\mathcal{H}$, of second-degree functions. We show there a two-dimensional parabolic surface above the x_1, x_2 plane that fits the points. This parabolic function, h, is our hypothesis about the function, f, that produced the four samples. In this case, $h = f$ at the four samples, but we need not have required exact matches.

In the other setting, termed *unsupervised learning*, we simply have a training set of vectors without function values for them. The problem in this case, typically, is to partition the training set into subsets, $\Xi_1, \ldots, \Xi_R$, in some appropriate way. (We can still regard the problem as one of learning a function; the value of the function is the name of the subset to which an input vector belongs.) Unsupervised learning methods have application in taxonomic problems in which it is desired to invent ways to classify data into meaningful categories.

We shall also describe methods that are intermediate between supervised and unsupervised learning.

We might either be trying to find a new function, h, or to modify an existing one. An interesting special case is that of changing an existing function into an equivalent one that is computationally more efficient. This type of learning is sometimes called *speed-up* learning. A very simple example of speed-up learning involves deduction processes. From the formulas $A \supset B$ and $B \supset C$, we can deduce C if we are given A. From this deductive process, we can create the formula $A \supset C$—a new formula but one that does not sanction any more con-

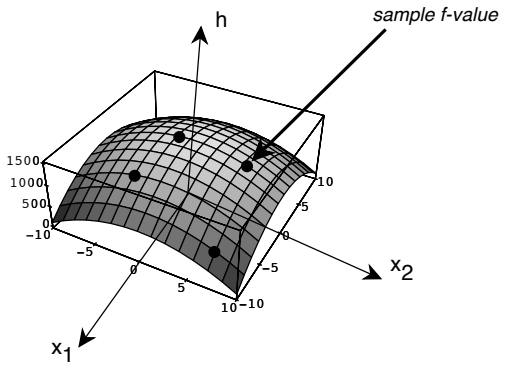

Figure 1.3: A Surface that Fits Four Points

clusions than those that could be derived from the formulas that we previously had. But with this new formula we can derive C more quickly, given A, than we could have done before. We can contrast speed-up learning with methods that create genuinely new functions—ones that might give different results after learning than they did before. We say that the latter methods involve *inductive* learning. As opposed to deduction, there are no *correct* inductions—only useful ones.

1.2.2 Input Vectors

Because machine learning methods derive from so many different traditions, its terminology is rife with synonyms, and we will be using most of them in this book. For example, the input vector is called by a variety of names. Some of these are: *input vector*, *pattern vector*, *feature vector*, *sample*, *example*, and *instance*. The components, x_i, of the input vector are variously called *features*, *attributes*, *input variables*, and *components*.

The values of the components can be of three main types. They might be real-valued numbers, discrete-valued numbers, or *categorical values*. As an example illustrating categorical values, information about a student might be represented by the values of the attributes *class, major, sex, adviser*. A particular student would then be represented by a vector such as: (sophomore, history, male, higgins). Additionally, categorical values may be *ordered* (as in {*small, medium, large*}) or *unordered* (as in the example just given). Of course, mixtures of all these types of values are possible.

In all cases, it is possible to represent the input in unordered form by listing the names of the attributes together with their values. The vector form assumes that the attributes are ordered and given implicitly by a form. As an example of an *attribute-value* representation, we might have: (major: history, sex: male,

class: sophomore, adviser: higgins, age: 19). We will be using the vector form exclusively.

An important specialization uses Boolean values, which can be regarded as a special case of either discrete numbers (1,0) or of categorical variables (*True*, *False*).

1.2.3 Outputs

The output may be a real number, in which case the process embodying the function, h, is called a *function estimator*, and the output is called an *output value* or *estimate*.

Alternatively, the output may be a categorical value, in which case the process embodying h is variously called a *classifier*, a *recognizer*, or a *categorizer*, and the output itself is called a *label*, a *class*, a *category*, or a *decision*. Classifiers have application in a number of recognition problems, for example in the recognition of hand-printed characters. The input in that case is some suitable representation of the printed character, and the classifier maps this input into one of, say, 64 categories.

Vector-valued outputs are also possible with components being real numbers or categorical values.

An important special case is that of Boolean output values. In that case, a training pattern having value 1 is called a *positive instance*, and a training sample having value 0 is called a *negative instance*. When the input is also Boolean, the classifier implements a *Boolean function*. We study the Boolean case in some detail because it allows us to make important general points in a simplified setting. Learning a Boolean function is sometimes called *concept learning*, and the function is called a *concept*.

1.2.4 Training Regimes

There are several ways in which the training set, Ξ, can be used to produce a hypothesized function. In the *batch* method, the entire training set is available and used all at once to compute the function, h. A variation of this method uses the entire training set to modify a current hypothesis iteratively until an acceptable hypothesis is obtained. By contrast, in the *incremental* method, we select one member at a time from the training set and use this instance alone to modify a current hypothesis. Then another member of the training set is selected, and so on. The selection method can be random (with replacement) or it can cycle through the training set iteratively. If the entire training set becomes available one member at a time, then we might also use an incremental method—selecting and using training set members as they arrive. (Alternatively, at any stage all training set members so far available could be used in a "batch" process.) Using the training set members as they become available is called an *online* method. Online methods might be used, for example, when the

next training instance is some function of the current hypothesis and the previous instance—as it would be when a classifier is used to decide on a robot's next action given its current set of sensory inputs. The next set of sensory inputs will depend on which action was selected.

1.2.5 Noise

Sometimes the vectors in the training set are corrupted by noise. There are two kinds of noise. *Class noise* randomly alters the value of the function; *attribute noise* randomly alters the values of the components of the input vector. In either case, it would be inappropriate to insist that the hypothesized function agree precisely with the values of the samples in the training set.

1.2.6 Performance Evaluation

Even though there is no correct answer in inductive learning, it is important to have methods to evaluate the result of learning. We will discuss this matter in more detail later, but, briefly, in supervised learning the induced function is usually evaluated on a separate set of inputs and function values for them called the *testing set* . A hypothesized function is said to *generalize* when it guesses well on the testing set. Both mean-squared-error and the total number of errors are common measures.

1.3 Learning Requires Bias

Long before now the reader has undoubtedly asked why is learning a function possible at all? Certainly, for example, there are an uncountable number of different functions having values that agree with the four samples shown in Fig. 1.3. Why would a learning procedure happen to select the quadratic one shown in that figure? In order to make that selection we had at least to limit *a priori* the set of hypotheses to quadratic functions and then to insist that the one we chose passed through all four sample points. This kind of *a priori* information is called *bias*, and useful learning without bias is impossible.

We can gain more insight into the role of bias by considering the special case of learning a Boolean function of n dimensions. There are 2^n different Boolean inputs possible. Suppose we had no bias; that is $\mathcal{H}$ is the set of *all* 2^{2^n} Boolean functions, and we have no preference among those that fit the samples in the training set. In this case, after being presented with one member of the training set and its value we can rule out precisely one-half of the members of $\mathcal{H}$—those Boolean functions that would misclassify this labeled sample. The remaining functions constitute what is called a "version space;" we'll explore that concept in more detail later. As we present more members of the training set, the graph of the number of hypotheses not yet ruled out as a function of the number of different patterns presented is as shown in Fig. 1.4. At any stage of the process,

half of the remaining Boolean functions have value 1 and half have value 0 for *any* training pattern not yet seen. No generalization is possible in this case because the training patterns give no clue about the value of a pattern not yet seen. Only memorization is possible here, which is a trivial sort of learning.

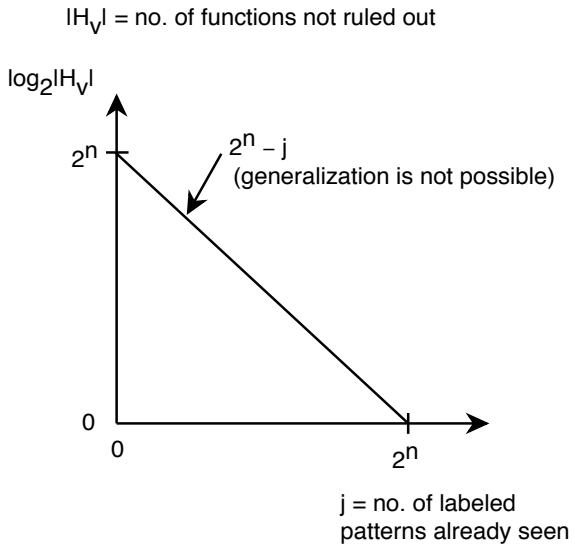

Figure 1.4: Hypotheses Remaining as a Function of Labeled Patterns Presented

But suppose we limited $\mathcal{H}$ to some subset, $\mathcal{H}_c$, of all Boolean functions. Depending on the subset and on the order of presentation of training patterns, a curve of hypotheses not yet ruled out might look something like the one shown in Fig. 1.5. In this case it is even possible that after seeing fewer than all 2^n labeled samples, there might be only one hypothesis that agrees with the training set. Certainly, even if there is more than one hypothesis remaining, *most* of them may have the same value for *most* of the patterns not yet seen! The theory of *Probably Approximately Correct (PAC)* learning makes this intuitive idea precise. We'll examine that theory later.

Let's look at a specific example of how bias aids learning. A Boolean function can be represented by a hypercube each of whose vertices represents a different input pattern. We show a 3-dimensional version in Fig. 1.6. There, we show a training set of six sample patterns and have marked those having a value of 1 by a small square and those having a value of 0 by a small circle. If the hypothesis set consists of just the *linearly separable* functions—those for which the positive and negative instances can be separated by a linear surface, then there is only one function remaining in this hypothsis set that is consistent with the training set. So, in this case, even though the training set does not contain all possible patterns, we can already pin down what the function must be—given the bias.

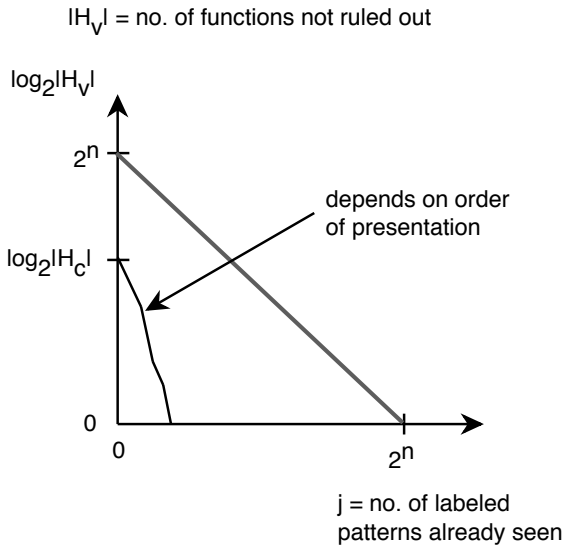

Figure 1.5: Hypotheses Remaining From a Restricted Subset

Machine learning researchers have identified two main varieties of bias, absolute and preference. In *absolute bias* (also called *restricted hypothesis-space bias*), one restricts $\mathcal{H}$ to a definite subset of functions. In our example of Fig. 1.6, the restriction was to linearly separable Boolean functions. In *preference bias*, one selects that hypothesis that is minimal according to some ordering scheme over all hypotheses. For example, if we had some way of measuring the *complexity* of a hypothesis, we might select the one that was simplest among those that performed satisfactorily on the training set. The principle of *Occam's razor*, used in science to prefer simple explanations to more complex ones, is a type of preference bias. (William of Occam, 1285-?1349, was an English philosopher who said: "*non sunt multiplicanda entia praeter necessitatem*," which means "entities should not be multiplied unnecessarily.")

1.4 Sample Applications

Our main emphasis in this book is on the concepts of machine learning—not on its applications. Nevertheless, if these concepts were irrelevant to real-world problems they would probably not be of much interest. As motivation, we give a short summary of some areas in which machine learning techniques have been successfully applied. [Langley, 1992] cites some of the following applications and others:

a. Rule discovery using a variant of ID3 for a printing industry problem

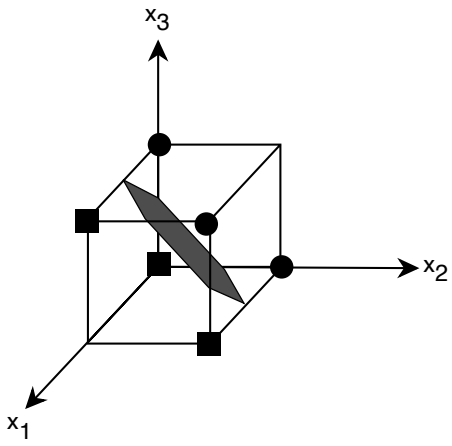

Figure 1.6: A Training Set That Completely Determines a Linearly Separable Function

[Evans & Fisher, 1992].

b. Electric power load forecasting using a k-nearest-neighbor rule system [Jabbour, K., *et al.*, 1987].

c. Automatic "help desk" assistant using a nearest-neighbor system [Acorn & Walden, 1992].

d. Planning and scheduling for a steel mill using ExpertEase, a marketed (ID3-like) system [Michie, 1992].

e. Classification of stars and galaxies [Fayyad, *et al.*, 1993].

Many application-oriented papers are presented at the annual conferences on Neural Information Processing Systems. Among these are papers on: speech recognition, dolphin echo recognition, image processing, bio-engineering, diagnosis, commodity trading, face recognition, music composition, optical character recognition, and various control applications [Various Editors, 1989-1994].

As additional examples, [Hammerstrom, 1993] mentions:

a. Sharp's Japanese kanji character recognition system processes 200 characters per second with 99+% accuracy. It recognizes 3000+ characters.

b. NeuroForecasting Centre's (London Business School and University College London) trading strategy selection network earned an average annual profit of 18% against a conventional system's 12.3%.

c. Fujitsu's (plus a partner's) neural network for monitoring a continuous steel casting operation has been in successful operation since early 1990.

In summary, it is rather easy nowadays to find applications of machine learning techniques. This fact should come as no surprise inasmuch as many machine learning techniques can be viewed as extensions of well known statistical methods which have been successfully applied for many years.

1.5 Sources

Besides the rich literature in machine learning (a small part of which is referenced in the Bibliography), there are several textbooks that are worth mentioning [Hertz, Krogh, & Palmer, 1991, Weiss & Kulikowski, 1991, Natarjan, 1991, Fu, 1994, Langley, 1996]. [Shavlik & Dietterich, 1990, Buchanan & Wilkins, 1993] are edited volumes containing some of the most important papers. A survey paper by [Dietterich, 1990] gives a good overview of many important topics. There are also well established conferences and publications where papers are given and appear including:

- The Annual Conferences on Advances in Neural Information Processing Systems
- The Annual Workshops on Computational Learning Theory
- The Annual International Workshops on Machine Learning
- The Annual International Conferences on Genetic Algorithms

  (The Proceedings of the above-listed four conferences are published by Morgan Kaufmann.)
- The journal *Machine Learning* (published by Kluwer Academic Publishers).

There is also much information, as well as programs and datasets, available over the Internet through the World Wide Web.

1.6 Bibliographical and Historical Remarks

To be added. Every chapter will contain a brief survey of the history of the material covered in that chapter.

Chapter 2

Boolean Functions

2.1 Representation

2.1.1 Boolean Algebra

Many important ideas about learning of functions are most easily presented using the special case of Boolean functions. There are several important subclasses of Boolean functions that are used as hypothesis classes for function learning. Therefore, we digress in this chapter to present a review of Boolean functions and their properties. (For a more thorough treatment see, for example, [Unger, 1989].)

A Boolean function, $f(x_1, x_2, \ldots, x_n)$ maps an n-tuple of (0,1) values to $\{0, 1\}$. *Boolean algebra* is a convenient notation for representing Boolean functions. Boolean algebra uses the connectives $\cdot$, $+$, and $\bar{}$. For example, the *and* function of two variables is written $x_1 \cdot x_2$. By convention, the connective, "$\cdot$" is usually suppressed, and the *and* function is written x_1x_2. x_1x_2 has value 1 if and only if *both* x_1 and x_2 have value 1; if either x_1 or x_2 has value 0, x_1x_2 has value 0. The (inclusive) *or* function of two variables is written $x_1 + x_2$. $x_1 + x_2$ has value 1 if and only if either or both of x_1 or x_2 has value 1; if both x_1 and x_2 have value 0, $x_1 + x_2$ has value 0. The *complement* or *negation* of a variable, x, is written $\overline{x}$. $\overline{x}$ has value 1 if and only if x has value 0; if x has value 1, $\overline{x}$ has value 0.

These definitions are compactly given by the following rules for Boolean algebra:

$1 + 1 = 1$, $1 + 0 = 1$, $0 + 0 = 0$,

$1 \cdot 1 = 1$, $1 \cdot 0 = 0$, $0 \cdot 0 = 0$, and

$\overline{1} = 0$, $\overline{0} = 1$.

Sometimes the arguments and values of Boolean functions are expressed in terms of the constants T (*True*) and F (*False*) instead of 1 and 0, respectively.

The connectives $\cdot$ and $+$ are each commutative and associative. Thus, for example, $x_1(x_2x_3) = (x_1x_2)x_3$, and both can be written simply as $x_1x_2x_3$. Similarly for $+$.

A Boolean formula consisting of a single variable, such as x_1 is called an *atom*. One consisting of either a single variable or its complement, such as $\overline{x_1}$, is called a *literal*.

The operators $\cdot$ and $+$ do not commute between themselves. Instead, we have DeMorgan's laws (which can be verified by using the above definitions):

$\overline{x_1x_2} = \overline{x_1} + \overline{x_2}$, and

$\overline{x_1 + x_2} = \overline{x_1}\ \overline{x_2}$.

2.1.2 Diagrammatic Representations

We saw in the last chapter that a Boolean function could be represented by labeling the vertices of a cube. For a function of n variables, we would need an n-dimensional *hypercube*. In Fig. 2.1 we show some 2- and 3-dimensional examples. Vertices having value 1 are labeled with a small square, and vertices having value 0 are labeled with a small circle.

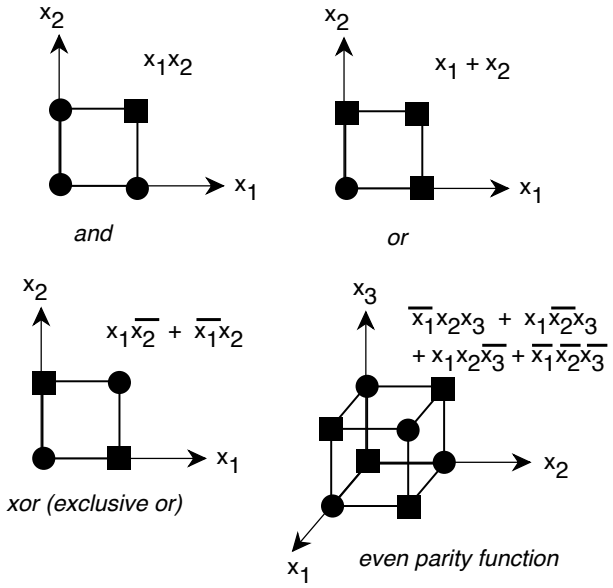

Figure 2.1: Representing Boolean Functions on Cubes

Using the hypercube representations, it is easy to see how many Boolean functions of n dimensions there are. A 3-dimensional cube has $2^3 = 8$ vertices, and each may be labeled in two different ways; thus there are $2^{(2^3)} = 256$

different Boolean functions of 3 variables. In general, there are 2^{2^n} Boolean functions of n variables.

We will be using 2- and 3-dimensional cubes later to provide some intuition about the properties of certain Boolean functions. Of course, we cannot visualize hypercubes (for $n > 3$), and there are many surprising properties of higher dimensional spaces, so we must be careful in using intuitions gained in low dimensions. One diagrammatic technique for dimensions slightly higher than 3 is the *Karnaugh map*. A Karnaugh map is an array of values of a Boolean function in which the horizontal rows are indexed by the values of some of the variables and the vertical columns are indexed by the rest. The rows and columns are arranged in such a way that entries that are adjacent in the map correspond to vertices that are adjacent in the hypercube representation. We show an example of the 4-dimensional even parity function in Fig. 2.2. (An *even parity function* is a Boolean function that has value 1 if there are an even number of its arguments that have value 1; otherwise it has value 0.) Note that all adjacent cells in the table correspond to inputs differing in only one component.

Also describe *general logic diagrams*, [Wnek, *et al.*, 1990].

x_1,x_2 \ x_3,x_4	00	01	11	10
00	1	0	1	0
01	0	1	0	1
11	1	0	1	0
10	0	1	0	1

Figure 2.2: A Karnaugh Map

2.2 Classes of Boolean Functions

2.2.1 Terms and Clauses

To use absolute bias in machine learning, we limit the class of hypotheses. In learning Boolean functions, we frequently use some of the common sub-classes of those functions. Therefore, it will be important to know about these subclasses.

One basic subclass is called *terms*. A term is any function written in the form $l_1 l_2 \cdots l_k$, where the l_i are literals. Such a form is called a *conjunction* of literals. Some example terms are $x_1 x_7$ and $x_1 x_2 \overline{x_4}$. The *size* of a term is the number of literals it contains. The examples are of sizes 2 and 3, respectively. (Strictly speaking, the *class* of conjunctions of literals is called the *monomials*,

and a conjunction of literals itself is called a *term*. This distinction is a fine one which we elect to blur here.)

It is easy to show that there are exactly 3^n possible terms of n variables. The number of terms of size k or less is bounded from above by $\sum_{i=0}^{k} C(2n, i) = O(n^k)$, where $C(i, j) = \frac{i!}{(i-j)!j!}$ is the binomial coefficient.

Probably I'll put in a simple term-learning algorithm here—so we can get started on learning! Also for DNF functions and decision lists—as they are defined in the next few pages.

A *clause* is any function written in the form $l_1 + l_2 + \cdots + l_k$, where the l_i are literals. Such a form is called a *disjunction* of literals. Some example clauses are $x_3 + x_5 + x_6$ and $x_1 + \overline{x_4}$. The *size* of a clause is the number of literals it contains. There are 3^n possible clauses and fewer than $\sum_{i=0}^{k} C(2n, i)$ clauses of size k or less. If f is a term, then (by De Morgan's laws) $\overline{f}$ is a clause, and vice versa. Thus, terms and clauses are duals of each other.

In psychological experiments, conjunctions of literals seem easier for humans to learn than disjunctions of literals.

2.2.2 DNF Functions

A Boolean function is said to be in *disjunctive normal form (DNF)* if it can be written as a *disjunction* of terms. Some examples in DNF are: $f = x_1x_2 + x_2x_3x_4$ and $f = x_1\overline{x_3} + \overline{x_2}\ \overline{x_3} + x_1x_2\overline{x_3}$. A DNF expression is called a k-term DNF expression if it is a disjunction of k terms; it is in the class k-DNF if the size of its largest term is k. The examples above are 2-term and 3-term expressions, respectively. Both expressions are in the class 3-DNF.

Each term in a DNF expression for a function is called an *implicant* because it "implies" the function (if the term has value 1, so does the function). In general, a term, t, is an implicant of a function, f, if f has value 1 whenever t does. A term, t, is a *prime implicant* of f if the term, t', formed by taking any literal out of an implicant t is no longer an implicant of f. (The implicant cannot be "divided" by any term and remain an implicant.)

Thus, both $x_2\overline{x_3}$ and $\overline{x_1}\ \overline{x_3}$ are prime implicants of $f = x_2\overline{x_3} + \overline{x_1}\ \overline{x_3} + x_2x_1\overline{x_3}$, but $x_2x_1\overline{x_3}$ is not.

The relationship between implicants and prime implicants can be geometrically illustrated using the cube representation for Boolean functions. Consider, for example, the function $f = x_2\overline{x_3} + \overline{x_1}\ \overline{x_3} + x_2x_1\overline{x_3}$. We illustrate it in Fig. 2.3. Note that each of the three planes in the figure "cuts off" a group of vertices having value 1, but none cuts off any vertices having value 0. These planes are pictorial devices used to isolate certain lower dimensional *subfaces* of the cube. Two of them isolate one-dimensional *edges*, and the third isolates a zero-dimensional *vertex*. Each group of vertices on a subface corresponds to one of the implicants of the function, f, and thus each implicant corresponds to a subface of some dimension. A k-dimensional subface corresponds to an $(n-k)$-size implicant term. The function is written as the disjunction of the implicants—corresponding to the union of all the vertices cut off by all of the planes. Geometrically, an implicant is prime if and only if its corresponding subface is the largest dimensional subface that includes all of its vertices and

no other vertices having value 0. Note that the term $x_2x_1\overline{x_3}$ is not a prime implicant of f. (In this case, we don't even have to include this term in the function because the vertex cut off by the plane corresponding to $x_2x_1\overline{x_3}$ is already cut off by the plane corresponding to $x_2\overline{x_3}$.) The other two implicants are prime because their corresponding subfaces cannot be expanded without including vertices having value 0.

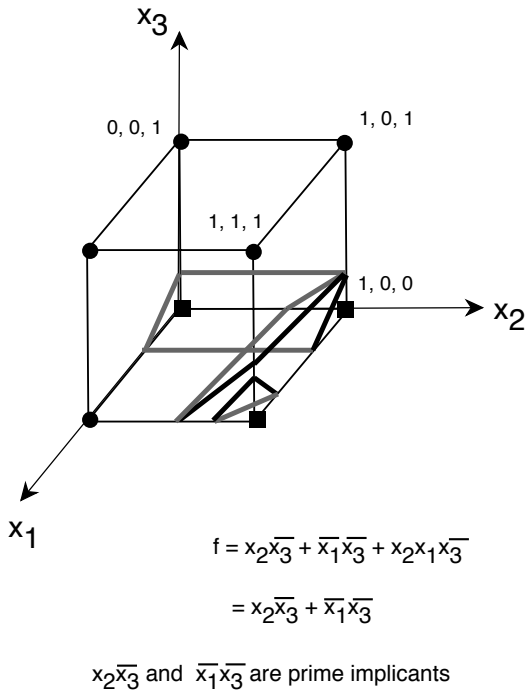

Figure 2.3: A Function and its Implicants

Note that all Boolean functions can be represented in DNF—trivially by disjunctions of terms of size n where each term corresponds to one of the vertices whose value is 1. Whereas there are 2^{2^n} functions of n dimensions in DNF (since any Boolean function can be written in DNF), there are just $2^{O(n^k)}$ functions in k-DNF.

All Boolean functions can also be represented in DNF in which each term is a prime implicant, but that representation is not unique, as shown in Fig. 2.4.

If we can express a function in DNF form, we can use the *consensus* method to find an expression for the function in which each term is a prime implicant. The consensus method relies on two results:

We may replace this section with one describing the Quine-McCluskey method instead.

- Consensus:

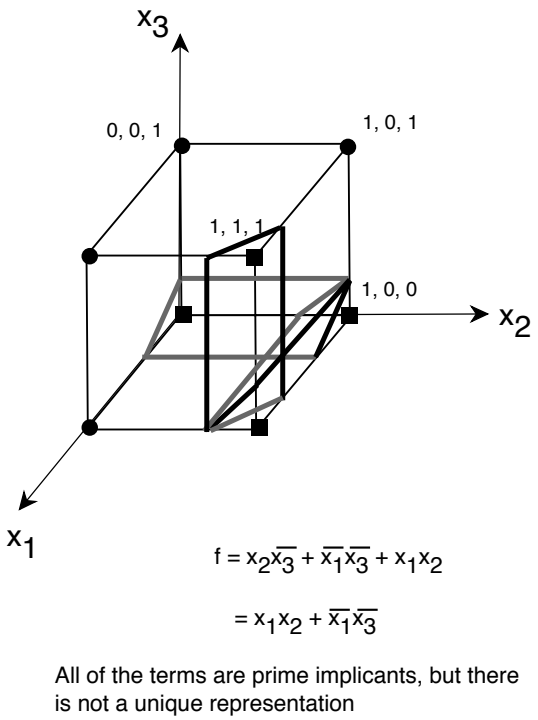

Figure 2.4: Non-Uniqueness of Representation by Prime Implicants

$$x_i \cdot f_1 + \overline{x_i} \cdot f_2 = x_i \cdot f_1 + \overline{x_i} \cdot f_2 + f_1 \cdot f_2$$

where f_1 and f_2 are terms such that no literal appearing in f_1 appears complemented in f_2. $f_1 \cdot f_2$ is called the *consensus* of $x_i \cdot f_1$ and $\overline{x_i} \cdot f_2$. Readers familiar with the *resolution* rule of inference will note that consensus is the dual of resolution.

Examples: x_1 is the consensus of x_1x_2 and $x_1\overline{x_2}$. The terms $\overline{x_1}x_2$ and $x_1\overline{x_2}$ have no consensus since each term has more than one literal appearing complemented in the other.

- Subsumption:

$$x_i \cdot f_1 + f_1 = f_1$$

where f_1 is a term. We say that f_1 *subsumes* $x_i \cdot f_1$.

Example: $\overline{x_1}\ \overline{x_4}x_5$ subsumes $\overline{x_1}\ \overline{x_4}\ \overline{x_2}x_5$

The consensus method for finding a set of prime implicants for a function, f, iterates the following operations on the terms of a DNF expression for f until no more such operations can be applied:

a. initialize the process with the set, $\mathcal{T}$, of terms in the DNF expression of f,

b. compute the consensus of a pair of terms in $\mathcal{T}$ and add the result to $\mathcal{T}$,

c. eliminate any terms in $\mathcal{T}$ that are subsumed by other terms in $\mathcal{T}$.

When this process halts, the terms remaining in $\mathcal{T}$ are all prime implicants of f.

Example: Let $f = \overline{x_1}x_2 + \overline{x_1}\ \overline{x_2}x_3 + \overline{x_1}\ \overline{x_2}\ \overline{x_3}\ \overline{x_4}x_5$. We show a derivation of a set of prime implicants in the *consensus tree* of Fig. 2.5. The circled numbers adjoining the terms indicate the order in which the consensus and subsumption operations were performed. Shaded boxes surrounding a term indicate that it was subsumed. The final form of the function in which all terms are prime implicants is: $f = \overline{x_1}x_2 + \overline{x_1}x_3 + \overline{x_1}\ \overline{x_4}x_5$. Its terms are all of the non-subsumed terms in the consensus tree.

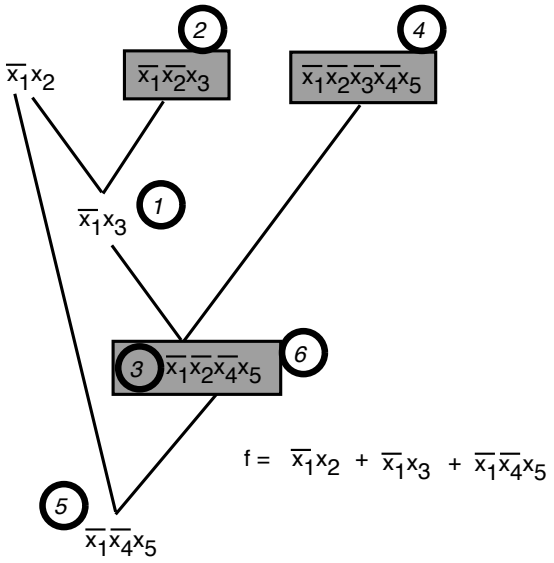

Figure 2.5: A Consensus Tree

2.2.3 CNF Functions

Disjunctive normal form has a dual: *conjunctive normal form (CNF)*. A Boolean function is said to be in CNF if it can be written as a *conjunction* of clauses.

An example in CNF is: $f = (x_1 + x_2)(x_2 + x_3 + x_4)$. A CNF expression is called a k-clause CNF expression if it is a conjunction of k clauses; it is in the class k-CNF if the size of its largest clause is k. The example is a 2-clause expression in 3-CNF. If f is written in DNF, an application of De Morgan's law renders $\overline{f}$ in CNF, and vice versa. Because CNF and DNF are duals, there are also $2^{O(n^k)}$ functions in k-CNF.

2.2.4 Decision Lists

Rivest has proposed a class of Boolean functions called *decision lists* [Rivest, 1987]. A decision list is written as an ordered list of pairs:

(t_q, v_q)

(t_{q-1}, v_{q-1})

$\ldots$

(t_i, v_i)

$\ldots$

(t_2, v_2)

(T, v_1)

where the v_i are either 0 or 1, the t_i are terms in $(x_1, \ldots, x_n)$, and T is a term whose value is 1 (regardless of the values of the x_i). The value of a decision list is the value of v_i for the first t_i in the list that has value 1. (At least one t_i will have value 1, because the last one does; v_1 can be regarded as a *default* value of the decision list.) The decision list is of *size* k, if the size of the largest term in it is k. The class of decision lists of size k or less is called k-DL.

An example decision list is:

$f =$

$(\overline{x_1}x_2, 1)$

$(\overline{x_1}\ \overline{x_2}x_3, 0)$

$\overline{x_2}x_3, 1)$

$(1, 0)$

f has value 0 for $x_1 = 0$, $x_2 = 0$, and $x_3 = 1$. It has value 1 for $x_1 = 1$, $x_2 = 0$, and $x_3 = 1$. This function is in 3-DL.

It has been shown that the class k-DL is a strict superset of the union of k-DNF and k-CNF. There are $2^{O[n^k k \log(n)]}$ functions in k-DL [Rivest, 1987].

Interesting generalizations of decision lists use other Boolean functions in place of the terms, t_i. For example we might use linearly separable functions in place of the t_i (see below and [Marchand & Golea, 1993]).

2.2.5 Symmetric and Voting Functions

A Boolean function is called *symmetric* if it is invariant under permutations of the input variables. For example, any function that is dependent only on the number of input variables whose values are 1 is a symmetric function. The *parity* functions, which have value 1 depending on whether or not the number of input variables with value 1 is even or odd is a symmetric function. (The *exclusive or* function, illustrated in Fig. 2.1, is an odd-parity function of two dimensions. The *or* and *and* functions of two dimensions are also symmetric.)

An important subclass of the symmetric functions is the class of *voting functions* (also called m-of-n functions). A k-voting function has value 1 if and only if k or more of its n inputs has value 1. If $k = 1$, a voting function is the same as an n-sized clause; if $k = n$, a voting function is the same as an n-sized term; if $k = (n + 1)/2$ for n odd or $k = 1 + n/2$ for n even, we have the *majority* function.

2.2.6 Linearly Separable Functions

The linearly separable functions are those that can be expressed as follows:

$$f = \text{thresh}(\sum_{i=1}^{n} w_i x_i, \theta)$$

where w_i, $i = 1, \ldots, n$, are real-valued numbers called *weights*, θ is a real-valued number called the *threshold*, and $\text{thresh}(\sigma, \theta)$ is 1 if $\sigma \geq \theta$ and 0 otherwise. (Note that the concept of linearly separable functions can be extended to non-Boolean inputs.) The k-voting functions are all members of the class of linearly separable functions in which the weights all have unit value and the threshold depends on k. Thus, terms and clauses are special cases of linearly separable functions.

A convenient way to write linearly separable functions uses vector notation:

$$f = \text{thresh}(\mathbf{X} \cdot \mathbf{W}, \theta)$$

where $\mathbf{X} = (x_1, \ldots, x_n)$ is an n-dimensional vector of input variables, $\mathbf{W} = (w_1, \ldots, w_n)$ is an n-dimensional vector of weight values, and $\mathbf{X} \cdot \mathbf{W}$ is the *dot* (or *inner*) product of the two vectors. Input vectors for which f has value 1 lie in a half-space on one side of (and on) a hyperplane whose orientation is normal to $\mathbf{W}$ and whose position (with respect to the origin) is determined by θ. We saw an example of such a separating plane in Fig. 1.6. With this idea in mind, it is easy to see that two of the functions in Fig. 2.1 are linearly separable, while two are not. Also note that the terms in Figs. 2.3 and 2.4 are linearly separable functions as evidenced by the separating planes shown.

There is no closed-form expression for the number of linearly separable functions of n dimensions, but the following table gives the numbers for n up to 6.

n	Boolean Functions	Linearly Separable Functions
1	4	4
2	16	14
3	256	104
4	65,536	1,882
5	$\approx 4.3 \times 10^9$	94,572
6	$\approx 1.8 \times 10^{19}$	15,028,134

[Muroga, 1971] has shown that (for $n > 1$) there are no more than 2^{n^2} linearly separable functions of n dimensions. (See also [Winder, 1961, Winder, 1962].)

2.3 Summary

The diagram in Fig. 2.6 shows some of the set inclusions of the classes of Boolean functions that we have considered. We will be confronting these classes again in later chapters.

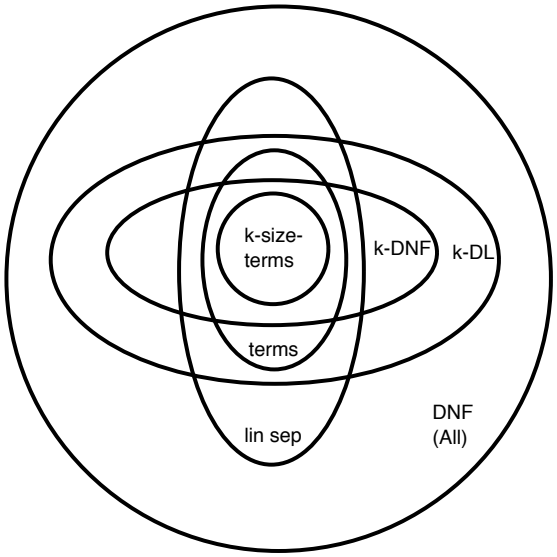

Figure 2.6: Classes of Boolean Functions

The sizes of the various classes are given in the following table (adapted from [Dietterich, 1990, page 262]):

Class	Size of Class
terms	3^n
clauses	3^n
k-term DNF	$2^{O(kn)}$
k-clause CNF	$2^{O(kn)}$
k-DNF	$2^{O(n^k)}$
k-CNF	$2^{O(n^k)}$
k-DL	$2^{O[n^k k \log(n)]}$
lin sep	$2^{O(n^2)}$
DNF	2^{2^n}

2.4 Bibliographical and Historical Remarks

To be added.

Chapter 3

Using Version Spaces for Learning

3.1 Version Spaces and Mistake Bounds

The first learning methods we present are based on the concepts of *version spaces* and *version graphs*. These ideas are most clearly explained for the case of Boolean function learning. Given an initial hypothesis set $\mathcal{H}$ (a subset of all Boolean functions) and the values of $f(\mathbf{X})$ for each $\mathbf{X}$ in a training set, Ξ, the version space is that subset of hypotheses, $\mathcal{H}_v$, that is consistent with these values. A hypothesis, h, is *consistent* with the values of $\mathbf{X}$ in Ξ if and only if $h(\mathbf{X}) = f(\mathbf{X})$ for all $\mathbf{X}$ in Ξ. We say that the hypotheses in $\mathcal{H}$ that are not consistent with the values in the training set are *ruled out* by the training set.

We could imagine (conceptually only!) that we have devices for implementing every function in $\mathcal{H}$. An incremental training procedure could then be defined which presented each pattern in Ξ to each of these functions and then eliminated those functions whose values for that pattern did not agree with its given value. At any stage of the process we would then have left some subset of functions that are consistent with the patterns presented so far; this subset is the version space for the patterns already presented. This idea is illustrated in Fig. 3.1.

Consider the following procedure for classifying an arbitrary input pattern, $\mathbf{X}$: the pattern is put in the same class (0 or 1) as are the majority of the outputs of the functions in the version space. During the learning procedure, if this majority is not equal to the value of the pattern presented, we say a *mistake* is made, and we revise the version space accordingly—eliminating all those (majority of the) functions voting incorrectly. Thus, whenever a mistake is made, we rule out at least half of the functions remaining in the version space.

How many mistakes can such a procedure make? Obviously, we can make no more than $\log_2(|\mathcal{H}|)$ mistakes, where $|\mathcal{H}|$ is the number of hypotheses in the

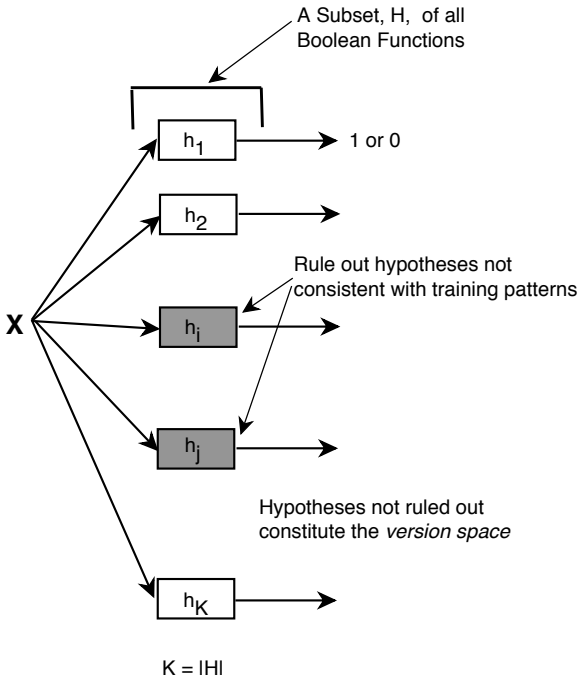

Figure 3.1: Implementing the Version Space

original hypothesis set, $\mathcal{H}$. (Note, though, that the number of training patterns seen before this maximum number of mistakes is made might be much greater.) This theoretical (and very impractical!) result (due to [Littlestone, 1988]) is an example of a *mistake bound*—an important concept in machine learning theory. It shows that there must exist a learning procedure that makes no more mistakes than this upper bound. Later, we'll derive other mistake bounds.

As a special case, if our bias was to limit $\mathcal{H}$ to terms, we would make no more than $\log_2(3^n) = n \log_2(3) = 1.585n$ mistakes before *exhausting* the version space. This result means that if f were a term, we would make no more than $1.585n$ mistakes before learning f, and otherwise we would make no more than that number of mistakes before being able to decide that f is not a term.

Even if we do not have sufficient training patterns to reduce the version space to a single function, it may be that there are enough training patterns to reduce the version space to a set of functions such that most of them assign the same values to most of the patterns we will see henceforth. We could select one of the remaining functions at random and be reasonably assured that it will generalize satisfactorily. We next discuss a computationally more feasible method for representing the version space.

3.2 Version Graphs

Boolean functions can be ordered by *generality*. A Boolean function, f_1, is *more general* than a function, f_2, (and f_2 is *more specific* than f_1), if f_1 has value 1 for all of the arguments for which f_2 has value 1, and $f_1 \neq f_2$. For example, x_3 is more general than x_2x_3 but is not more general than $x_3 + x_2$.

We can form a graph with the hypotheses, $\{h_i\}$, in the version space as nodes. A node in the graph, h_i, has an arc directed to node, h_j, if and only if h_j is more general than h_i. We call such a graph a *version graph*. In Fig. 3.2, we show an example of a version graph over a 3-dimensional input space for hypotheses restricted to terms (with none of them yet ruled out).

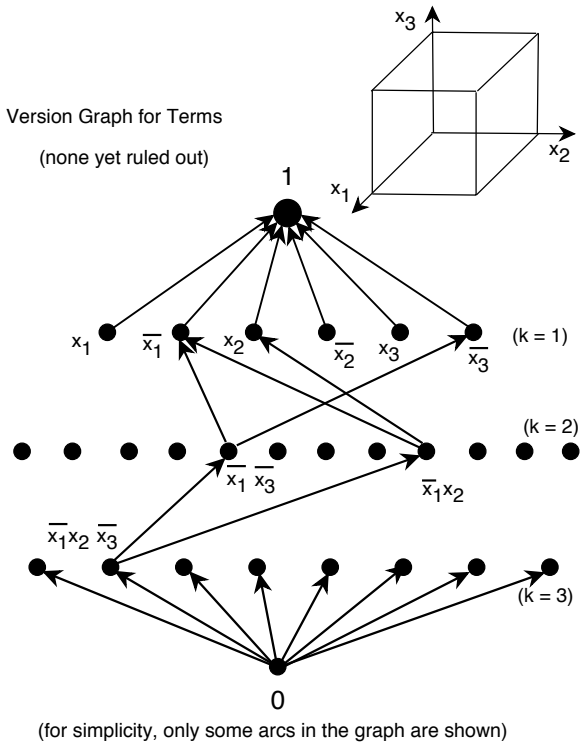

Figure 3.2: A Version Graph for Terms

That function, denoted here by "1," which has value 1 for all inputs, corresponds to the node at the top of the graph. (It is more general than any other term.) Similarly, the function "0" is at the bottom of the graph. Just below "1" is a row of nodes corresponding to all terms having just one literal, and just below them is a row of nodes corresponding to terms having two literals, and

so on. There are $3^3 = 27$ functions altogether (the function "0," included in the graph, is technically not a term). To make our portrayal of the graph less cluttered only some of the arcs are shown; each node in the actual graph has an arc directed to all of the nodes above it that are more general.

We use this same example to show how the version graph changes as we consider a set of labeled samples in a training set, Ξ. Suppose we first consider the training pattern (1, 0, 1) with value 0. Some of the functions in the version graph of Fig. 3.2 are inconsistent with this training pattern. These ruled out nodes are no longer in the version graph and are shown shaded in Fig. 3.3. We also show there the three-dimensional cube representation in which the vertex (1, 0, 1) has value 0.

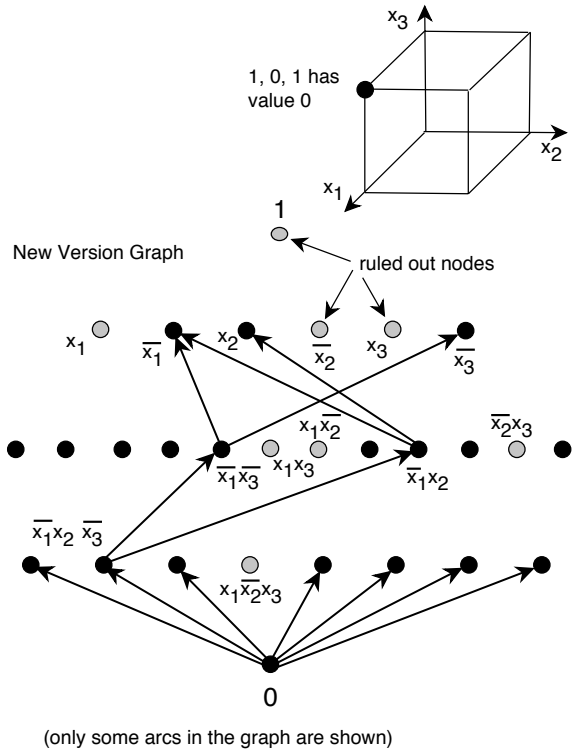

Figure 3.3: The Version Graph Upon Seeing (1, 0, 1)

In a version graph, there are always a set of hypotheses that are maximally general and a set of hypotheses that are maximally specific. These are called the *general boundary set (gbs)* and the *specific boundary set (sbs)*, respectively. In Fig. 3.4, we have the version graph as it exists after learning that (1,0,1) has value 0 and (1, 0, 0) has value 1. The gbs and sbs are shown.

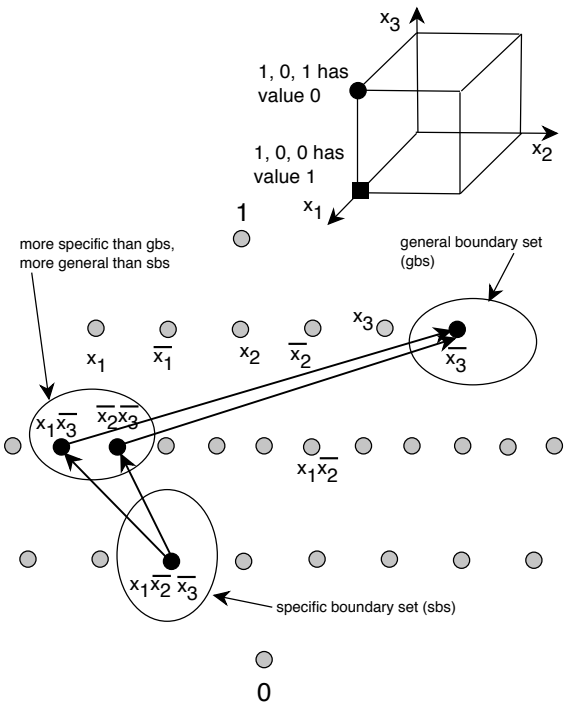

Figure 3.4: The Version Graph Upon Seeing (1, 0, 1) and (1, 0, 0)

Boundary sets are important because they provide an alternative to representing the entire version space explicitly, which would be impractical. Given only the boundary sets, it is possible to determine whether or not any hypothesis (in the prescribed class of Boolean functions we are using) is a member or not of the version space. This determination is possible because of the fact that any member of the version space (that is not a member of one of the boundary sets) is more specific than some member of the general boundary set and is more general than some member of the specific boundary set.

If we limit our Boolean functions that can be in the version space to terms, it is a simple matter to determine maximally general and maximally specific functions (assuming that there is some term that is in the version space). A maximally specific one corresponds to a subface of *minimal dimension* that contains all the members of the training set labelled by a 1 and no members labelled by a 0. A maximally general one corresponds to a subface of *maximal dimension* that contains all the members of the training set labelled by a 1 and no members labelled by a 0. Looking at Fig. 3.4, we see that the subface of minimal dimension that contains (1, 0, 0) but does not contain (1, 0, 1) is just the vertex (1, 0, 0) itself—corresponding to the function $x_1\overline{x_2}\ \overline{x_3}$. The subface

of maximal dimension that contains (1, 0, 0) but does not contain (1, 0, 1) is the bottom face of the cube—corresponding to the function $\overline{x_3}$. In Figs. 3.2 through 3.4 the sbs is always singular. Version spaces for terms always have singular specific boundary sets. As seen in Fig. 3.3, however, the gbs of a version space for terms need not be singular.

3.3 Learning as Search of a Version Space

[To be written. Relate to term learning algorithm presented in Chapter Two. Also discuss best-first search methods. See Pat Langley's example using "pseudo-cells" of how to generate and eliminate hypotheses.]

Selecting a hypothesis from the version space can be thought of as a search problem. One can start with a very general function and specialize it through various specialization operators until one finds a function that is consistent (or adequately so) with a set of training patterns. Such procedures are usually called *top-down* methods. Or, one can start with a very special function and generalize it—resulting in *bottom-up* methods. We shall see instances of both styles of learning in this book.

Compare this view of top-down versus bottom-up with the *divide-and-conquer* and the *covering* (or AQ) methods of decision-tree induction.

3.4 The Candidate Elimination Method

The *candidate elimination method*, is an incremental method for computing the boundary sets. Quoting from [Hirsh, 1994, page 6]:

> "The *candidate-elimination algorithm* manipulates the boundary-set representation of a version space to create boundary sets that represent a new version space consistent with all the previous instances plus the new one. For a positive exmple the algorithm generalizes the elements of the [sbs] as little as possible so that they cover the new instance yet remain consistent with past data, and removes those elements of the [gbs] that do not cover the new instance. For a negative instance the algorithm specializes elements of the [gbs] so that they no longer cover the new instance yet remain consistent with past data, and removes from the [sbs] those elements that mistakenly cover the new, negative instance."

The method uses the following definitions (adapted from [Genesereth & Nilsson, 1987]):

- a hypothesis is called *sufficient* if and only if it has value 1 for all training samples labeled by a 1,
- a hypothesis is called *necessary* if and only if it has value 0 for all training samples labeled by a 0.

Here is how to think about these definitions: A hypothesis implements a *sufficient* condition that a training sample has value 1 if the hypothesis has value 1 for all of the positive instances; a hypothesis implements a *necessary* condition that a training sample has value 1 if the hypothesis has value 0 for all of the negative instances. A hypothesis is consistent with the training set (and thus is in the version space) if and only if it is both sufficient and necessary.

We start (before receiving any members of the training set) with the function "0" as the singleton element of the specific boundary set and with the function "1" as the singleton element of the general boundary set. Upon receiving a new labeled input vector, the boundary sets are changed as follows:

a. If the new vector is labelled with a 1:

   The new general boundary set is obtained from the previous one by excluding any elements in it that are not sufficient. (That is, we exclude any elements that have value 0 for the new vector.)

   The new specific boundary set is obtained from the previous one by replacing each element, h_i, in it by all of its *least generalizations*.

   The hypothesis h_g is a *least generalization* of h if and only if: a) h is more specific than h_g, b) h_g is sufficient, c) no function (including h) that is more specific than h_g is sufficient, and d) h_g is more specific than some member of the new general boundary set. It might be that $h_g = h$. Also, least generalizations of two different functions in the specific boundary set may be identical.

b. If the new vector is labelled with a 0:

   The new specific boundary set is obtained from the previous one by excluding any elements in it that are not necessary. (That is, we exclude any elements that have value 1 for the new vector.)

   The new general boundary set is obtained from the previous one by replacing each element, h_i, in it by all of its *least specializations*.

   The hypothesis h_s is a *least specialization* of h if and only if: a) h is more general than h_s, b) h_s is necessary, c) no function (including h) that is more general than h_s is necessary, and d) h_s is more general than some member of the new specific boundary set. Again, it might be that $h_s = h$, and least specializations of two different functions in the general boundary set may be identical.

As an example, suppose we present the vectors in the following order:

vector	label
(1, 0, 1)	0
(1, 0, 0)	1
(1, 1, 1)	0
(0, 0, 1)	0

We start with general boundary set, "1", and specific boundary set, "0." After seeing the first sample, (1, 0, 1), labeled with a 0, the specific boundary set stays at "0" (it is necessary), and we change the general boundary set to $\{\overline{x_1}, x_2, \overline{x_3}\}$. Each of the functions, $\overline{x_1}$, x_2, and $\overline{x_3}$, are least specializations of "1" (they are necessary, "1" is not, they are more general than "0", and there are no functions that are more general than they and also necessary).

Then, after seeing (1, 0, 0), labeled with a 1, the general boundary set changes to $\{\overline{x_3}\}$ (because $\overline{x_1}$ and x_2 are not sufficient), and the specific boundary set is changed to $\{x_1\overline{x_2}\ \overline{x_3}\}$. This single function is a least generalization of "0" (it is sufficient, "0" is more specific than it, no function (including "0") that is more specific than it is sufficient, and it is more specific than some member of the general boundary set.

When we see (1, 1, 1), labeled with a 0, we do not change the specific boundary set because its function is still necessary. We do not change the general boundary set either because $\overline{x_3}$ is still necessary.

Finally, when we see (0, 0, 1), labeled with a 0, we do not change the specific boundary set because its function is still necessary. We do not change the general boundary set either because $\overline{x_3}$ is still necessary.

Maybe I'll put in an example of a version graph for non-Boolean functions.

3.5 Bibliographical and Historical Remarks

The concept of version spaces and their role in learning was first investigated by Tom Mitchell [Mitchell, 1982]. Although these ideas are not used in practical machine learning procedures, they do provide insight into the nature of hypothesis selection. In order to accomodate noisy data, version spaces have been generalized by [Hirsh, 1994] to allow hypotheses that are not necessarily consistent with the training set.

More to be added.

Chapter 4

Neural Networks

In chapter two we defined several important subsets of Boolean functions. Suppose we decide to use one of these subsets as a hypothesis set for supervised function learning. We next have the question of how best to implement the function as a device that gives the outputs prescribed by the function for arbitrary inputs. In this chapter we describe how networks of non-linear elements can be used to implement various input-output functions and how they can be trained using supervised learning methods.

Networks of non-linear elements, interconnected through adjustable weights, play a prominent role in machine learning. They are called *neural networks* because the non-linear elements have as their inputs a weighted sum of the outputs of other elements—much like networks of biological neurons do. These networks commonly use the threshold element which we encountered in chapter two in our study of linearly separable Boolean functions. We begin our treatment of neural nets by studying this threshold element and how it can be used in the simplest of all networks, namely ones composed of a single threshold element.

4.1 Threshold Logic Units

4.1.1 Definitions and Geometry

Linearly separable (threshold) functions are implemented in a straightforward way by summing the weighted inputs and comparing this sum to a threshold value as shown in Fig. 4.1. This structure we call a *threshold logic unit (TLU)*. Its output is 1 or 0 depending on whether or not the weighted sum of its inputs is greater than or equal to a threshold value, θ. It has also been called an *Adaline* (for adaptive linear element) [Widrow, 1962, Widrow & Lehr, 1990], an LTU (linear threshold unit), a *perceptron*, and a *neuron*. (Although the word "perceptron" is often used nowadays to refer to a single TLU, Rosenblatt originally defined it as a class of *networks* of threshold elements [Rosenblatt, 1958].)

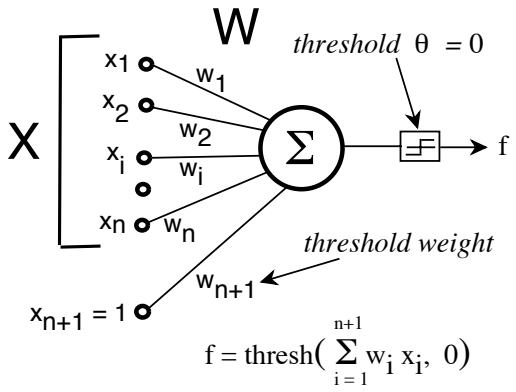

Figure 4.1: A Threshold Logic Unit (TLU)

The n-dimensional feature or input vector is denoted by $\mathbf{X} = (x_1, \ldots, x_n)$. When we want to distinguish among different feature vectors, we will attach subscripts, such as $\mathbf{X}_i$. The components of $\mathbf{X}$ can be any real-valued numbers, but we often specialize to the binary numbers 0 and 1. The weights of a TLU are represented by an n-dimensional *weight vector*, $\mathbf{W} = (w_1, \ldots, w_n)$. Its components are real-valued numbers (but we sometimes specialize to integers). The TLU has output 1 if $\sum_{i=1}^{n} x_i w_i \geq \theta$; otherwise it has output 0. The weighted sum that is calculated by the TLU can be simply represented as a vector dot product, $\mathbf{X}\bullet\mathbf{W}$. (If the pattern and weight vectors are thought of as "column" vectors, this dot product is then sometimes written as $\mathbf{X}^t\mathbf{W}$, where the "row" vector $\mathbf{X}^t$ is the transpose of $\mathbf{X}$.) Often, the threshold, θ, of the TLU is fixed at 0; in that case, arbitrary thresholds are achieved by using $(n+1)$-dimensional "augmented" vectors, $\mathbf{Y}$, and $\mathbf{V}$, whose first n components are the same as those of $\mathbf{X}$ and $\mathbf{W}$, respectively. The $(n+1)$-st component, x_{n+1}, of the augmented feature vector, $\mathbf{Y}$, always has value 1; the $(n+1)$-st component, w_{n+1}, of the augmented weight vector, $\mathbf{V}$, is set equal to the negative of the desired threshold value. (When we want to emphasize the use of augmented vectors, we'll use the $\mathbf{Y}$,$\mathbf{V}$ notation; however when the context of the discussion makes it clear about what sort of vectors we are talking about, we'll lapse back into the more familiar $\mathbf{X}$,$\mathbf{W}$ notation.) In the $\mathbf{Y}$,$\mathbf{V}$ notation, the TLU has an output of 1 if $\mathbf{Y}\bullet\mathbf{V} \geq 0$. Otherwise, the output is 0.

We can give an intuitively useful geometric description of a TLU. A TLU divides the input space by a hyperplane as sketched in Fig. 4.2. The hyperplane is the boundary between patterns for which $\mathbf{X}\bullet\mathbf{W} + w_{n+1} > 0$ and patterns for which $\mathbf{X}\bullet\mathbf{W} + w_{n+1} < 0$. Thus, the equation of the hyperplane itself is $\mathbf{X}\bullet\mathbf{W} + w_{n+1} = 0$. The unit vector that is normal to the hyperplane is $\mathbf{n} = \frac{\mathbf{W}}{|\mathbf{W}|}$, where $|\mathbf{W}| = \sqrt{(w_1^2 + \ldots + w_n^2)}$ is the length of the vector $\mathbf{W}$. (The *normal*

form of the hyperplane equation is $\mathbf{X} \bullet \mathbf{n} + \frac{\mathbf{W}}{|\mathbf{W}|} = 0$.) The distance from the hyperplane to the origin is $\frac{w_{n+1}}{|\mathbf{W}|}$, and the distance from an arbitrary point, $\mathbf{X}$, to the hyperplane is $\frac{\mathbf{X} \bullet \mathbf{W} + w_{n+1}}{|\mathbf{W}|}$. When the distance from the hyperplane to the origin is negative (that is, when $w_{n+1} < 0$), then the origin is on the negative side of the hyperplane (that is, the side for which $\mathbf{X} \bullet \mathbf{W} + w_{n+1} < 0$).

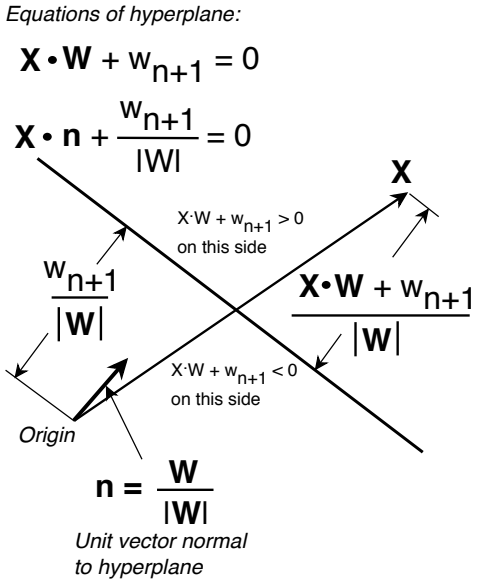

Figure 4.2: TLU Geometry

Adjusting the weight vector, $\mathbf{W}$, changes the orientation of the hyperplane; adjusting w_{n+1} changes the position of the hyperplane (relative to the origin). Thus, training of a TLU can be achieved by adjusting the values of the weights. In this way the hyperplane can be moved so that the TLU implements different (linearly separable) functions of the input.

4.1.2 Special Cases of Linearly Separable Functions

Terms

Any term of size k can be implemented by a TLU with a weight from each of those inputs corresponding to variables occurring in the term. A weight of $+1$ is used from an input corresponding to a positive literal, and a weight of -1 is used from an input corresponding to a negative literal. (Literals not mentioned in the term have weights of zero—that is, no connection at all—from their inputs.) The threshold, θ, is set equal to $k_p - 1/2$, where k_p is the number of positive literals in the term. Such a TLU implements a hyperplane boundary that is

parallel to a subface of dimension $(n - k)$ of the unit hypercube. We show a three-dimensional example in Fig. 4.3. Thus, linearly separable functions are a superset of terms.

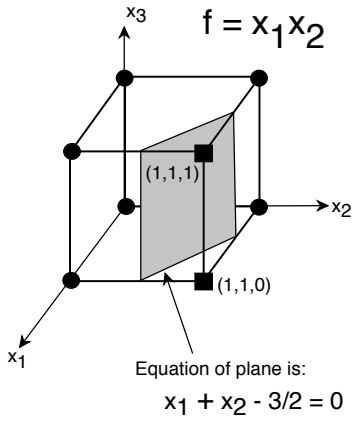

Figure 4.3: Implementing a Term

Clauses

The negation of a clause is a term. For example, the negation of the clause $f = x_1 + x_2 + x_3$ is the term $\overline{f} = \overline{x_1}\ \overline{x_2}\ \overline{x_3}$. A hyperplane can be used to implement this term. If we "invert" the hyperplane, it will implement the clause instead. Inverting a hyperplane is done by multiplying all of the TLU weights—even w_{n+1}—by -1. This process simply changes the orientation of the hyperplane—flipping it around by 180 degrees and thus changing its "positive side." Therefore, linearly separable functions are also a superset of clauses. We show an example in Fig. 4.4.

4.1.3 Error-Correction Training of a TLU

There are several procedures that have been proposed for adjusting the weights of a TLU. We present next a family of *incremental* training procedures with parameter c. These methods make adjustments to the weight vector only when the TLU being trained makes an error on a training pattern; they are called *error-correction* procedures. We use *augmented* feature and weight vectors in describing them.

a. We start with a finite training set, Ξ, of vectors, $\mathbf{Y}_i$, and their binary labels.

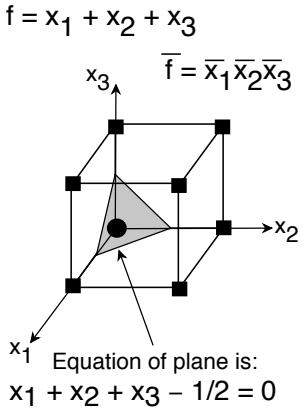

Figure 4.4: Implementing a Clause

b. Compose an infinite training sequence, Σ, of vectors from Ξ and their labels such that each member of Ξ occurs infinitely often in Σ. Set the initial weight values of an TLU to arbitrary values.

c. Repeat forever:

   Present the next vector, $\mathbf{Y}_i$, in Σ to the TLU and note its response.

   (a) If the TLU responds correctly, make no change in the weight vector.

   (b) If $\mathbf{Y}_i$ is supposed to produce an output of 0 and produces an output of 1 instead, modify the weight vector as follows:

$$\mathbf{V} \longleftarrow \mathbf{V} - c_i \mathbf{Y}_i$$

   where c_i is a positive real number called the *learning rate parameter* (whose value is differerent in different instances of this family of procedures and may depend on i).

   Note that after this adjustment the new dot product will be $(\mathbf{V} - c_i\mathbf{Y}_i)\bullet\mathbf{Y}_i = \mathbf{V}\bullet\mathbf{Y}_i - c_i\mathbf{Y}_i\bullet\mathbf{Y}_i$, which is smaller than it was before the weight adjustment.

   (c) If $\mathbf{Y}_i$ is supposed to produce an output of 1 and produces an output of 0 instead, modify the weight vector as follows:

$$\mathbf{V} \longleftarrow \mathbf{V} + c_i \mathbf{Y}_i$$

   In this case, the new dot product will be $(\mathbf{V} + c_i\mathbf{Y}_i)\bullet\mathbf{Y}_i = \mathbf{V}\bullet\mathbf{Y}_i + c_i\mathbf{Y}_i\bullet\mathbf{Y}_i$, which is larger than it was before the weight adjustment.

Note that all three of these cases can be combined in the following rule:

$$\mathbf{V} \longleftarrow \mathbf{V} + c_i(d_i - f_i)\mathbf{Y}_i$$

where d_i is the desired response (1 or 0) for $\mathbf{Y}_i$, and f_i is the actual response (1 or 0) for $\mathbf{Y}_i$.]

Note also that because the weight vector $\mathbf{V}$ now includes the w_{n+1} threshold component, the threshold of the TLU is also changed by these adjustments.

We identify two versions of this procedure:

1) In the *fixed-increment procedure*, the learning rate parameter, c_i, is the same fixed, positive constant for all i. Depending on the value of this constant, the weight adjustment may or may not correct the response to an erroneously classified feature vector.

2) In the *fractional-correction procedure*, the parameter c_i is set to $\lambda \frac{\mathbf{Y_i \bullet V}}{\mathbf{Y_i \bullet Y_i}}$, where $\mathbf{V}$ is the weight vector *before* it is changed. Note that if $\lambda = 0$, no correction takes place at all. If $\lambda = 1$, the correction is just sufficient to make $\mathbf{Y_i \bullet V} = 0$. If $\lambda > 1$, the error will be corrected.

It can be proved that if there is some weight vector, $\mathbf{V}$, that produces a correct output for all of the feature vectors in Ξ, then after a finite number of feature vector presentations, the fixed-increment procedure will find such a weight vector and thus make no more weight changes. The same result holds for the fractional-correction procedure if $1 < \lambda \leq 2$.

For additional background, proofs, and examples of error-correction procedures, see [Nilsson, 1990].

See [Maass & Turán, 1994] for a hyperplane-finding procedure that makes no more than $O(n^2 \log n)$ mistakes.

4.1.4 Weight Space

We can give an intuitive idea about how these procedures work by considering what happens to the augmented weight vector in "weight space" as corrections are made. We use augmented vectors in our discussion here so that the threshold function compares the dot product, $\mathbf{Y}_i \bullet \mathbf{V}$, against a threshold of 0. A particular weight vector, $\mathbf{V}$, then corresponds to a point in $(n + 1)$-dimensional weight space. Now, for any pattern vector, $\mathbf{Y}_i$, consider the locus of all points in weight space corresponding to weight vectors yielding $\mathbf{Y}_i \bullet \mathbf{V} = 0$. This locus is a hyperplane passing through the origin of the $(n+1)$-dimensional space. Each pattern vector will have such a hyperplane corresponding to it. Weight points in one of the half-spaces defined by this hyperplane will cause the corresponding pattern to yield a dot product less than 0, and weight points in the other half-space will cause the corresponding pattern to yield a dot product greater than 0.

We show a schematic representation of such a weight space in Fig. 4.5. There are four pattern hyperplanes, 1, 2, 3, 4 , corresponding to patterns $\mathbf{Y}_1$,

$\mathbf{Y}_2$, $\mathbf{Y}_3$, $\mathbf{Y}_4$, respectively, and we indicate by an arrow the half-space for each in which weight vectors give dot products greater than 0. Suppose we wanted weight values that would give positive responses for patterns $\mathbf{Y}_1$, $\mathbf{Y}_3$, and $\mathbf{Y}_4$, and a negative response for pattern $\mathbf{Y}_2$. The weight point, $\mathbf{V}$, indicated in the figure is one such set of weight values.

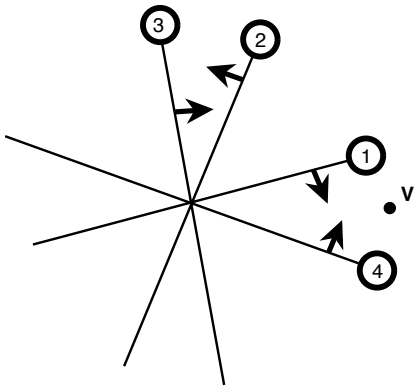

Figure 4.5: Weight Space

The question of whether or not there exists a weight vector that gives desired responses for a given set of patterns can be given a geometric interpretation. To do so involves reversing the "polarity" of those hyperplanes corresponding to patterns for which a negative response is desired. If we do that for our example above, we get the weight space diagram shown in Fig. 4.6.

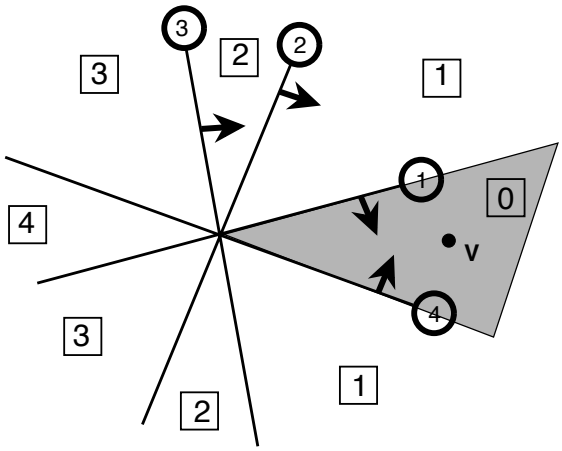

Figure 4.6: Solution Region in Weight Space

If a weight vector exists that correctly classifies a set of patterns, then the half-spaces defined by the correct responses for these patterns will have a non-empty intersection, called the solution region. The solution region will be a "hyper-wedge" region whose vertex is at the origin of weight space and whose cross-section increases with increasing distance from the origin. This region is shown shaded in Fig. 4.6. (The boxed numbers show, for later purposes, the number of errors made by weight vectors in each of the regions.) The fixed-increment error-correction procedure changes a weight vector by moving it normal to any pattern hyperplane for which that weight vector gives an incorrect response. Suppose in our example that we present the patterns in the sequence $\mathbf{Y}_1$, $\mathbf{Y}_2$, $\mathbf{Y}_3$, $\mathbf{Y}_4$, and start the process with a weight point $\mathbf{V}_1$, as shown in Fig. 4.7. Starting at $\mathbf{V}_1$, we see that it gives an incorrect response for pattern $\mathbf{Y}_1$, so we move $\mathbf{V}_1$ to $\mathbf{V}_2$ in a direction normal to plane 1. (That is what adding $\mathbf{Y}_1$ to $\mathbf{V}_1$ does.) $\mathbf{Y}_2$ gives an incorrect response for pattern $\mathbf{Y}_2$, and so on. Ultimately, the responses are only incorrect for planes bounding the solution region. Some of the subsequent corrections may overshoot the solution region, but eventually we work our way out far enough in the solution region that corrections (for a fixed increment size) take us within it. The proofs for convergence of the fixed-increment rule make this intuitive argument precise.

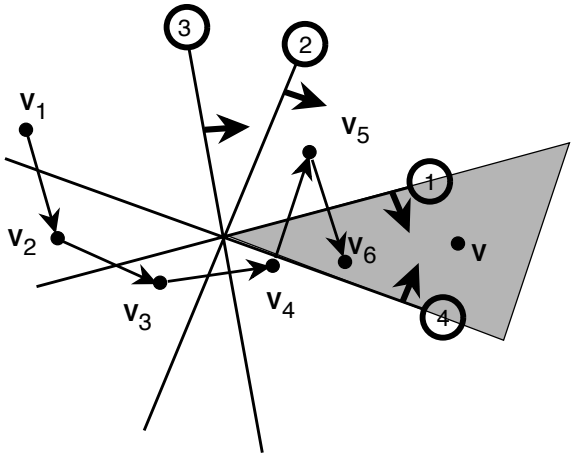

Figure 4.7: Moving Into the Solution Region

4.1.5 The Widrow-Hoff Procedure

The Widrow-Hoff procedure (also called the *LMS* or the *delta* procedure) attempts to find weights that minimize a squared-error function between the pattern labels and the dot product computed by a TLU. For this purpose, the pattern labels are assumed to be either $+1$ or -1 (instead of 1 or 0). The

squared error for a pattern, $\mathbf{X}_i$, with label d_i (for desired output) is:

$$\varepsilon_i = (d_i - \sum_{j=1}^{n+1} x_{ij} w_j)^2$$

where x_{ij} is the j-th component of $\mathbf{X}_i$. The total squared error (over all patterns in a training set, Ξ, containing m patterns) is then:

$$\varepsilon = \sum_{i=1}^{m} (d_i - \sum_{j=1}^{n+1} x_{ij} w_j)^2$$

We want to choose the weights w_j to minimize this squared error. One way to find such a set of weights is to start with an arbitrary weight vector and move it along the negative gradient of ε as a function of the weights. Since ε is quadratic in the w_j, we know that it has a global minimum, and thus this *steepest descent* procedure is guaranteed to find the minimum. Each component of the gradient is the partial derivative of ε with respect to one of the weights. One problem with taking the partial derivative of ε is that ε depends on *all* the input vectors in Ξ. Often, it is preferable to use an incremental procedure in which we try the TLU on just one element, $\mathbf{X}_i$, of Ξ at a time, compute the gradient of the single-pattern squared error, ε_i, make the appropriate adjustment to the weights, and then try another member of Ξ. Of course, the results of the incremental version can only approximate those of the batch one, but the approximation is usually quite effective. We will be describing the incremental version here.

The j-th component of the gradient of the single-pattern error is:

$$\frac{\partial \varepsilon_i}{\partial w_j} = -2(d_i - \sum_{j=1}^{n+1} x_{ij} w_j) x_{ij}$$

An adjustment in the direction of the negative gradient would then change each weight as follows:

$$w_j \longleftarrow w_j + c_i(d_i - f_i)x_{ij}$$

where $f_i = \sum_{j=1}^{n+1} x_{ij} w_j$, and c_i governs the size of the adjustment. The entire weight vector (in augmented, or $\mathbf{V}$, notation) is thus adjusted according to the following rule:

$$\mathbf{V} \longleftarrow \mathbf{V} + c_i(d_i - f_i)\mathbf{Y}_i$$

where, as before, $\mathbf{Y}_i$ is the i-th augmented pattern vector.

The Widrow-Hoff procedure makes adjustments to the weight vector whenever the dot product itself, $\mathbf{Y}_i \bullet \mathbf{V}$, does not equal the specified desired target

value, d_i (which is either 1 or -1). The learning-rate factor, c_i, might decrease with time toward 0 to achieve asymptotic convergence. The Widrow-Hoff formula for changing the weight vector has the same form as the standard fixed-increment error-correction formula. The only difference is that f_i is the thresholded response of the TLU in the error-correction case while it is the dot product itself for the Widrow-Hoff procedure.

Finding weight values that give the desired dot products corresponds to solving a set of linear equalities, and the Widrow-Hoff procedure can be interpreted as a descent procedure that attempts to minimize the mean-squared-error between the actual and desired values of the dot product. (For more on Widrow-Hoff and other related procedures, see [Duda & Hart, 1973, pp. 151ff].)

Examples of training curves for TLU's; performance on training set; performance on test set; cumulative number of corrections.

4.1.6 Training a TLU on Non-Linearly-Separable Training Sets

When the training set is not linearly separable (perhaps because of noise or perhaps inherently), it may still be desired to find a "best" separating hyperplane. Typically, the error-correction procedures will not do well on non-linearly-separable training sets because they will continue to attempt to correct inevitable errors, and the hyperplane will never settle into an acceptable place.

Several methods have been proposed to deal with this case. First, we might use the Widrow-Hoff procedure, which (although it will not converge to zero error on non-linearly separable problems) will give us a weight vector that minimizes the mean-squared-error. A mean-squared-error criterion often gives unsatisfactory results, however, because it prefers many small errors to a few large ones. As an alternative, error correction with a continuous decrease toward zero of the value of the learning rate constant, c, will result in ever decreasing changes to the hyperplane. Duda [Duda, 1966] has suggested keeping track of the average value of the weight vector during error correction and using this average to give a separating hyperplane that performs reasonably well on non-linearly-separable problems. Gallant [Gallant, 1986] proposed what he called the "pocket algorithm." As described in [Hertz, Krogh, & Palmer, 1991, p. 160]:

> . . . the pocket algorithm . . . consists simply in storing (or "putting in your pocket") the set of weights which has had the longest unmodified run of successes so far. The algorithm is stopped after some chosen time t . . .

After stopping, the weights in the pocket are used as a set that should give a small number of errors on the training set. Error-correction proceeds as usual with the ordinary set of weights.

Also see methods proposed by [John, 1995] and by [Marchand & Golea, 1993]. The latter is claimed to outperform the pocket algorithm.

4.2 Linear Machines

The natural generalization of a (two-category) TLU to an R-category classifier is the structure, shown in Fig. 4.8, called a *linear machine*. Here, to use more

familiar notation, the **W**s and **X** are meant to be augmented vectors (with an (n+1)-st component). Such a structure is also sometimes called a "competitive" net or a "winner-take-all" net. The output of the linear machine is one of the numbers, $\{1, \ldots, R\}$, corresponding to which dot product is largest. Note that when $R = 2$, the linear machine reduces to a TLU with weight vector $\mathbf{W} = (\mathbf{W}_1 - \mathbf{W}_2)$.

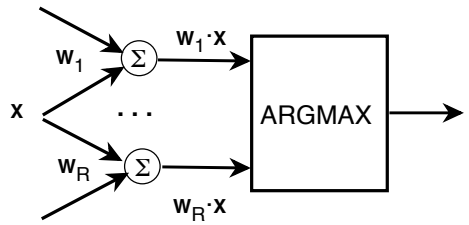

Figure 4.8: A Linear Machine

The diagram in Fig. 4.9 shows the character of the regions in a 2-dimensional space created by a linear machine for $R = 5$. In n dimensions, every pair of regions is either separated by a section of a hyperplane or is non-adjacent.

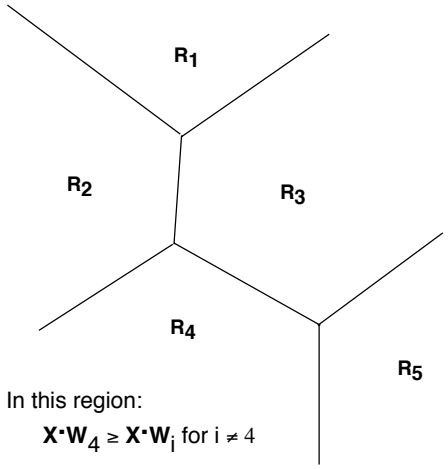

Figure 4.9: Regions For a Linear Machine

To train a linear machine, there is a straightforward generalization of the 2-category error-correction rule. Assemble the patterns in the training set into a sequence as before.

a. If the machine classifies a pattern correctly, no change is made to any of