Machine Learning[1]

Thomas G. Dietterich
Department of Computer Science
Oregon State University
Corvallis, OR 97331-3902

August 29, 1994

[1]To appear in *Annual Review of Computer Science*, Volume 4, Spring 1990

Contents

1 OVERVIEW

The study of machine learning methods has progressed greatly in the past few years. This progress has taken many directions. First, in the area of inductive learning, a new formal definition of learning introduced by Leslie Valiant has provided the foundation for several important theoretical results. Second, a number of new learning algorithms have been developed, and existing algorithms have been improved. Third, the collection of methods that perform so-called explanation-based learning have addressed the problem of speeding up the performance of problem-solving programs. Finally, the philosophical foundations of machine learning have been clarified.

The goal of this article is to present the major results in each of these four directions. We begin with a discussion of the philosophical foundations, since these will provide a framework for the remainder of the article. This is followed by sections that describe *(a)* theoretical results, *(b)* practical inductive learning algorithms, and *(c)* explanation-based learning.

2 PHILOSOPHICAL FOUNDATIONS

How can 'learning' be defined? This question has troubled researchers in the field for many years. While it is not necessary to have a clean definition in order to conduct research, the lack of a workable definition makes it hard to evaluate learning methods to determine whether they succeed.

Recently, two new approaches have been taken to defining 'learning', one introduced by Dietterich (1986) and one by Valiant (1984).

Dietterich (1986) reduces the problem of defining 'learning' to the problem of defining 'knowledge'. Given a satisfactory definition of 'knowledge', 'learning' can be defined as an increase in 'knowledge'. Depending on which definition of 'knowledge' one chooses, one obtains different results.

The definition preferred by Dietterich is the following. An agent (i.e., a person or a program) knows a fact F if the agent has been told F or if the agent can logically infer F from its other knowledge. No limit is placed on the computational resources (e.g., CPU time and memory space) consumed in performing these inferences. This form of knowledge can be called "knowledge in principle" or "deductive closure knowledge". The logical inferences are assumed to preserve correctness (i.e., they are monotonic, deductive inferences).

Given this definition of knowledge, learning (i.e., increases in knowledge) can occur under two circumstances. First, learning occurs when the agent is told a new fact F that it did not know. Second, learning occurs when the agent makes an "inductive leap" and chooses to believe some fact F that is not entailed by its existing knowledge.

For example, suppose an agent knows that a poker hand containing three Jacks is superior to a hand containing only two Queens. Suppose the agent also knows that a poker hand containing three Tens is superior to a hand containing two Eights. Learning occurs when the agent jumps to the conclusion that any hand containing three cards of rank R_1 is superior to any hand containing at most two cards of rank R_2. In short, a system that formulates general rules by analyzing specific examples is one kind of learning system.

Notice that according to this definition, learning does not take place if a system discovers a more efficient way to infer a fact that it already knows in principle. Consequently, simple speed ups (e.g., such as those obtained by caching inferences) do not count as learning. However, it also

Figure 1: The error between the correct fact F and $\hat{F}$.

has the unfortunate consequence that a program that knows the rules of chess would also know the optimal strategy.

Another definition of knowledge is the notion of "explicit belief" defined by Fagin and Halpern (1987). According to this definition, an agent has a combination of implicit beliefs (these correspond to the deductive closure definition of 'knowledge' discussed above) and explicit beliefs (i.e., beliefs the system is 'aware' of). Logical (monotonic) inference can make implicit beliefs explicit. In a particular program, one might define a belief to be explicit if it is stored in a database or if it can be computed within a fixed time limit. In any case, learning takes place, according to this definition, whenever new explicit beliefs are found. Hence, this definition does include simple speed-ups (e.g., those produced by traditional programming language compilers) as forms of learning. It also does not draw a distinction between learning as efficiency improvement and learning as the acquisition of a new rule from examples.

By considering these various definitions of "knowledge" and "learning," we can develop a three-part taxonomy of learning systems: *(a)* systems that receive no inputs and simply become more efficient over time (*speed-up learning*), *(b)* systems that receive new knowledge via inputs but otherwise perform no inductive leaps (*learning by being told*), and *(c)* systems that perform inductive leaps to acquire knowledge that was not previously known either explicitly or implicitly (*inductive learning*).

These definitions provide a basis for evaluating learning systems. Speed-up learning systems should be evaluated by measuring the efficiency improvement that they produce. Systems that learn by being told can be evaluated according to their ability to exploit the information they receive. Finally, inductive learning systems must be evaluated according to the correctness of the knowledge that they produce. This is difficult, however, because inductive learning systems can provide no guarantee of correctness unless they cease to make inductive leaps!

Leslie Valiant's probabilistic framework (Valiant 1984) provides a solution to this last difficulty. Valiant says that a system has learned a fact F if it can guarantee with high probability that F is approximately correct. This definition relaxes the goal of guaranteed correctness in two ways. First, the fact F is permitted to be only approximately correct. Second, with low probability, the learning system may produce an hypothesis F that is totally incorrect. It turns out that this definition provides us with a rigorous criterion for evaluating learning programs.

To understand what it means to be "approximately correct", let us view a fact F as a relation over some universe U of objects. In other words, F is the subset of objects (or tuples) in U that make F true. Intuitively, a second fact $\hat{F}$ is approximately correct if the symmetric difference $F \oplus \hat{F}$ is small (this corresponds to the shaded region in Figure 1). In other words, F and $\hat{F}$ agree over most of the universe U.

Valiant elaborates this definition by taking into consideration the possibility that some elements of U are more important than others. He considers $\hat{F}$ to be approximately correct to the degree that it matches F on the more important elements of U. Specifically, Valiant assumes that the learning system is going to be confronted with a series of "performance trials." In each trial, it will be presented an element $u \in U$ and asked whether $u \in F$ is true. Let P be an unchanging probability distribution over U such that $P(u)$ is the probability that u will be selected in any given trial. Then error$(F, \hat{F})$ is defined to be the probability that the learning system will make a mistake in any given performance trial. Formally,

$$\text{error}(F, \hat{F}) = \sum_{u \in F \oplus \hat{F}} P(u).$$

The fact $\hat{F}$ is approximately correct if error$(F, \hat{F})$ is less than ϵ, where ϵ is a small constant called the *accuracy parameter.*

Now that we understand what it means to be "approximately correct," we must consider the second part of Valiant's definition: The learning system that produces $\hat{F}$ may itself make mistakes from time to time and produce hypotheses that are not approximately correct. In particular, the learning system is usually constructing $\hat{F}$ by analyzing a collection of training examples. A training example is a pair of the form $\langle u, c \rangle$, where $u \in U$ and $c = 1$ if $u \in F$ and $c = 0$ otherwise. If those examples do not provide a representative sample of F, then the learning program may come up with a bad guess, $\hat{F}$.

By making some assumptions about the training sample, we can bound the probability that the learning system will produce an $\hat{F}$ with error greater than ϵ. Specifically, let us assume that the training sample is constructed by independently drawing m examples from U according to the same probability distribution $P(u)$ that will be used during the performance trials. We say that the learning system is *probably approximately correct (PAC)* if

$$Pr\left[\text{error}(F, \hat{F}) > \epsilon\right] \; < \; \delta,$$

where δ is called the *confidence parameter* and where the probability is taken over all training samples of size m.

What Valiant has done is to incorporate a notion of evidential support into the definition of 'learning'. According to Valiant, a program is not considered a learning program if it makes a lucky leap and comes up with a correct fact. Instead, Valiant requires that the learning program consider a large enough set of training examples so that its hypothesis $\hat{F}$ is statistically justified.

This is a major breakthrough because it provides a standard against which to compare inductive learning programs. It also provides a basis for proving results concerning the computational tractability of various learning problems. These results are the topic of the next section.

3 THEORETICAL RESULTS ON LEARNING FROM EXAMPLES

As we have seen above, the goal of learning from examples is to infer, from a set S of training examples, a probably approximately correct fact $\hat{F}$.[1] In principle, this is impossible, because the knowledge of whether $F(u)$ is true for one point in U tells us nothing about the values of F at any other points in U—it merely tells us the value of F at u. When people are confronted with such problems, they circumvent them by imposing some assumptions concerning F. They may assume, for example, that F can be represented as a Boolean conjunction over the features describing U. Or they may prefer the simplest hypothesis $\hat{F}$ consistent with the training examples. This amounts to assuming that F can be represented simply in some given language.

These assumptions concerning F are called the "bias" of the learning system, and they provide it with some means for making a guess concerning the identity of F. There are two general forms of bias: restricted hypothesis space bias and preference bias.

Under the restricted hypothesis space bias, the learning system assumes that the correct concept F is a member of some hypothesis set H, where H contains only some of the $2^{|U|}$ possible concepts over U. This is usually implemented by assuming that F has some restricted syntactic form (e.g., as a Boolean conjunction).

Under the preference bias, the learner imposes a preference ordering over the set of hypotheses and attempts to find the "best" hypothesis $\hat{F}$ according to this ordering. In this article, we will assume that the preference ordering is a total ordering, and we will let index $I(\hat{F})$ denote the numerical position of $\hat{F}$ in this ordering. The preference bias can be implemented by attempting to find a consistent hypothesis $\hat{F}$ of low index.

3.1 Restricted Hypothesis Space Bias

The first major result that we will discuss concerns concept learning with a restricted hypothesis space bias. Suppose that we are given m training examples labeled according to the correct concept F. The examples are drawn independently from U according to some unknown probability distribution $P(u)$. We are also given a restricted hypothesis space H. Our algorithm will attempt to find an hypothesis $\hat{F} \in H$ that is consistent with all m training examples. Assuming that such an $\hat{F}$ can be found, what is the probability that it has error greater than ϵ?

To answer this question, let us define the set $H_{bad} = \{h_1, \ldots, h_l\}$ to be the set of hypotheses in H that have error greater than ϵ. What we will compute is the probability that, after m examples have been processed, there is some element of H_{bad} that is consistent with the training examples. If this probability is small enough, then (with high probability) the only hypotheses remaining in H that are consistent with the training examples are hypotheses with error less than ϵ. Hence, if our learning algorithm finds a consistent hypothesis $\hat{F} \in H$, that hypothesis is probably approximately correct.

Let us begin by considering a particular element $h_1 \in H_{bad}$. What is the probability that h_1 is consistent with one randomly-drawn training example? It is just the probability that the training example was drawn from the region of U *outside* the shaded area of Figure 1. This probability is greatest when $\text{error}(F, h_1) = \epsilon$. That is, h_1 is as good as possible without being approximately

[1] This terminology is informal. Technically, we should say that $\hat{F}$ is produced by an *algorithm* that is probably approximately correct.

correct. So, the probability that h_1 is consistent with a single training example is no more than $1-\epsilon$.

It follows that the probability that h_1 is consistent with all m randomly-drawn training examples is no more than $(1-\epsilon)^m$. We will write this as $P^m[\text{consist}(h_1)] \leq (1-\epsilon)^m$.

Now let us consider all of the hypotheses in H_{bad}. What is the probability that after m examples there is some element of H_{bad} that has not been eliminated from consideration? This is

$$P^m\left[\text{consist}(H_{bad})\right] = P^m\left[\text{consist}(h_1) \vee \ldots \vee \text{consist}(h_l)\right].$$

Because the probability of a disjunction (union) of several events is no larger than the sum of the probabilities of each individual event,

$$P^m\left[\text{consist}(H_{bad})\right] \leq |H_{bad}| \cdot (1-\epsilon)^m.$$

In the worst case, $H_{bad} = H$ (i.e., there are no approximately correct hypotheses in H). Hence,

$$P^m\left[\text{consist}(H_{bad})\right] \leq |H|(1-\epsilon)^m.$$

Now that we have an expression for the probability that $\hat{F}$ is not approximately correct, we can set this equal to δ and solve for m to obtain a bound on the number of training examples to guarantee that $\hat{F}$ is probably approximately correct.

$$|H|(1-\epsilon)^m \leq \delta$$

is true if and only if

$$m \geq \frac{1}{-\ln(1-\epsilon)}\left(\ln\frac{1}{\delta} + \ln|H|\right).$$

But since $\epsilon \leq -\ln(1-\epsilon)$ over the interval [0,1), it suffices that

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

This gives us Theorem 1:

Theorem 1. (Blumer et al 1987) *Let H be a set of hypotheses over a universe U, S be a set of m training examples drawn independently according to $P(u)$, $\epsilon, \delta > 0$, then if $\hat{F} \in H$ is consistent with all training examples in S and*

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

then the probability that $\hat{F}$ has error greater than ϵ is less than δ.

Using this theorem, we can obtain bounds on the number of examples required for learning in various hypothesis spaces. Consider, for example, the set of hypotheses H_{conj} that can be expressed as simple conjunctions of n Boolean variables. There are 3^n such hypotheses, since in a conjunction, each variable may appear negated, un-negated, or it may be missing. Applying Theorem 1, we see that if

$$m \geq \frac{1}{\epsilon}\left(\ln\frac{1}{\delta} + n\ln 3\right)$$

Table 1: Sizes of various concept description languages

Hypothesis Space	Size
Boolean conjunctions	3^n
k-term-DNF	$2^{O(kn)}$
k-DNF	$2^{O(n^k)}$
k-CNF	$2^{O(n^k)}$
k-DL	$2^{O(n^k k \lg n)}$
LTU	$2^{O(n^2)}$
DNF	2^{2^n}

then any hypothesis consistent with the examples will be PAC. Furthermore, the number of examples required grows only linearly with the number of features.

Likewise, consider the set of hypotheses that can be expressed as linear threshold functions over n Boolean variables, $x_1, \ldots, x_n$. A linear threshold function is described by a vector of real-valued weights, $w_1, \ldots, w_n$ and a real-valued threshold, θ. It returns a 1 if $\sum_{i=1}^{n} w_i x_i \geq \theta$. In (Muroga 1971), it is shown that $|H| \leq 2^{n^2}$. Hence, if

$$m \geq \frac{1}{\epsilon}\left(\ln\frac{1}{\delta} + n^2 \ln 2\right)$$

then any linear threshold function consistent with the training examples is PAC.

Table 1 shows the hypothesis space sizes for several popular concept representations. The class k-term-DNF contains Boolean formulas in disjunctive normal form with at most k disjuncts (i.e., a k-term disjunction where each term is a conjunction of unlimited size). The class k-DNF contains Boolean formulas in disjunctive normal form in which each conjunction has at most k variables (i.e., a disjunction of any number of conjunctive terms, but each conjunction is limited to length k). A class analogous to k-DNF is the class k-CNF. Each formula in k-CNF is a conjunction of clauses (disjunctions). Each clause contains at most k variables. The class k-DL is the class of decision lists introduced by Rivest (1987). A decision list is an ordered list of pairs of the form $\langle (F_1, C_1), \ldots, (F_i, C_i), \ldots, (T, C_{l+1}) \rangle$. Each F_i is a Boolean conjunction of at most k variables, and each C_i indicates the result (either 0 or 1). A decision list is processed like a lisp COND clause. The pairs are considered in order until one of the F_i is true. Then the corresponding C_i is returned as the result. By convention, the condition for the last pair in the list, F_{l+1}, is always true (T). The class LTU contains all Boolean functions that can be represented by linear threshold units.

For comparison, we also show the full class DNF, consisting of any arbitrary Boolean expression in disjunctive normal form. DNF is capable or representing any of the Boolean functions.

Note that for fixed k, each of these classes (except DNF) requires only a polynomial number of training examples to guarantee PAC learning according to Theorem 1.

Theorem 1 gives results for finite hypothesis spaces. However, there are many applications in which hypotheses contain real-valued parameters, and consequently there are uncountably many hypotheses in these spaces. In spite of this, it is still possible to develop learning algorithms for these cases. Consider for example the universe U consisting of points on the real number line. An hypothesis $F \subset U$ describes some subset of these points. Suppose we restrict our hypotheses to be single closed intervals over the real line (i.e., our hypotheses have the form $[a, b]$). One algorithm for discovering closed intervals would be to let a be the value of the smallest positive example and b be the value of the largest positive example. How many training examples are needed to ensure that this algorithm will return an interval that is probably approximately correct?

Answers for problems such as this can be obtained using a measure of bias called the Vapnik-Chervonenkis dimension (VC-dimension). The idea behind the VC-dimension is that although an hypothesis space may contain uncountably many hypotheses, those hypotheses may still have restricted expressive power. Specifically, we will say that a set of hypotheses can *completely fit* a collection of examples $E \subset U$ if, for every possible way of labeling the elements of E positive or negative, there exists an hypothesis in H that will produce that labeling. The VC-dimension will be defined to be the size $|E|$ of the largest set of points that H can completely fit. This will provide a measure of the expressive power of H.

To continue with the real-interval illustration, let us consider the set of two points $E = \{3, 4\}$. There are four different ways that these two points can be labeled as positive or negative, corresponding to four different training sets:

$$S_0 = \{\langle 3, 0\rangle, \langle 4, 0\rangle\}$$
$$S_1 = \{\langle 3, 0\rangle, \langle 4, 1\rangle\}$$
$$S_2 = \{\langle 3, 1\rangle, \langle 4, 0\rangle\}$$
$$S_3 = \{\langle 3, 1\rangle, \langle 4, 1\rangle\}$$

For each possible labeling, there is a real interval that will produce that labeling:

S_0 can be labeled by $[0, 1]$
S_1 can be labeled by $[4, 5]$
S_2 can be labeled by $[2, 3]$
S_3 can be labeled by $[2, 5]$

Hence, the hypothesis space H_{int} consisting of closed intervals on the real line can *completely fit* the set E. Indeed, it is easy to see that any set of two points can be fitted completely by H_{int}.

However, consider the set of points $E' = \{2, 3, 4\}$. The hypothesis space H_{int} cannot completely fit this set. In particular, there is no hypothesis in H_{int} that can label E' as follows:

$$S_4 = \{\langle 2, 1\rangle, \langle 3, 0\rangle, \langle 4, 1\rangle\}$$

This is because any interval containing 2 and 4 will also contain 3.

Since the VC-dimension of H is defined as the largest set of points that H can completely fit, it is easy to see that VC-dim$(H_{int}) = 2$.

A more interesting example concerns linear threshold units over arbitrary points in n-dimensional Euclidian space. A linear threshold unit is equivalent to a hyperplane that splits R^n into two half spaces. If a given set of training examples can be separated such that the positive examples are all on one side of the hyperplane and the negative examples are all on the other side, then the training examples are said to be *linearly separable*. When $n = 2$, it is easy to see that half-spaces (in this case, half-planes) can completely fit any set of three points. However, half-planes are unable to completely fit any collection of four points (i.e., some labelings of the points will not be linearly separable). In general, the VC-dimension for linear threshold units over n-dimensional Euclidian space is $n + 1$.

Intuitively, the VC-dimension is proportional to the logarithm of the size of the *effective* hypothesis space. Indeed, the following theorem shows how Theorem 1 can be extended using the VC-dimension:

Theorem 2. (Blumer et al 1989). *A set of hypotheses H is PAC learnable if*

$$m \geq \frac{1}{\epsilon} \max \left[4 \lg \frac{2}{\delta}, 8 \cdot VCdim(H) \lg \frac{13}{\epsilon} \right]$$

and the algorithm outputs any hypothesis $\hat{h} \in H$ consistent with S.

Using Theorem 2, we can tighten the bound on the number of examples required for learning linear threshold units to $O(\frac{1}{\epsilon}(n \ln \frac{1}{\epsilon} + \frac{1}{\delta}))$.

Perhaps the most interesting application of Theorem 2 (and its relatives) is to the problem of training feed-forward multi-layer neural networks. A difficulty with the practical application of these networks is to decide how large the network should be for each application. If the network is too large, it is easy to find a setting of the weights that is consistent with the training examples. However, the resulting network is unlikely to classify additional points in U correctly.

Baum and Haussler (1988) consider feed-forward networks of N linear threshold units and W weights. They show that if the weights can be set so that at least a fraction $1 - \frac{\epsilon}{2}$ of the m training examples are classified correctly and if

$$m \geq O\left(\frac{W}{\epsilon} \log \frac{N}{\epsilon}\right),$$

then the network is PAC with $0 < \epsilon < 1$ and $0 < \delta < O(e^{-\epsilon m})$.

The VC-dimension turns out to be a fundamental notion. It permits us to exactly characterize the set of learnable concepts, and it allows us to derive a lower bound on the number of examples needed for learning. These results are given in the following two theorems.

Theorem 3. (Blumer et al 1989) *A space of hypotheses H is PAC learnable iff it has finite Vapnik-Chervonenkis (VC) dimension.*[2]

Theorem 4. (Ehrenfeucht et al 1988). *Any PAC learning algorithm for H must examine*

$$\Omega\left(\frac{1}{\epsilon}\left[\ln \frac{1}{\delta} + VCdim(H)\right]\right)$$

training examples.

3.2 Preference Bias

With Theorems 1–4, we have a fairly complete understanding of learning with a restricted hypothesis space bias. Let us now briefly turn our attention to the problem of learning with a preference bias. Recall that a preference bias establishes an ordering over all of the hypotheses in H. We will let the index $I(F)$ be the numerical position of hypothesis F in this ordering. By definition, hypotheses with smaller index values $I(F)$ will be considered simpler than hypotheses with higher index values.

[2] It is possible to learn concept classes having infinite VC-dimension if the number of training examples is permitted to vary with the complexity of the concepts in the hypothesis space. See Linial et al (1989).

Now suppose we have an excellent learning algorithm that works as follows. For any given set of training examples S, it finds the hypothesis $\hat{F} \in H$ of lowest index that is consistent with S. It turns out that if the number of examples in S is sufficiently large and if the hypothesis found by the algorithm has sufficiently small index, then we can be quite confident that $\hat{F}$ is probably approximately correct. The reason is that for sufficiently large S, it is unlikely that we could have found such a simple (i.e., small index) hypothesis $\hat{F}$ that is consistent with the training examples.

Following (Blumer et al 1987), we can formalize this by letting H' be the space of hypotheses of index less than or equal to $I(\hat{F})$. The set H' can be viewed as the effective hypothesis space for our preference-bias algorithm for this particular sample S, and therefore, from Theorem 1, we can conclude that the number of examples required is

$$\frac{1}{\epsilon}\left(\ln\frac{1}{\delta} + \ln I(\hat{F})\right).$$

This result can be generalized to allow the learning algorithm to output an hypothesis $\hat{F}$ that has small, but not minimal, index. See Blumer et al (1987) for details.

The famous bias of Occam's Razor (prefer the simplest hypothesis consistent with the data) can thus be seen to have a mathematical basis. If we choose our simplicity ordering *before* examining the data, then a simple hypothesis that is consistent with the data is provably likely to be approximately correct. This is true regardless of the nature of the simplicity ordering, because no matter what the ordering, there are relatively few simple hypotheses. Therefore, a simple hypothesis is unlikely to be consistent with the data by chance.

Another way of thinking about this result is to view learning programs as data compression algorithms. They compress the training examples into an hypothesis, $\hat{F}$, by taking advantage of some predefined encoding scheme (i.e., simplicity ordering). If the data compression is substantial (i.e., the number of bits needed to represent the hypothesis is much less than the number of training examples), then the hypothesis is likely to be approximately correct.

3.3 Noisy Data

All of the results described above have assumed that the training examples are complete and correct. Unfortunately, there are many applications where the training data are incomplete and incorrect. For incorrect training examples—that is, examples that are incorrectly classified—all of the results discussed above can be generalized as follows. Instead of trying to find a concept $\hat{F} \in H$ that is consistent with all of the training examples, it suffices to find an $\hat{F}$ that is consistent with fraction $1 - \frac{\epsilon}{2}$ of the training examples. Theorems 1–4 still apply under these conditions with some slight adjustments (see Appendix 3 of Blumer et al 1989).

3.4 Computational Complexity

In our review so far, we have only considered what is called the *sampling complexity*—that is, the number of training examples required to guarantee PAC learning. There is a second aspect of learning that has also been investigated within the Valiant framework, namely, the *computational complexity* of finding an hypothesis in H consistent with the training examples.

If we look again at Theorem 1, we see that the number of examples required for learning is proportional to the log of the size of the hypothesis space. This means that with a linear number

Table 2: Computational complexity of finding a consistent hypothesis.

Hypothesis Space	Time Complexity
Boolean conjunction	Polynomial
k-term-DNF	NP-hard
k-DNF	Polynomial
k-CNF	Polynomial
k-DL	Polynomial
LTU	Polynomial
k-3NN	NP-hard

of examples, we can learn an exponential number of hypotheses. The most trivial algorithm for finding an hypothesis consistent with the examples would simply enumerate each hypothesis in H and test it for consistency with the examples. However, when there are exponentially many hypotheses, this approach will require exponential time. Therefore, the challenge is to find ways of computing a consistent hypothesis by analyzing the training examples more directly. Our goal is to find algorithms that require time polynomial in the number of input features n and in $\frac{1}{\epsilon}$ and $\frac{1}{\delta}$.

Table 2 shows the computational complexities for the best known algorithms for several hypothesis spaces. Following Valiant, we say that an hypothesis space H is polynomially learnable if *(a)* only a polynomial number of training examples are required (as a function of n, $\frac{1}{\epsilon}$, and $\frac{1}{\delta}$) and *(b)* a consistent hypothesis from H can be found in time polynomial in n, $\frac{1}{\epsilon}$, and $\frac{1}{\delta}$. Hence, from the table, we can see that conjunctions, k-DNF, k-DL, and the linear threshold units are all polynomially learnable. The hypothesis space k-3NN consists of feed-forward neural networks containing two layers of linear threshold units (often called three-layer networks). The first layer of units (usually called the "hidden layer") contains exactly k units. There are robust proofs that this hypothesis space is not polynomially learnable (Judd 1987, 1988; Blum & Rivest 1988; Lin & Vitter 1989).

As an example of a polynomial-time learning algorithm, consider the following algorithm for learning Boolean conjunctions. We will represent a conjunction C as a list of Boolean variables or their negations. Given a collection S of training examples, we find the first positive example p_1 in that list and initialize C to contain all of the variables (or their negations) present in that positive example (if there are no positive examples, we exit and guess the null concept, $x_1 \wedge \neg x_1$). Then for each additional positive example p_i, we delete from C any Boolean variables appearing in p_i with a different sign than they appear in C. After processing all of the positive examples, we check all of the negative examples to make sure that none of them are covered by C. Finally, we return C as the answer.

As an example, consider the following positive examples:

$$\langle(0\ 1\ 1\ 0), 1\rangle$$
$$\langle(1\ 1\ 1\ 0), 1\rangle$$
$$\langle(1\ 1\ 0\ 0), 1\rangle$$

After processing the first example, $C = \{\neg x_1, x_2, x_3, \neg x_4\}$; After processing the second example, $C = \{x_2, x_3, \neg x_4\}$; after the third example, $C = \{x_2, \neg x_4\}$. This algorithm requires $O(nm)$ steps.

Surprisingly, smaller hypothesis spaces are not always easier to learn. For example, the space k-term-DNF is a proper subspace of the space k-CNF, yet k-CNF is polynomially learnable but k-term-DNF is not (Pitt & Valiant 1988). Similarly, the space of Boolean threshold units (i.e., linear threshold units in which the weights are all Boolean) is not polynomially learnable, but LTU (which properly contains it) is. One explanation for this is that in some cases, by enlarging the hypothesis space, it becomes easier to find an hypothesis consistent with the training examples. The larger space provides more freedom to choose the syntactic form of the hypothesis. Another explanation is that different representations, even of the same space, have different computational properties. Hence, some representations for concepts are easier to relate to the representation of the training examples.

These observations indicate that if we want to prove that learning a concept class is computationally intractable, we need to show that it is intractable *regardless of the representation employed by the learning algorithm.* In other words, suppose the correct concept F can be represented by a k-term-DNF formula. Although the problem of finding a k-term-DNF formula consistent with a training sample for F is NP-complete, we know that in polynomial time we can find an $\hat{F}$ represented as an equivalent k-CNF formula. Hence, we can construct an algorithm that can learn every concept in k-term-DNF by using hypotheses represented in k-CNF.

This point is particularly important for classes, such as k-3NN, where although it is intractable to find a consistent hypothesis using k hidden units, it might be easier to find a consistent hypothesis using $k' > k$ hidden units. If k' is only moderately bigger than k, the number of training examples required to guarantee PAC learning would still be polynomial. In general, if s is the number of bits required to represent the correct hypothesis F, then any algorithm that can represent $\hat{F}$ using $p(s)$ bits (where p is some polynomial) will still have polynomial sample complexity.

The question of whether every concept in k-3NN can be learned by finding (in polynomial time) a concept in k'-3NN (where $k' \leq p(k)$ for some polynomial p) is open. However, for two other important concept classes, the analogous questions have been answered negatively.

Let $DFA(s)$ be the space of concepts that can be represented as deterministic finite state automata of size $\leq s$. If S is a training sample for a concept $F \in DFA(s)$, then the problem of finding an hypothesis $\hat{F} \in DFA(p(s))$ consistent with S, for some polynomial p is NP-complete (Pitt & Warmuth 1988).

Similarly, if $BF(s)$ is the space of concepts that can be represented as boolean formulas of size $\leq s$ and if S is a training sample for a concept $F \in BF(s)$, then the problem of finding an hypothesis $\hat{F} \in BF(p(s))$ consistent with S, for some polynomial p is as hard as factoring integers (Kearns & Valiant 1988, 1989). In fact, this result can be strengthened to apply to *any* representation language in which $\hat{F}$ has size $\leq p(s)$.

An important way of looking at these results is from the perspective of Occam's Razor. Consider the class of all Boolean formulas and suppose we adopt the bias of preferring shorter formulas. The problem of finding the smallest Boolean formula consistent with a set of training examples has long been known to be NP-complete (Gold 1978). However, we might settle for an approximation to Occam's Razor—we could accept any Boolean formula that is of size $\leq p(s)$, where s is the size of the smallest Boolean formula consistent with the data. If we assume that factoring is hard, these results imply that there is no polynomial time algorithm for finding these "nearly simplest" hypotheses.

In short, it appears that there are "simple" concepts (i.e., that can be represented by polynomial-sized finite state machines or regular expressions) that cannot be discovered by any learning algorithm using any representation. Nature may be simple, but (in the worst case) no computing device can reveal that simplicity in polynomial time (unless $P = NP$, of course).

3.5 Summary

The Valiant theory allows us to quantify the role of bias in inductive learning. The main implication of this theory is that there are no efficient, general purpose inductive learning methods. Specifically, in order to learn using a polynomial number of training examples, by Theorem 4 the VC-dimension must be a polynomial function of n, $\frac{1}{\epsilon}$, and $\frac{1}{\delta}$. The VC-dimension of the entire space of 2^{2^n} Boolean functions over n variables is clearly 2^n, so it is impossible to learn arbitrary Boolean functions using only a polynomial number of examples.

On the positive side, the theory states conditions under which we can determine, with high confidence, whether a given learning algorithm has succeeded. For a given bias, the theory says that *if* a consistent hypothesis $\hat{F} \in H$ can be found and the number of examples m is large enough, then $\hat{F}$ is probably approximately correct. Unfortunately, the hypothesis space H must constitute only a small fraction of the possible hypotheses, and therefore any particular learning algorithm is unlikely to succeed for a randomly chosen concept $F \subset U$. Indeed, it is because H is a small fraction of the space of possible hypotheses (2^U) that we can have statistical confidence in the results of the learning algorithm.

Hence, for a particular application, the vocabulary of features chosen to represent training examples and hypotheses must allow a consistent $\hat{F}$ to be found. In many applications (Michalski & Chilausky 1980, Quinlan et al 1986), this has turned out to be easily achieved, but there are others where it has been quite difficult (Quinlan 1983).

4 RECENT DEVELOPMENTS IN PRACTICAL LEARNING ALGORITHMS

There have been many interesting developments in practical learning algorithms—too many to permit a complete review here. Therefore we will focus on three significant directions: *(a)* improvements to decision tree induction algorithms, *(b)* the back-propagation algorithm for multi-layer feed-forward neural nets, and *(c)* "hybrid" algorithms.

4.1 Improvements to Decision Tree Methods

For many years, the most popular concept learning algorithm has been Quinlan's (1983, 1986a) ID3. ID3 is a top-down recursive algorithm for constructing a decision tree. Points in U are represented as feature vectors (i.e., a point $u \in U$ is represented by $\langle f_1(u), f_2(u), \ldots, f_k(u)\rangle$, where the f_i are Boolean features). Here is a sketch of the basic algorithm:

$ID3(S)$
  If S contains only positive examples, return $+$
  Elseif S contains only negative examples, return $-$
  Else choose the best feature f_i to be the root of the tree
    partition S into
      $S_i^0 = \{s \in S | f_i(s) = 0\}$ and

$S_i^1 = \{s \in S | f_i(s) = 1\}$.
return tree with root f_i, left subtree $ID3(S_i^0)$, and
right subtree $ID3(S_i^1)$.

The best feature f_i is the feature with highest "information gain." This is a measure of how much information about the correct class $F(s)$ is obtained by knowing $f_i(s)$. It can be computed as follows. First, let $n = |\{s \in S | F(s) = 0\}|$ and $p = |\{s \in S | F(s) = 1\}|$. These simply count up the number of positive and negative examples in the training set S. Then, compute $n_{ij} = |\{s \in S_i^j | F(s) = 0\}|$ and $p_{ij} = |\{s \in S_i^j | F(s) = 1\}|$. With these, define

$$I(p_{ij}, n_{ij}) = -\frac{p_{ij}}{p_{ij} + n_{ij}} \lg \frac{p_{ij}}{p_{ij} + n_{ij}} - \frac{n_{ij}}{p_{ij} + n_{ij}} \lg \frac{n_{ij}}{p_{ij} + n_{ij}}.$$

Then the information gain can be defined as

$$gain(f_i) = I(p, n) - \sum_{j=0}^{1} \frac{p_{ij} + n_{ij}}{p + n} I(p_{ij}, n_{ij}).$$

There are three major shortcomings of this algorithm. First, as the decision tree (and the recursive calls) become deeper, the number of training examples in the set S becomes so small that it is difficult to choose the root feature f_i wisely. In other words, because the algorithm operates by recursively subdividing the training set, eventually the decisions made by the algorithm lack statistical support.

The second shortcoming is that decision trees do not provide very compact representations for Boolean concepts in disjunctive normal form (DNF). For example, the smallest decision tree for the concept $(f_1 \wedge f_2) \vee (f_3 \wedge \neg f_4 \wedge f_5)$ contains 8 nodes, because the expression $f_3 \wedge \neg f_4 \wedge f_5$ appears twice as shown in Figure 2. This is sometimes called the "replication problem."

The third shortcoming is that the algorithm is a batch algorithm that requires all of the training examples in order to operate.

There are two techniques that have been developed to repair these shortcomings. The first two problems can be solved by converting the decision tree to a collection of production rules. This conversion process allows us to simplify the decision tree and express DNF concepts compactly. The third problem—that ID3 is a batch algorithm—has been solved by ID5, which is an incremental implementation of ID3. We describe these two techniques briefly.

The procedure for converting decision trees to production rules is described in Quinlan (1987a). It contains three steps. First, each leaf node in the decision tree is converted into an equivalent rule of the form

$$f_1 \wedge f_2 \wedge \ldots \wedge f_k \supset class,$$

where the f_i are the ancestors of the leaf node in the tree and the *class* is either $+$ or $-$.

Then, each of these rules is analyzed to prune useless conditions from the left-hand side. Each condition f_i is evaluated to determine whether it makes a statistically significant contribution to the rule. If not, then it is eliminated, and the analysis is repeated on the remaining conditions.

Once each rule has been pruned in this way, the entire collection of rules is analyzed to remove whole rules whose presence does not significantly improve the performance of the rule set on the training examples. Let R be the collection of rules, and let r be an element of R. Define c to be the number of training examples incorrectly classified by $R - \{r\}$ that are correctly classified by

Figure 2: Decision tree illustrating the replication problem. The right branch at each node is taken if $f_i = 1$.

R. This is the number of correct classifications that r creates. Let d be the number of training examples incorrectly classified by R that are correctly classified by $R - \{r\}$. The *advantage* of r is $c - d$, the net change in the number of training examples correctly classified by introducing r. The algorithm repeatedly selects the rule with lowest advantage and deletes it from R as long as the advantage is not positive.

Quinlan presents data showing that this procedure is capable of dramatically reducing the complexity of the learned concept and simultaneously improving the accuracy of the concept on unseen examples. For instance, in the domain of endocrinology (specifically discordant assay), the average number of nodes in the decision tress produced by 10 independent runs of ID3 was 52.4. This procedure converted those trees into an average of 1.8 rules and reduced the average error rate on unseen cases from 1.9% to 1.3%.

Utgoff (1988a, 1989) presents an incremental version of ID3 called ID5. ID5 processes the training examples one-at-a-time and produces an updated decision tree after each example. The basic idea is to grow the decision tree in the same top-down fashion (and using the same criterion for selecting the root of each subtree) as ID3. Hence, each time a new training example is presented, the example is filtered through the current decision tree until it reaches a leaf node, where it is stored. If the leaf node contains a mix of positive and negative examples, then a new feature is selected to split the node as in the ID3 algorithm.

A problem with this procedure is that the choice of the new feature to split a node is based on a relatively small number of training examples, and therefore it is likely to be incorrect. ID5 recovers from poor choices of these "splitting features" as follows. As the training example is being filtered through the current decision tree, ID5 reconsiders the choice of "splitting feature" at each internal node (starting with the root). If the information gain criterion would have chosen a different feature f_j instead of f_i, then ID5 searches each path in the subtree rooted at f_i to find an internal node that tests f_j (if none exists, then one is created at a leaf node). Then, the tree is rearranged so that f_j replaces f_i. This rearrangement process exploits the fact that the following two trees are equivalent:

Several rearrangements may need to be performed (recursively) in order to get f_i and f_j into this (locally balanced) configuration.

If every path through the final decision tree has been successfully traversed by a training example (without causing a rearrangement), then this tree will be the same one produced by ID3. In general, this condition is not satisfied, but the trees produced by ID5 are virtually identical to those produced by ID3. The only overhead required by ID5 is to store all of the training examples at the leaves

Figure 3: A simple three-layer feed-forward network.

and to maintain the statistics for computing information gain at each internal node. Utgoff (1989) describes ID5R, which is a modification of ID5 that guarantees that the tree produced by ID5 is the same as the tree produced by ID3.

There are many other extensions to ID3 that have been developed. Two particularly important extensions involve *(a)* making ID3 tolerant to noise in the training data and *(b)* finding ways to learn good decision trees even when the training data may contain missing values (i.e., training examples in which the values for some features are unknown). Quinlan (1986b, 1987b) discusses noise-tolerance techniques. The best technique simply applies the algorithm in the normal way (except that tree-growth is terminated when there is no feature with positive information gain) and then applying the procedure for converting the tree into production rules. Quinlan (1989) compares several techniques for learning in the presence of missing values.

4.2 The Backpropagation Algorithm for Training Multi-layer Neural Networks

Since the early days of computer science, researchers have been intrigued by the possibility of structuring programs in ways that mimic the neural structures of the human brain. Early work focused on immitating single neurons, and one of the best known artificial neurons was Rosenblatt's (1962) perceptron (or linear threshold unit). As we mentioned above, a perceptron is specified by a vector of real-valued weights $\mathbf{w}$ and a real-valued threshold θ. It accepts a vector of real-valued inputs $\mathbf{x}$ and outputs a 1 if $\mathbf{w} \cdot \mathbf{x} \geq \theta$ (and a 0 otherwise).

The perceptron was widely criticized because it can only implement a restricted class of functions—namely, functions that characterize a region of R^n bounded by a single hyperplane. Hence, although several efficient algorithms for learning perceptrons were discovered, research in this area nearly died out during the sixties and seventies.

In the past five years, however, interest in this area has exploded. There are many reasons for this, but one significant factor has been the exploration of networks containing multiple layers of neuron-like elements and the development of learning algorithms for these multi-layer networks.

Figure 3 shows a simple multi-layer feed-forward network. The input $\mathbf{x}$ values are fed simultaneously to a layer of simple neuron-like elements called "units." The outputs of these units are then fed simultaneously to a second layer of units, and so on. In general, it is possible to have arbitrarily many layers, but in practice, usually only a few layers are employed. In the network of Figure 3, the outputs of the first layer are all fed to a single unit in the second (final) layer, and its output comprises the output of the entire network. The units in all but the last layer are normally called "hidden units". Some authors describe the vector of inputs as an "input layer," with the consequence that Figure 3 is called a three-layer network, even though there are only two layers of units (and only one hidden layer).

Multi-layer feed-forward networks of linear threshold functions can implement a much wider range of functions than a single perceptron. Indeed, given enough hidden units, *any* function can be closely approximated (K. Hornik et al, unpublished manuscript). The difficulty is to find a learning algorithm that can process a collection of training examples and set the weights and thresholds of each unit correctly. One of the best algorithms for this purpose is the error back-propagation algorithm (Rumelhart et al 1986).

The goal of the error back-propagation algorithm is to minimize the squared error between the output of the network and the correct outputs provided in the training examples:

$$\text{minimize } E = \sum_{i=1}^{m} \left[net(\mathbf{x}_i) - F(\mathbf{x}_i) \right]^2 .$$

where $net(\mathbf{x}_i)$ is the output of the network on example i, and $F(\mathbf{x}_i)$ is the correct output supplied in the training example.

This is accomplished by performing gradient descent search in weight space. In other words, the algorithm iteratively computes a slight change in all of the weights (and thresholds) in the network in the direction of fastest decrease of E.

To apply gradient descent, it is necessary that the functions computed by the individual units be differentiable. Linear threshold units, because they are discontinuous at θ, lack this property. So the standard practice is to approximate the linear threshold unit by the logistic function,

$$y = \frac{1}{1 + e^{-(\mathbf{w}\cdot\mathbf{x}+\theta)}}.$$

Each unit in the network computes this function.

To describe the algorithm, it is useful to make the following definitions. Let n be the length of the input vector $\mathbf{x}$. Let us assign a number $j = n + 1 \ldots M$ to each unit in the network (where unit M is the output unit). Let y_j be the output value computed by unit j, for $j > n$, and $y_j = x_j$ otherwise. Let $w_{j,k}$ be the weight on the input to unit k that comes from the output of unit j. It is customary to view the threshold, θ as another weight corresponding to an input whose value is always -1. With this convention, let $w_{0,k}$ be the threshold for unit k, and let unit 0 always produce the value -1 (i.e., $y_0 = -1$). For (j, k) pairs that do not correspond to connections in the network, $w_{j,k} = 0$. Finally, the parameter ρ is called the learning rate.

The back-propagation algorithm starts by initializing the weights in the network to small randomly-chosen values. Then, for each training example, $\langle \mathbf{x}, c \rangle$, the weights are updated as follows. First, each layer in the network is evaluated in sequence, and the output values (y_j) are saved. Then, a generalized error value $\delta_M = (c - y_M)y_M(1 - y_M)$ is calculated. Each weight for the output unit is adjusted using this error value:

$$w_{j,M} := w_{j,M} + \rho \delta_M y_j .$$

Once the output layer has been updated, the hidden layers are updated, one at a time proceeding in reverse order. When updating the weights for unit j in a hidden layer, the generalized error value to use is

$$\delta_j = y_j(1 - y_j) \sum_k \delta_k w_{j,k},$$

where k ranges over all units to which the output of unit j is connected. The weights of unit j are updated according to the formula,

$$w_{i,j} := w_{i,j} + \rho \delta_j y_i.$$

The updating equations modify a weight $w_{i,j}$ in proportion to *(a)* the error committed by unit j and *(b)* the input value y_i. This makes sense intuitively, since the weight should not be changed if either *(a)* no error was committed or *(b)* the weight $w_{i,j}$ did not contribute the y_j because y_i was zero.

To obtain gradient descent, the learning rate ρ should be very small, and the weight changes should be accumulated over the entire training set before any weights are changed. In general, the training set must be processed many times (sometimes hundreds or thousands of times) before the weight values converge. Furthermore, it is not uncommon for the weight values to converge to a local optimum that is not a global optimum.

In practice, the weights are updated after every training example, the learning rate is set to be as large as possible, and the updating equations are modified to contain a *momentum term*. Let $\Delta w_{i,j}(t)$ be the change to weight $w_{i,j}$ during iteration t. The updating rule can then be written as

$$\Delta w_{i,j}(t) = \rho \delta_j y_i + \alpha \Delta w_{i,j}(t-1).$$

The parameter α is normally set to a large value, such as 0.9. The momentum term generally speeds convergence, because it allows us to increase the learning rate ρ without causing oscillations in the weight values. Additional improvements in the back propagation algorithm are reported in (Becker & le Cun 1988).

Finally, it should be noted that it is possible to have more than one output unit in the network. When several, closely related, concepts are being learned, they can share the values computed by hidden units, with the result that the representation of the several concepts is significantly compressed (and hence, the correctness of the learned concepts is probably enhanced).

There have been many successful applications of the back-propagation algorithm. For example, Sejnowski and Rosenberg (1987) trained a two-layer network (one hidden layer) to learn to pronounce English words. After learning on a sample of the 1000 most common words, their NETtalk program correctly pronounces 77% of the phonemes in a 20,012-word dictionary (which includes the 1000 words in the training set).

The major advantages of using neural-like networks for machine learning appear to be *(a)* the ability to learn a wide variety of concepts and *(b)* the ability to learn concepts involving real-valued features. A few recent studies have compared back-propagation with ID3 (Mooney et al 1989, Fisher & McKusick 1988, Weiss & Kapouless). The results generally show that a 2-layer neural network trained with back-propagation performs at the same level (and sometimes at a slightly better level) than ID3 when tested on unseen examples.

The major disadvantages of neural network learning methods are *(a)* the need to choose the number of hidden units and *(b)* the high cost of the learning process. The number of hidden units determines the "strength" of the bias of the learning system. If there are too many hidden units, then there will be many different settings of the weights that will be consistent with the training examples, so a trained network is unlikely to be probably-approximately correct. If there are too few hidden units, then there may be no setting of the weights consistent with the training examples. Research is continuing on techniques for automatically adjusting the number of hidden units during the learning process (Ash 1989, D. Rumelhart, personal communication).

Because of the NP-completeness result discussed above, it is unlikely that a general, efficient learning algorithm can be found for training multi-layer feed-forward networks. However, research into more restricted kinds of networks might discover representations with similar expressiveness that can be trained more efficiently.

4.3 Hybrid Algorithms

In each of the learning methods that we have reviewed thus far, the hypotheses are constructed from a single "combining mechanism." In ID3, for example, the combining mechanism is the decision tree. In multi-layer neural networks, the combining mechanism is the logistic unit. Many other algorithms employ the AND, OR, and NOT connectives of propositional logic (Michalski 1969, Haussler 1989). In the past few years, researchers have explored the properties of "hybrid" methods that mix two or more of these combining mechanisms in a single algorithm. The primary motivation for developing hybrid methods is that they may allow a learning algorithm to find a more compact representation for the hypothesis (and therefore enhance the performance of the hypothesis on unseen examples). We will review three hybrid methods: Stagger (Schlimmer & Granger 1986), Fringe (Pagallo 1989), and Perceptron trees (Utgoff 1988b).

The idea of hybrid methods was pioneered by Schlimmer with the Stagger system (although Utgoff is responsible for the term "hybrid"). Stagger combines a Bayesian weight-learning algorithm with a method for constructing Boolean expressions. Let $f_1, f_2, \ldots, f_n$ be the Boolean features used to represent each training example. The Bayesian learning algorithm computes the odds that a new example will be positive given the values of the n features: $odds[F(u) = 1 | f_1(u) = v_1, \ldots, f_n(u) = v_n]$. The key to making this computation feasible is to assume that the features are conditionally independent (given the value of $F(u)$) and apply the odds likelihood formulation of Bayes rule to obtain

$$odds\,[F(u) = 1 | f_1(u) = v_1, \ldots, f_n(u) = v_n] = odds\,[F(u) = 1] \cdot \prod_{i=1}^{n} L\,[f_i(u) = v_i] \qquad (1)$$

where $L\,[f_i(u) = v_i]$ is the likelihood ratio:

$$L\,[f_i(u) = v_i] = \frac{\Pr\,[f_i(u) = v_i | F(u) = 1]}{\Pr\,[f_i(u) = v_i | F(u) = 0]}.$$

It is straight-forward to estimate $odds\,[F(u) = 1]$ and $L\,[f_i(u) = v_i]$ from the training examples. Let S be the training sample, and let $n = |\{s \in S | F(s) = 0\}|$ and $p = |\{s \in S | F(s) = 1\}|$. These simply count up the number of negative and positive examples in the training set S. Furthermore, let $n_i = |\{s \in S | F(s) = 0, f_i(s) = v_i\}|$ and $p_i = |\{s \in S | F(s) = 1, f_i(s) = v_i\}|$. Then, the odds that an unseen example is a positive example is simply p/n. The likelihood ratio for $f_i(u) = v_i$ is estimated by

$$\frac{p_i \cdot n}{n_i \cdot p}.$$

To classify an unseen example u, the odds that u is positive are calculated using equation (1). If the odds are greater than 1, then the algorithm will predict that $\hat{F}(u) = 1$; otherwise, $\hat{F}(u) = 0$. It easy to show (by taking logarithms) that equation (1) is equivalent to a linear threshold function, and therefore, any concept representable by this Bayesian algorithm must be linearly separable.

To extend the range of concepts that can be represented (and learned), Stagger combines this Bayesian algorithm with a procedure for defining interesting Boolean combinations of the given features $\{f_i\}$. The learning process is incremental. When each training example is presented, equation (1) is evaluated to classify the example. If the classification is correct, the $odds\,[F(u) = 1]$ and the likelihood ratios are incrementally updated and processing continues with the next training example. If the classification is incorrect, Stagger introduces "new" features as Boolean combinations of the existing features and updates all likelihood ratios, including new ratios corresponding to the new features.

For example, when Stagger incorrectly classifies a positive example as negative, the algorithm selects the two features $f_i = v_i$ and $f_j = v_j$ in the training example whose likelihood ratios are largest and defines a new feature $f_k = 1 \equiv (f_i = v_i \lor f_j = v_j)$, which has the value 1 whenever either $f_i = v_i$ or $f_j = v_j$. This new feature will tend to boost the estimated $odds\,[F(u) = 1]$, and therefore increase the chances that the algorithm will correctly classify this example in the future.

Conversely, when a negative example is incorrectly classified as positive, Stagger finds the two features $f_i = v_i$ and $f_j = v_j$ in the training example whose likelihood ratios are smallest and defines the new feature $f_k \equiv (f_i = v_i \land f_j = v_j)$. This new feature will tend to pull down the estimated $odds\,[F(u) = 1]$, and therefore decrease the chances of incorrectly classifying this example as positive.

In addition to these two simple cases, there are four other heuristics that Stagger employs for introducing disjunctions, conjunctions, and negations of existing features. Stagger also employs heuristics for pruning features that turn out to be unnecessary. The net result is that Stagger is able to overcome the limitations of the Bayesian weight-learning algorithm by introducing Boolean combinations of the given features.

The Fringe algorithm (Pagallo 1989) is a hybrid algorithm that integrates decision trees and Boolean feature combinations. The general strategy is quite similar to (and inspired by) Stagger. Fringe begins by executing ID3 on the training set. Then, it analyzes the resulting decision tree and defines new features as Boolean combinations of existing features. It then discards the first decision tree and repeats the process—now considering the newly introduced features as well as the original features. This iteration continues until no new features are defined.

The heuristic for defining new features is simple: For every leaf node in the tree that is labeled $+$, Fringe defines a new feature as the conjunction of the parent and grandparent nodes of the leaf. Consider again the decision tree shown in Figure 2. For this tree, the heuristic will define the new features $f_6 = \neg f_4 \land f_5$ and $f_7 = f_1 \land f_2$. In the next iteration, ID3 will produce the tree shown in Figure 4. After analyzing this tree, Fringe will define $f_8 = f_6 \land f_3$. In the final iteration, ID3 will produce the tree shown in Figure 5.

By defining new features, Fringe is able to overcome some of the problems plaguing ID3. Recall that one problem with decision trees is that, when they are used to represent DNF expressions, many of the conjunctions in the expression must be replicated in the tree. Fringe can learn larger and more complex DNF expressions than ID3, because each conjunction in the expression eventually is defined as a single new "feature" that appears only once in the tree.

As a side-effect, this also overcomes another of ID3's problems. Recall that, because ID3 operates by recursively subdividing the training set, the choices of "root" features made toward the leaves of the tree are based on relatively little data and consequently lack statistical support. Fringe, because it eliminates replicated conjunctions, effectively pools all of the training examples that would have been split across the multiple replications of each conjunction. Hence, in subsequent

Figure 4: The decision tree from Figure 2 after one iteration of Fringe.

Figure 5: The final tree produced by Fringe.

iterations, ID3 can make better "root" feature choices.

Fringe's performance on large, randomly constructed DNF expressions is very impressive. For example, Pagallo presents a DNF expression containing 10 conjunctions defined over 64 attributes (each conjunction contains an average of 4.1 features). Fringe was presented with 1760 training examples for this concept and it converged after 10 iterations. When tested on 2000 additional examples, it classified them all correctly. Indeed, inspection of the decision tree showed that it was completely correct. By contrast, ID3 incorrectly classified 25.1% of the test examples after learning on the same 1760 training examples. Not unrelated is the fact that the decision tree produced by ID3 contains 101 nodes, while the final expression produced by Fringe contains the rough equivalent of 45.1 nodes (11 actual nodes, but each node tests a high level feature that is defined in terms of an average of 4.1 original features).

The last hybrid method that we will review is Utgoff's (1988b) perceptron tree algorithm. A perceptron tree is a decision tree in which the leaf nodes are perceptrons, and the internal nodes are standard decision nodes. Figure 6 shows a decision tree and the equivalent perceptron tree. In his perceptrons, Utgoff maintains one weight for each value of each feature, rather than just one weight per feature (this is called the symmetric-model of instance representation; Hampson & Volper 1986). These are shown in the figure as two rows of weights, one row corresponding to $f_i = 0$ and another corresponding to $f_i = 1$. The final weight (labeled θ) encodes the threshold. To evaluate each perceptron, a weight is multiplied by 1 if the corresponding feature value is present and by -1 if the corresponding feature is absent. (The threshold is always present.)

To see how this works, consider the example $\langle f_1 = 0, f_2 = 1, f_3 = 1, f_4 = 0\rangle$. To classify this example in the perceptron tree from Figure 6*(b)*, we could start at the root node and take the left branch, since $f_1 = 0$. At the next node, we would take the right branch, since $f_2 = 1$. Finally, we would evaluate the perceptron at the leaf over features f_3, and f_4. To do this, we would convert the training example into a feature vector $\langle -1, 1, 1, -1, 1\rangle$ corresponding to $\langle f_3 = 0, f_3 = 1, f_4 = 0, f_4 = 1, \theta\rangle$. The dot product of this vector with the weights in the perceptron is

$$\langle -1, 1, 1, -1, 1\rangle \cdot \langle 1, -1, 0, -1, 1\rangle \geq 0,$$

so the example is classified as positive.

The perceptron tree learning algorithm is an incremental algorithm that gradually expands the tree as training examples are processed. It begins by creating a single perceptron node as the root of the tree. As new examples arrive, the weights of this perceptron are updated [using the absolute error correction procedure from Nilsson (1965)] until either all of the examples are processed or else it is discovered that the training examples are not easily separated by a perceptron (explained below). When this is detected, the perceptron is discarded and the information gain criterion of ID3 is applied to choose a feature to form a decision node. During subsequent iterations, the learning algorithm will then create perceptrons at each of the two leaves of this decision node. (In order to apply the information gain criterion, it is necessary to maintain, at each perceptron node, counts of the number of positive and negative examples having each value of each feature. This is the same information computed by ID3 and ID5.)

To determine whether the examples are easily separated by a perceptron, Utgoff keeps track of the maximum and minimum values of each weight in the perceptron. Using this information, he maintains a counter C that counts the number of perceptron updates that have *not* changed the maximum or minimum value of any weight. If C becomes larger than the number of weights, the algorithm decides to replace the perceptron node with a decision node. The justification for this

(a)

(b)

Figure 6: A decision tree *(a)* and the equivalent perceptron tree *(b)* for the concept $(f_1 \vee f_2) \oplus (f_3 \wedge f_4)$ (Utgoff 1988b).

heuristic is that if the maximum and minimum weight values are not changing, then it is likely that the perceptron is failing to converge (since if it converged, no more perceptron updates would be needed).

Each of these three hybrid learning algorithms employs two different syntactic combination methods to find more compact representations for learned concepts. The hope is that expressive concept languages can be found that—unlike multi-layer neural networks—still have polynomial time learning algorithms. In the immediate future, it is expected that much more research will be pursued on the development and testing of hybrid methods.

4.4 Summary

Although the theoretical results discussed in the previous section show that there can be no general purpose learning algorithms that can learn all possible concepts efficiently, recent advances in practical inductive algorithms demonstrate that, for a wide range of concepts commonly encountered in applications, domain-independent learning methods are possible. The methods can learn concepts such as decision trees (ID3), disjunctive-normal-form Boolean expressions (Fringe), and disjunctions of linear threshold units (Perceptron trees) in reasonable times. Moreover, the back-propagation algorithm demonstrates that multi-layer feed-forward neural networks can be learned for non-trivial problems. This area is advancing rapidly, with many new algorithms and new applications developed each year.

5 EXPLANATION-BASED LEARNING

In the section on philosophical foundations, we discussed two different kinds of learning: acquisition of new knowledge (typically by analyzing training examples) and speed-up learning. Thus far, this review has focused only on concept learning from examples. In this section, we shift our attention to an important new method for speed-up learning, called *explanation-based learning*.

5.1 The Basic EBL Procedure

To introduce explanation-based learning (EBL), it is convenient to begin by considering traditional caching. Suppose we have an expensive-to-evaluate function, $f(x)$, that we will need to compute many times. If we frequently evaluate $f(x)$ on the same value of x, we can gain speed by maintaining a cache memory of $\langle x, f(x)\rangle$ pairs. Whenever the value of $f(x_i)$ is needed, we first search this cache memory for the pair $\langle x_i, f(x_i)\rangle$, and if it is found, we can immediately return the value for $f(x_i)$. If it is not found, we go ahead and call the expensive function $f(x_i)$ and then store the resulting value into the cache.

One of the main drawbacks of caching is that it only succeeds when *exactly the same* x value is encountered a second time. The technique of explanation-based learning can be viewed as a solution to this problem. Like caching, EBL maintains a memory for the results of previous problem solving activity. Unlike simple caching, though, the $\langle x, f(x)\rangle$ pairs in this memory are generalized so that for x values *similar to* previously computed values, we can efficiently compute the corresponding $f(x)$ value. In particular, the x and $f(x)$ expressions saved by EBL can contain pattern variables and tests for pattern applicability. When a new x value is presented, the EBL system must apply a pattern-matching procedure (typically unification) to determine whether this x value is similar to some previously-stored value. If so, then the variables in the corresponding $f(x)$ pattern are instantiated, and the solution is returned.

To illustrate the EBL method, consider the task of solving simple algebraic equations in one variable. Each instance of this task (i.e., an x value) is an equation involving only one variable (which we will denote by `y`) and the four arithmetic operators. A solution is an equation of the form `y = E`, where `E` is an expression containing only constants.[3] For example, given the problem `6 = 4 * y`, the solution is `y = 6/4`. A simple caching system would memorize the pair $\langle \texttt{6} = \texttt{4} * \texttt{y}, \texttt{y} = \texttt{6/4}\rangle$. However, by using EBL, we can instead memorize the generalized pair $\langle \texttt{V3} = \texttt{V4} * \texttt{y}, \texttt{y} = \texttt{V3/V4}\rangle$. To this pair, we must attach three applicability conditions: `V3` and `V4` must be constants and `V4` must not be equal to zero.

When a new problem, `3 = 2 * y` is presented to the system, it matches the stored pattern (with substitution $\{\texttt{V3/3}, \texttt{V4/2}\}$).[4] Furthermore, the three applicability conditions are satisfied. Therefore, the solution can be constructed by instantiating the stored solution pattern to obtain `y = 3/2`.

If there is no memorized pair that matches the new problem, then the system must solve the problem itself and store a new generalized problem/solution pair into memory. To construct the new pair, the explanation-based learning procedure maintains a record of the problem-solving steps performed to solve the problem. After the solution is obtained, this problem-solving record is analyzed to determine what other problems could be solved by applying *the same problem solving steps*. Two patterns are constructed: one describing these problems and another describing their solutions. The resulting pair is stored in memory as a generalized problem/solution pair.

For example, in the algebraic simplification task, the problem solving steps all involve applying algebraic simplification rules and performing simple tests. Table 3 gives a collection of rules and facts that formalize this task. In this table, we have employed standard logical (prefix) notation, so that, for example, the equation `6 = 4 * y` is written `eq(6,times(4,y))`. The symbol '`=`' is reserved for logical equality. The overall goal of problem solving is captured by Rule 3, which says that a problem is solved if it has the form `eq(y,E)` where `E` is an expression involving only constants (i.e., a constant expression, abbreviated `ce(E)`). Rules 1 and 2 describe two simple operations for rewriting equations (dividing both sides by a value; swapping the two sides of the equation). Rules 4 through 8 and Facts 1 through 4 define constant expressions as expressions constructed from the arithmetic operators and simple constants (denoted by `c(E)`). Finally, Facts 5 and 6 are needed to test the applicability of Rule 1. In a real system, Facts 1 though 6 would be implemented using the computer's arithmetic hardware.

To solve the problem `eq(6,times(4,y))`, a problem solving system could proceed as follows:

1. Apply Rule 2 to obtain `eq(times(4,y),6)`.
2. Apply Fact 5 to show that $\texttt{4} \neq \texttt{0}$.
3. Apply Rule 1 to obtain `eq(y,divide(6,4))`.
4. Apply Fact 4 to show that `c(6)`.

[3] In the following, we follow the Prolog convention of capitalizing pattern variables while keeping constants (and algebraic variables) in lower case.

[4] A substitution is a list containing pairs of the form `T1/T2`, which states that term `T1` should be replaced by the term `T2` in order to make the stored pattern match the new problem. See Nilsson (1980) for more details.

Table 3: Algebraic Simplification Rules and Facts

Rule 1:	F1 $\neq$ 0	$\rightarrow$	eq(times(F1,F2),F3)))= eq(F2,divide(F3,F1))
Rule 2:	eq(F1,F2) = eq(F2,F1)		
Rule 3:	ce(E)	$\rightarrow$	solved(eq(y,E))
Rule 4:	c(E)	$\rightarrow$	ce(E)
Rule 5:	ce(E1) & ce(E2)	$\rightarrow$	ce(times(E1,E2))
Rule 6:	ce(E1) & ce(E2)	$\rightarrow$	ce(plus(E1,E2))
Rule 7:	ce(E1) & ce(E2)	$\rightarrow$	ce(divide(E1,E2))
Rule 8:	ce(E1) & ce(E2)	$\rightarrow$	ce(minus(E1,E2))
Fact 1:	c(2)		
Fact 2:	c(3)		
Fact 3:	c(4)		
Fact 4:	c(6)		
Fact 5:	4$\neq$0		
Fact 6:	minus(4,2)$\neq$0		

5. Apply Rule 4 to show that `ce(6)`.
6. Apply Fact 3 to show that `c(4)`.
7. Apply Rule 4 to show that `ce(4)`.
8. Apply Rule 7 to show that `ce(divide(6,4))`.
9. Apply Rule 3 to show that `solved(eq(y,divide(6,4)))`.

The history of this problem-solving procedure can be viewed as a logical proof. Figure 7 shows the proof explicitly as an AND-tree. In the tree, rules appear as pure AND nodes, facts appear as leaf nodes, and unification steps are shown as vertical equalities ($||$). The variables in each rule have been renamed to avoid name conflicts. Table 4 lists all of the substitutions that are needed to make each rule apply. When these substitutions are composed, the result includes $\langle$`V1/divide`$(6,4)\rangle$ as expected.

There are many equivalent ways to describe how the EBL procedure constructs a generalized problem/solution pair. Visually, the simplest way is to view EBL as pruning all of the leaf nodes from the proof tree of Figure 7, recomposing the remaining substitutions, and re-expressing the tree as a problem/solution pair. The rationale for this procedure is that the leaf nodes describe specific facts about the particular problem that was solved; the interior of the proof tree describes the rules that were applied. Since the purpose of EBL is to determine what other problems could be solved by applying the same sequence of rules, it makes sense to construct a generalized proof tree containing only those rules.

Figure 8 shows the proof tree after the leaves have been pruned. The remaining substitutions are shown in Table 5. When these substitutions are composed, the final tree takes the form shown in Figure 9.

Figure 7: Proof tree for algebraic simplification

Table 4: Substitions required for Figure 7

V2/y	V5/times(4,y)
V1/divide(V3,V4)	V1/divide(V7,V8)
V4/4	V7/V9
V5/times(V4,V2)	V9/6
V6/V3	V8/V10
V6/6	V10/4

Figure 8: Pruning the leaves of the tree.

Table 5: Substitutions remaining after pruning.

`V2/y`	`V1/divide(V7,V8)`
`V1/divide(V3,V4)`	`V7/V9`
`V5/times(V4,V2)`	`V8/V10`
`V6/V3`	

Figure 9: The final proof tree after composing substitutions.

To extract the generalized problem/solution pair, EBL simply extracts the leaf `eq(V3,times(V4,y))` and the root `eq(y,divide(V3,V4))`. The remaining leaves provide the applicability conditions: `V4` $\neq$ `0`, `c(V3)`, and `c(V4)`.

There are many different ways to implement the EBL generalization procedure. The earliest appears in Fikes et al (1972). Subsequent improvements include DeJong & Mooney (1986) and Kedar-Cabelli & McCarty (1987).

In a rule-based system—such as the algebraic simplification system that we have been examining—there is generally no need to maintain a separate "cache" memory of problem/solution pairs. Instead, the results of EBL can be represented as a new (macro) rule to be added to the rule base. In this example, the new rule would be

```
Rule 9   V4≠0 & c(V3) & c(v4)   →   eq(V3,times(V4,y)) =
                                    eq(y,divide(V3,V4)).
```

This is the usual approach taken in EBL systems.

By this point, the reader must be wondering why this is called explanation-based learning. The answer is that in many learning situations, a learning system is presented with $\langle x, y \rangle$ pairs. The learning task is to *explain* why $y = f(x)$ and acquire a general rule for future application. A nice example of this is the LEAP system (Mitchell et al 1985), which learns VLSI design rules by "watching over the shoulder" of an expert designer. When the designer implements a functional specification (e.g., `(AND (OR p1 p2) (OR p3 p4))`) in a clever way (e.g., using three NOR gates: `(NOR (NOR p1 p2) (NOR p3 p4))`), LEAP applies its knowledge of Boolean logic to explain why this implementation works, and then generalizes the explanation to provide a general design rule.

In the sense of "knowledge in principle", LEAP already knows—before seeing the designer's example—that the triple-NOR circuit is a legal solution. The learning process involves converting this implicit knowledge into an explicit design rule that can be cached for future use. It is reasonable to ask whether there is any benefit to giving LEAP an $\langle x, y \rangle$ pair. After all, by applying program transformation methods such as partial evaluation (van Harmelen & Bundy 1988) the same triple-NOR rule could be discovered and cached. The advantage of providing the training example is that it focuses LEAP's efforts on problem/solution pairs that are likely to arise in practice.

From this perspective, explanation-based learning can be defined as follows (Mitchell el at 1986):

Given: A domain theory (e.g., the rules and facts in Table 3)
A target concept (e.g., `solved(eq(y,E))`)
A training example (e.g., `eq(4,times(6,y))`)
An operationality criterion or pruning policy (e.g., prune all leaves of the proof tree)

Find: An operational sufficient condition for the target concept.

The EBL method applies the domain theory to find a proof (explanation) of why the training example is an instance of the target concept. It then prunes this proof according to the operationality criterion, and extracts a generalized rule from the pruned proof. This rule is a sufficient condition for the target concept.

The operationality criterion (or pruning policy) specifies what kinds of applicability tests can be easily evaluated at execution time. For example, it is easy to verify that something is a constant or that a constant is non-zero. It is much more time-consuming to determine whether a large expression is made up only of constants and evaluates to a non-zero value. Hence, in our algebra example, we have effectively specified that the predicate `c(V)` is operational, but the predicate `ce(V)` is not. One can imagine many other pruning policies (including dynamic, context-specific policies), and some have been investigated (Braverman & Russell 1988, Keller 1987, Segre 1988).

One interesting pruning policy exploits multiple training examples. Normally, EBL only considers a single example. However, when several similar examples are available, one approach—sometimes called mEBL—is to compute a proof tree for each example and then find the largest subtree shared by all of these proofs. Everything else is pruned away, and a general rule is extracted from the shared subtree. An advantage of this approach is that the learned rule is typically more general and will be matched more often during subsequent problem solving (see Kedar-Cabelli 1988, Hirsh 1989, Cohen 1988, Pazzani 1988, Flann & Dietterich 1989).

Without this kind of pruning strategy, the rules learned by EBL are often overly specific. For example, if we give EBL the problem

```
eq(plus(2,3), times(minus(4,2),y)),
```

it will produce the rule

```
Rule 10   minus(V5,V6)≠0 &
             c(V3) & c(V4)
             c(V5) & c(V6)   →   eq(plus(V3,V4),times(minus(V5,V6),y)) =
                                 eq(y,divide(plus(V3,V4),minus(V5,V6))).
```

If, on the other hand, we use mEBL and give it the two problems `eq(6,times(4,y))` and `eq(plus(2,3),times(minus(4,2),y))`, then the new rule will be

```
Rule 11  V4≠0 & ce(V3) & ce(v4)  →  eq(V3,times(V4,y)) =
                                     eq(y,divide(V3,V4)).
```

This rule has pruned away the details of how ce(V3) and ce(V4) are checked—these conditions are thereby deferred until the rule is applied. The result is a more general, more useful (but potentially more expensive) rule.

This points up an important issue in any form of explanation-based learning. EBL is basically a process of converting problem-solving search (i.e., stringing together rules) into pattern-matching search (i.e., checking a large collection of problem/solution pairs to see which ones apply). Although this is usually a tradeoff of space against time, there are problems where the pattern-match cost can far exceed the problem-solving cost. As an example (due to Tambe & Newell 1988), consider the problem of determining whether there is a path between two specified nodes in a given graph representing a partial order. This is can be solved by computing the transitive closure of the graph, and it can be performed in time $O(n^3)$ for a graph of n nodes. Suppose now that whenever we find a path between two nodes, we apply EBL to extract a rule. Each such rule will describe the subgraph connecting the two nodes. Matching such rules against future graphs involves performing a graph sub-isomorphism computation, which is NP-complete. Hence, by applying EBL it is possible to convert a polynomial-time algorithm into an exponential-time algorithm.

5.2 Integrating EBL Into Problem-Solving Architectures

The past five years have seen the development of two problem-solving architectures that perform EBL automatically, as a side-effect of normal problem-solving activity: SOAR (Laird et al 1986, 1987) and Prodigy (Minton 1988a, 1988b). One goal of these architectures is to realize the long-held dream of creating a problem-solving system that automatically improves its performance with practice. To a limited extent, these systems succeed: For any program written according to certain conventions, these architectures will automatically speed up the program each time it is executed.

In these systems, the principal application of EBL is not to collect problem/solution pairs for the inputs and outputs of the user's program, but instead to acquire *control rules.* In other words, EBL is applied primarily at the meta-level rather than at the base-level of problem solving. Each of these architectures is a meta-level, deliberative architecture. For example, SOAR is a general-purpose problem solver that searches a problem space of states by applying operators until some goal is achieved. At the meta-level, SOAR confronts four basic decisions: *(a)* what goal should be processed next? *(b)* what problem space should be searched to achieve that goal? *(c)* what state in that problem space should be explored next? and *(d)* what operator (i.e., rule) should be applied to the selected state?

Like most meta-level problem solvers, SOAR operates in a continuous two-phase loop called the decision cycle. Each time through the loop, SOAR confronts one of these four meta-level problems and selects a solution. Then it executes the solution (e.g., applies the chosen operator) at the base level.

To solve the four meta-level problems, SOAR applies a collection of control rules that identify and rank candidates. For example, in the algebraic simplification domain, SOAR could learn a control rule such as

If current state matches eq(V3,times(V4,y))
and V3 and V4 are constants

Table 6: Meta-rules for Algebraic Simplification

Rule 1:	`F1` $\neq$ `0`	$\rightarrow$	`apply(op1,eq(times(F1,F2),F3)))) = eq(F2,divide(F3,F1))`
Rule 2:			`apply(op2,eq(F1,F2)) = eq(F2,F1)`
Rule 12:	`solved(S)`	$\rightarrow$	`solvable(S).`
Rule 13:	`solvable(apply(Op,S))`	$\rightarrow$	`solvable(S).`
Rule 14:	`solvable(apply(Op,S))`	$\rightarrow$	`good-operator(Op,S).`

and `V4` $\neq$ `0`
then a good rule to apply is Rule 2.

This control rule can be viewed as a memorized problem/solution pair—but now the problem is a meta-problem (what rule to apply to the given state), and the solution is an answer to the meta-problem (Rule 2).

To learn these kinds of control rules, all that we need to do is write a meta-level "domain theory" and apply the EBL procedure to explanations that are constructed using it. Table 6 gives a portion of such a meta-level domain theory for the algebraic simplificiation domain. Rules 1 and 2 are restatements of the corresponding rules in Table 3. These restatements are needed to give each simplification operator a name (e.g., `op1`, `op2`). The notation `apply(Op,S1)=S2` indicates that the result of applying operator `Op` to state `S1` is a new state `S2`. This explicit naming of operators is the key factor distinguishing the rules of the meta-level domain theory from the rules in a base-level domain theory.

The most important rule in Table 6 is Rule 14, which says that an operator `Op` is a good operator to apply to state `S` if it results in a solvable state. Rules 12 and 13 define a solvable state to be either a completely solved state or else a state that can be completely solved by recursively applying another operator. In short, a solvable state is one for which there exists some sequence of operators that can be applied to solve the problem.

Now let us see how we can learn the control rule given above for operator `op2` (i.e., Rule 2). Suppose we are given the base-level problem `eq(6,times(4,y))` as before. This time, however, our goal is to prove that `good-operator(op2,eq(6,times(4,y)))`. Figure 10 shows the required proof tree. It is very similar to the proof in Figure 7.

As with EBL applied to the base-level, we prune the leaves of this explanation tree, and the result is the following control rule:

```
Rule 15  V3≠0 & c(V1) &
                  c(V3)  →  good-operator(op2,eq(V1,times(V3,y)))
```

Similar reasoning can be applied in the same problem to learn a control rule for operator `op1`:

```
Rule 16  V3≠0 & c(V1) &
                  c(V3)  →  good-operator(op1,eq(times(V3,y),V1)).
```

When SOAR confronts a new (base-level) problem, such as `eq(3,times(2,y))`, it will again confront the meta-level problem of deciding which operator to apply. Rule 15 can then fire and recommend operator `op2`. After `op2` is applied, Rule 16 can fire and recommend operator `op1`, which will produce the solution.

Figure 10: Proof that `op2` is a good operator to apply.

The reader may wonder why meta-level control rules are worth learning, since we have already seen that base-level (macro) rules can solve this same problem directly—without the need to apply the various operators at run time. The answer is that in many domains (e.g., STRIPS robot planning), a few control rules can produce the same results as hundreds of base-level macro rules. This is the case in domains where it is easier to describe a general purpose (perhaps heuristic) strategy than it is to produce a list of generalized problem/solution pairs.

It is possible to learn meta-level control rules for any meta-level decision of interest. For example, we might want to learn rules that describe bad operators—operators that should *not* be applied to particular states. By defining `bad-operator(Op,S)` as a meta-level domain theory, such rules can be learned via EBL. Typically, a bad operator is defined to be one that converts a solvable state into an unsolvable state.

In Prodigy, meta-rules are also learned for the target concepts `sole-alternative(Op,S)` and `goals-interfere(G1,G2)`. A sole alternative is an operator that is the *only* operator that will result in a solvable state. Goal `G2` interferes with goal `G1` if any plan for achieving `G2` in states where `G1` is already achieved must undo `G1`.

Ideally, one would like to learn meta-rules for the concept of `best-operator(Op,S)`. However, to learn such rules, it would be necessary to perform a very expensive search to prove that applying operator `Op` to state `S` is the best way to solve the problem (i.e., results in the shortest, cheapest solution). In practice this is generally too expensive, so Prodigy and SOAR work with the weaker concepts of good and bad operators. In states where one operator is known to be good, but the value of other operators is unknown, Prodigy and SOAR will select the known good operator (even though one of the other operators might be better). This amounts to making the assumption that a good operator is the best operator in the absence of information to the contrary.

5.3 Lessons and Problems

Prodigy and SOAR have each been tested in many domains, and as a result, several important lessons have been learned.

First, the vocabulary of the domain theory must be chosen carefully in order to obtain improvements in problem-solving performance. In particular, if the vocabulary is not carefully designed, EBL can easily degenerate into simple caching of ungeneralized problem/solution pairs. For example, if the rules in the domain theory are very specific (e.g., `eq(times(8,y),4) = eq(y,divide(8,4))`) instead of very general (e.g., `eq(times(F1,F2),F3) = eq(F2,divide(F3,F1))`), then when the EBL procedure computes the set of problems that can be solved by applying *the same sequence of operators*, this set will contain only the original problem. This is most evident when rules for computing arithmetic are included in the domain theory (e.g., `times(4,5)=20`). Any time a rule of this kind is applied to evaluate a constant expression, the resulting explanation becomes very specific—which is why we did not simplify the constant expressions appearing above in our examples.

A similar difficulty can arise if the definition of a solved problem is very specific (e.g., the desired configuration in the 8-puzzle, Laird et al 1986).

Second, the quality of the rule learned by the system can be greatly affected by the quality of the explanation given to the EBL procedure. In some domain theories, for example, it is eventually necessary to evaluate arithmetic expressions in order to solve the given problem. However, if this evaluation occurs at the very end of problem solving (i.e., at the leaves of the explanation), it can be pruned, and the resulting rule will be quite general. On the other hand, if the simplifications are performed as soon as possible, it will not be possible to prune them from the explanation, and the learned rule will be very specific. In general, the best explanation for EBL is the shortest, most general one that can be found. Explanations exploiting special-case rules will result in learned rules that are also only applicable to a few special cases.

Third, some form of post-optimization of the learned rules is critical. In SOAR, learned rules are optimized by carefully ordering the conditions appearing on the left-hand side of the rule so that they can be tested most efficiently. In Prodigy, three techniques are applied to simplify learned rules: *(a)* partial evaluation, *(b)* condition ordering, and *(c)* simplification via domain theorems.

During partial evaluation, equalities, constructors (e.g., `CONS`), selectors (e.g., `CDR`), and logical connectives (e.g., `AND`, `FORALL`) are all simplified as much as possible. For example, `(AND A A)` is simplified to `A`. As with SOAR, conditions are carefully ordered to minimize the cost of testing the rule for applicability. Finally, new rules are compared to existing rules to determine whether facts from the domain can be applied to construct a simpler rule. For example, in the blocks world, every block must either be on the table, on another block, or held by the robot arm. This can be expressed as an axiom so that when a condition such as `(or (holding x) (on-table x) (on x z))` is constructed, it can be replaced by `TRUE`.

Simplification is attempted both within a single learned rule and between pairs of learned rules. Significant improvements can be obtained in the latter case. For example, in the STRIPS robot domain, Prodigy can discover one rule stating that it is possible to travel between two connected rooms when the door joining them is open. It can discover a second rule stating that it is possible to travel between two connected rooms when the door joining them is closed (i.e., by opening the door). Then, by applying the domain axiom that says a door must be either open or closed, it can combine these two rules into a single rule that says it is always possible to travel between two connected rooms.[5]

In experimental studies, Minton found that without post-optimization, the Prodigy system actually slowed down rather than speeding up during learning. The application of domain-specific axioms is alone responsible for a 30% speedup.

The fourth lesson from this research is that it is important to be selective in applying explanation-based learning. In Prodigy, for example, heuristics are evaluated to suggest particular points in the problem-solving process where EBL should be performed. Additionally, once a rule has been learned, it is subjected to a utility analysis that estimates the net benefit of including the rule in the system (i.e., the savings obtained when the rule succeeds versus the cost of matching the rule whether it succeeds or not). Without utility analysis, Prodigy obtains only a 35% speedup in the blocks world, whereas with utility analysis, Prodigy obtains a 110% speedup.

In Minton's Prodigy research, two other interesting results were obtained. First, Minton compared the control rules learned by Prodigy with control rules coded by humans. The human-coded rules performed better than the rules learned by Prodigy, but the differences were not great. Prodigy's rules reduced the time required to solve 100 scheduling problems to 43% of the time required without control rules, whereas the human-coded rules reduced the time to 30%. Furthermore, the human-coded rules contained several errors that were discovered and corrected after noticing cases where Prodigy's rules were performing better. Hence, the main result is that automatically-learned control rules are more complete and more correct than human-coded rules

[5]This example is from Minton (1988a), p. 72.

Figure 11: A robot planning problem.

(although ultimately, human-coded rules perform somewhat better). Substantial performance improvements can be obtained by learning control knowledge.

The second interesting study by Minton compared learned meta-level control rules to learned base-level macros. In two of his test domains (the STRIPS robot-world and a job-shop scheduling domain), base-level macros produced virtually no speedup at all, whereas the meta-level control rules produced very substantial (more than 100% speedups). In his third test domain (which was the simplest), selective learning of base-level macros obtained results very similar to the meta-level control rules (although the resulting plans were far from optimal). The main conclusion is that meta-level control rules can be significantly more effective than base-level macro rules in speedup learning.

5.4 Generalization-to-n

One important problem with explanation-based learning is its inability to learn iterative procedures. This has come to be called the *generalization-to-n* problem. Consider, for example, a robot that must pass through a sequence of rooms (`r2`, `r3`, `r4`, and `r5`) in order to get from room `r1` to the goal, room `r6` (see Figure 11). The solution is simply the following plan:

```
gothrudoor(r1,r2)
gothrudoor(r2,r3)
gothrudoor(r3,r4)
gothrudoor(r4,r5)
gothrudoor(r5,r6)
```

When EBL is applied to determine what other problems could be solved by this same plan, it will construct a rule that will apply only in situations where the robot is attempting to reach a destination room that is connected to the current room by a string of exactly four intermediate rooms. The problem is that EBL is unable to generalize to the case where the destination is n rooms away.

One approach to solving this problem is to analyze the explanation to find iterative structure. This iterative structure is then represented as a recursive rule, and the explanation is reexpressed using this recursive rule. In this case, the recursive rule is

```
traverse(N,Seq,Dest) & inroom(robot,R) →
      [N=1 & Seq=nil & connected(R,Dest) & gothrudoor(R,Dest)]
      or
```

```
[N≠1 & Seq=cons(First,Rest) & connected(R,First) &
gothrudoor(R,First) & traverse(N-1,Rest,Dest)].
```

Once the explanation has been re-represented using the recursive rule, all of the recursive calls to the rule can be pruned from the proof tree, and the remaining proof can be generalized by the EBL procedure. In this case, the proof tree, when pruned, collapses to the single statement `traverse(5,[r2,r3,r4,r5],r6)`, which when generalized, is converted into the general statement `traverse(N,Seq,Dest)`.

In general, the secret to successfully generalizing to n is to reformulate the proof so that the number of iterations, n, appears as an explicit argument to a recursive rule. Once this is achieved, the EBL procedure can generalize it to take any value.

This brief description has omitted several subtleties and alternative approaches to this problem. See Shavlik (1987), Shavlik & DeJong (1987), and Cohen (1988) for more details.

5.5 Imperfect Domain Theories

In order to successfully apply EBL, it is necessary to have a complete and correct domain theory—that is, a domain theory that can provide a correct explanation for every problem. How can EBL be extended to handle cases where the domain theory is incomplete (i.e., missing important rules) or incorrect (i.e., produces incorrect explanations)? Research on these questions is still in an early phase, but we will describe two techniques that provide partial solutions to these problems.

Let us first consider the case where a training example is presented to the system, but the domain theory is unable to produce a complete proof that the example is an instance of the target concept. In such situations, one approach that appears promising is to construct a maximal partial proof and then hypothesize new rules to fill the remaining "holes." Generally, each new rule is constructed by taking the "bottom" and "top" of the hole and converting them into the left- and right-hand sides of the rule. Hence, each hole is filled by exactly one new rule. This form of inference is a kind of *abduction* (Peirce 1931–1958), so this approach to repairing incomplete theories is sometimes called *abductive theory completion.*

In Wilkins (1988), for example, the ODYSSEUS learning system "watches over the shoulder" of a physician as the physician performs a diagnostic interview. Every time the physician asks a question, the learning system attempts to explain why that question is being asked. In one case where the physician is attempting to diagnose meningitis, the physician asks the patient if he has "visual problems." ODYSSEUS cannot find an explanation for this question. However, it can construct a partial explanation containing the following steps:

- The physician is trying to test the hypothesis that the patient has viral meningitis.
- Acute meningitis is evidence for viral meningitis.
- Photophobia is a kind of visual problem.
- Physicians usually ask a general question (i.e., visual problems) before specific subtypes (i.e., photophobia).

However, there is a missing connection between acute-meningitis and photophobia. ODYSSEUS knows that the explanation could be completed if photophobia is evidence for acute-meningitis. Hence, it proposes this new rule as a "hole filler." The new rule can then be tested by consulting a database of previous cases or by interacting with the physician. Similar systems have been developed by Hedrick (1976), Hall (1988), Berwick (1985), and VanLehn (1987).

In all of these systems, the process of constructing a maximal partial explanation is implemented by a parser. The domain theory is viewed as a collection of grammar rules, and the parser must find a maximal partial parse of the given example. This can be very expensive—it involves both top-down and bottom-up parsing. Furthermore, if the remaining "holes" are very large, the rules proposed to fill them will be very specific and ad hoc. Hence, this technique is primarily limited to cases where the domain theory is nearly complete, so that the remaining holes are easy to find and can be plausibly filled by single new rules.

Now that we have considered incomplete theories, let us turn our attention to domain theories that produce incorrect explanations. There are many causes of incorrect explanations. Perhaps the simplest is that the domain theory is overly general, so that it produces explanations when it should not. Such domain theories are called "promiscuous" domain theories, because they tend to be able to explain anything. In the Meta-DENDRAL system (Buchanan & Mitchell 1978), for example, the initial domain theory is a very weak "half-order" theory of mass spectrometry that can provide several alternative explanations for just about every data point it sees (including data points that are actually caused by thermal noise).

One approach to refining promiscuous domain theories is called Induction Over Explanations (IOE, Dietterich & Flann 1988). The idea is to collect a set of training examples that are all believed to have similar (true) explanations. The domain theory is employed to construct all possible explanations for each of these examples, and then an inductive learning algorithm is applied to these alternative explanations to find a single, maximally-specific shared explanation (i.e., a generalized explanation that explains all of the examples). If negative examples (i.e., examples that should not have any explanation) are also available, they can constrain the process further. The generalized explanation found by IOE can be adopted as the new, corrected domain theory. Flann and Dietterich (1989) have applied IOE to specialize a promiscuous domain theory for chess in order to develop correct domain theories for several tactical chess concepts (e.g., knight fork, skewer, etc.).

The Meta-DENDRAL system applied a similar technique to specialize its half-order theory to obtain a highly accurate, specialized domain theory for mass spectroscopy.

5.6 Summary

Explanation-based learning is a technique for improving the computational efficiency of reasoning programs. In its simplest form, EBL is a kind of generalized caching that acquires generalized problem/solution pairs (or equivalently, macro rules). When EBL is integrated into a meta-level problem solving architecture, it can be applied to learn control rules. There is some evidence that learning control rules is more effective for speeding up problem solving than learning base-level macro rules.

When EBL cannot be applied, some form of inductive learning must be introduced. Abductive theory completion is a technique for generating plausible new rules to extend an incomplete domain theory. Once generated, the rules must be tested—typically by performing statistical tests on a collection of examples. Induction over explanations is a technique for refining a promiscuous domain theory by finding a maximally-specific shared explanation. This area of combining inductive learning with explanation-based learning is currently very active.

6 CONCLUDING REMARKS

This review has covered four areas of machine learning where substantial progress has been made in the past five years: philosophical foundations, theory of inductive learning, practical algorithms for learning from examples, and explanation-based learning.

There are many topics that have been omitted—three of the most important require mention. First, there are many applications where the task is to discover new concepts (or patterns) in a collection of training examples. This task is often termed "clustering." Recently, for instance, a program called Autoclass was applied to a large database of infrared stellar spectra, and it discovered a new class of stars (Cheeseman et al 1988). Several interesting clustering algorithms have been developed in the past few years.

The second major omission is the paradigm of case-based reasoning (Kolodner 1988). In its purest form, case-based reasoning involves simply caching previous problem-solving experience (e.g., caching problem/solution pairs or caching examples and explanations) and then solving future problems by retrieving stored solutions to "similar" problems. Elaborations to the case-based reasoning approach include developing clever indexes for speeding retrieval and performing sophisticated "patching" or "tweaking" of retrieved solutions so that they solve the new problem. Several interesting applications of case-based reasoning have been developed (e.g., Koton 1988).

Finally, this article has not discussed the application of machine learning techniques to the development and refinement of expert systems. A few laboratory studies have shown that rules acquired through inductive learning can match or exceed the performance of rules acquired via intervewing experts (Michalski & Chilausky 1980, Quinlan et al 1986). Furthermore, several commercially-available expert system shells include inductive learning components. Hence, machine learning techniques are providing additional tools for aiding the construction of high-performance expert systems.

7 ACKNOWLEDGMENTS

The author is extremely grateful to Jim Bennett, William Cohen, Nicholas Flann, and David Haussler for reading drafts of this article.

8 BIBLIOGRAPHY

Ash, T. 1989. Dynamic node creation in backpropagation networks. Tech. Rep. ICS-8901. Institute for Cognitive Science, University of California, San Diego

Baum, E. B., Haussler, D. 1988. What size net gives valid generalization? Tech. Rep., Department of Physics, Princeton University, Princeton, NJ

Becker, S., le Cun, Y. 1988. Improving the convergence of back-propagation learning with second order methods. *Proceedings of the the 1988 Connectionist Models Summer School*, pp. 29–37. San Mateo, CA: Morgan-Kaufmann

Berwick, R. C. 1985. *The Acquisition of Syntactic Knowledge.* Cambridge, MA: MIT Press

Blum, A., Rivest, R. L. 1988. Training a 3-node neural network is NP-complete (extended abstract). *Proceedings of the 1988 Workshop on Computational Learning Theory*, pp. 9-18. San Mateo, CA: Morgan-Kaufmann

Blumer, A., Ehrenfeucht, A., Haussler, D., Warmuth, M. K., 1987. Occam's Razor. *Information Processing Letters.* 24:377–80

Blumer, A., Ehrenfeucht, A., Haussler, D., Warmuth, M. K. 1989. Learnability and the Vapnik-Chervonenkis dimension. *Journal of the ACM*, In press

Braverman, M. S., Russell, S. J. 1988. Boundaries of operationality. *Proceedings of the Fifth International Conference on Machine Learning*, pp. 221-34. San Mateo, CA: Morgan-Kaufmann

Buchanan, B. G., Mitchell, T. M. 1978. Model-directed learning of production rules. In *Pattern-Directed Inference Systems,* eds. D. A. Waterman, F. Hayes-Roth, 297–312. New York: Academic Press

Cheeseman, P., Self, M., Kelly, J., Taylor, W., Freeman, D., Stutz, J. 1988. Bayesian classification. *AAAI-88: Proceedings of the Seventh National Conference on Artificial Intelligence*, pp. 607–11. San Mateo, CA: Morgan-Kaufmann

Cohen, W. 1988. Generalizing number and learning from multiple examples in explanation based learning. In *Proceedings of the Fifth International Conference on Machine Learning,* pp. 256–69. San Mateo, CA: Morgan Kaufmann

DeJong, G., Mooney, R. 1986. Explanation-based learning: an alternative view. *Machine Learning* 1(2):145–76

Dietterich, T. G. 1986. Learning at the knowledge level, *Machine Learning*, 1(3):287–316

Dietterich, T. G., Flann, N. S. 1988. An inductive approach to solving the imperfect theory problem. In *Proceedings of the AAAI Spring Symposium*, pp. 42–46. San Mateo, CA: Morgan-Kaufmann

Ehrenfeucht, A., Haussler, D., Kearns, M., Valiant, L. 1988. A general lower bound on the number of examples needed for learning. *Proceedings of the 1988 Workshop on Computational Learning Theory*, pp. 110-20. San Mateo, CA: Morgan-Kaufmann

Fagin, R. Halpern, J. Y. 1987. Belief, awareness, and limited reasoning. *Artificial Intelligence* 34(1):39–76

Fikes, R. E., Hart, P. E., Nilsson, N. J. 1972. Learning and executing generalized robot plans. *Artificial Intelligence* 3:251–288

Fisher, D. H., McKusick, K. B. 1988. An empirical comparison of ID3 and back-propagation. Tech. Rep. CS-88-14, Department of Computer Science, Vanderbilt, University

Flann, N. S., Dietterich, T. G. 1989. Induction over explanations: a method that exploits domain knowledge to learn from examples. *Machine Learning.* In press

Gold, E. M. 1978. Complexity of automaton identification from given data. *Information and Control*, 37:302-20.

Hall, R. P. 1988. Learning by failing to explain: Using partial explanations to learn in incomplete or intractable domains. *Machine Learning* 3(1):45–78

Hampson, S. E., Volper, D. J. 1986. Linear function neurons: Structures and training. *Biological Cybernetics* 53:203–17

Haussler, D. 1989. Quantifying inductive bias: AI learning algorithms and Valiant's learning framework. *Artificial Intelligence.* In press

Hedrick, C. L. 1976. Learning production systems from examples. *Artificial Intelligence* 7(1):21–49

Hirsh, H. 1989. Incremental version-space merging: a general framework for concept learning. Ph.D. thesis, Department of Computer Science, Stanford University

Judd, J. S. 1987. Learning in networks is hard. In *Proceedings of the First International Conference on Neural Networks*, San Diego, CA, pp. 685–92. IEEE

Judd, J. S. 1988. Learning in neural networks (extended abstract). In *Proceedings of the 1988 Workshop on Computational Learning Theory*, pp. 2–8. San Mateo, CA: Morgan-Kaufmann

Kearns, M., Valiant, L. G. 1988. Learning Boolean formulae or finite automata is as hard as factoring. Tech. Rep. 14-88, Aiken Computation Laboratory, Harvard University, Cambridge MA

Kearns, M., Valiant, L. G. 1989. Cryptographic limitations on learning boolean formulae and finite automata. In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, pp. 433–44.

Kedar-Cabelli, S. T. 1988. Formulating concepts and analogies according to purpose. Ph.D. thesis. Tech. Rep. ML-TR-26. Deparment of Computer Science, Rutgers University.

Kedar-Cabelli, S. T., McCarty, L. T. 1987. Explanation-based generalization as resolution theorem proving. In *Proceedings of the Fourth International Workshop on Machine Learning*, pp. 383-89. San Mateo, CA: Morgan-Kaufmann

Keller, R. M. 1987. The role of explicit contextual knowledge in learning concepts to improve performance. Ph.D. thesis. Tech. Rep. ML-TR-7, Department of Computer Science, Rutgers University

Kolodner, J. ed. 1988. *Proceedings of a Workshop on Case-Based Reasoning*, Clearwater Beach, FL. San Mateo, CA: Morgan-Kaufmann

Koton, P. 1988. Reasoning about evidence in causal explanations. *AAAI-88: Proceedings of the Seventh National Conference on Artificial Intelligence*, pp. 256–63. San Mateo, CA: Morgan-Kaufmann

Laird, J. E., Rosenbloom, P. S., Newell, A. 1986. Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning* 1(1):11–46

Laird, J. E., Newell, A., Rosenbloom, P. S. 1987. SOAR: An architecture for general intelligence. *Artificial Intelligence* 33(1):1–64

Lin, J-H., Vitter, J. S. 1989. Complexity issues in learning by neural networks (extended abstract). In *Proceedings of the 1989 Workshop on Computational Learning Theory*, pp. 118–33. San Mateo, CA: Morgan-Kaufmann

Linial, N., Mansour, Y., Rivest, R. L. 1988. Results on learnability and the Vapnik-Chervonenkis dimension. In *Proceedings of the 1988 Workshop on Computational Learning Theory*, pp. 56–68. San Mateo, CA: Morgan-Kaufmann

Michalski, R. S. 1969. On the quasi-minimal solution of the general covering problem. In *Proceedings of the Fifth International Federation on Automatic Control* 27:109–129

Michalski, R. S., Chilausky, R. L., Learning by being told and learning from examples: an experimental comparison of the two methods of knowledge acquisition in the context of developing an expert system for soybean disease diagnosis, *Policy Analysis and Information Systems* 4(2): 125-60.

Minton, S. 1988a. Learning effective search control knowledge: an explanation-based approach. Tech. Rep. CMU-CS-88-133. Carnegie-Mellon University

Minton, S. 1988b. Quantitative results concerning the utility of explanation-based learning. In *AAAI-88: Proceedings of the Seventh National Conference on Artificial Intelligence*, pp. 564–69. San Mateo, CA: Morgan-Kaufmann

Mitchell, T.M., Mahadevan, S., Steinberg, L. I. 1985. LEAP: A learning apprentice for VLSI design. In *IJCAI-85: Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pp. 573–80. San Mateo, CA: Morgan-Kaufmann

Mitchell, T. M., Keller, R. M., Kedar-Cabelli, S. T. 1986. Explanation-based generalization: a unifying view. *Machine Learning* 1(1):47–80

Mooney, R., Shavlik, J., Towell, G., Gove, A. 1989. An experimental comparison of symbolic and connectionist learning algorithms. In *IJCAI-89: Eleventh International Joint Conference on Artificial Intelligence.* In press

Muroga, S. 1971. *Threshold Logic and Its Applications.* New York: Wiley

Nilsson, N. J. 1965. *Learning Machines.* New York: McGraw Hill

Nilsson, N. J. 1980. *Principles of Artificial Intelligence*, Palo Alto: Tioga Publishing Co.

Pagallo, G. 1989. Learning DNF by decision trees. In *IJCAI-89: Proceedings of the Eleventh International Joint Conference on Artificial Intelligence.* In press

Pazzani, M. J. 1988. Learning causal relationships: an intergration of empirical and explanation based learning methods. Ph.D. thesis, Computer Science Department, University of California, Los Angeles, CA.

Peirce, C. S. S. 1931–1958. *Collected papers of Charles Sanders Peirce (1839–1914).* eds. Hartchorne, C., Weiss, P., Burks, A. Cambridge, MA: Harvard University Press

Pitt, L., Valiant, L. G. 1988. Computational limitations on learning from examples. *Journal of the ACM*, 35(4):965–84.

Pitt, L., Warmuth, M. K. 1989. The minimum DFA consistency problem cannot be approximated within any polynomial. In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, pp. 421–32.

Quinlan, J. R. 1983. Learning efficient classification procedures and their application to chess endgames. In *Machine learning: An artificial intelligence approach,* eds. Michalski, R. S., Carbonell, J., Mitchell, T. M. 1:463–82. San Mateo, CA: Morgan-Kaufmann

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

Quinlan, J. R. 1986b. The effect of noise on concept learning. In *Machine learning: An artificial intelligence approach,* eds., Michalski, R. S., Carbonell, J., Mitchell, T. M. 2:149–66. San Mateo, CA: Morgan-Kaufmann

Quinlan, J. R., Compton, P. J., Horn, K. A., Lazarus, L. 1986. Inductive knowledge acquisition: a case study. In *Proceedings of the Second Australian Conference on Applications of Expert Systems*, Sydney. To appear in *Applications of Expert Systems*, ed. Quinlan, J. R. Maidenhead: Academic Press

Quinlan, J. R. 1987a. Generating production rules from decision trees. In *IJCAI-87: Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pp. 304-7. San Mateo: Morgan-Kaufmann

Quinlan, J. R. 1987b. Simplifying decision trees. *International Journal of Man-Machine Studies*, 27: 221–34

Quinlan, J. R. 1989. Unknown atttribute values in induction. In *Proceedings of the Sixth International Workshop on Machine Learning*, pp. 164–68. San Mateo: Morgan-Kaufmann

Rosenblatt, F. 1962. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms.* Washington, D. C.: Spartan Books

Rivest, R. L. 1987. Learning decision lists. *Machine Learning* 2(3):229–46

Rumelhart, D. E., Hinton, G. E., Williams, R. J. 1986. Learning internal representations by error propagation. In *Parallel Distributed Processing*, eds. Rumelhart, D. E., McClelland, J. L. 1:318–362. Cambridge, MA: MIT Press.

Schlimmer, J. C., Granger, R. H. Jr. 1986. Incremental learning from noisy data. *Machine Learning* 1(3):317–54

Segre, A. 1988. *Machine Learning of Robot Assembly Plans.* Dordrecht: Kluwer Academic Publishers.

Sejnowski, T. J., Rosenberg, C. R. 1987. Parallel networks that learn to pronouce English text. *Complex Systems* 1:145–68

Shavlik, J. W. 1987. Augmenting and generalizing explanations in explanation-based learning. Ph.D. thesis. Department of Computer Science, University of Illinois, Urbana

Shavlik, J. W., DeJong, G. 1987. BAGGER: An EBL system that extends and generalizes explanations. In *AAAI-87: Proceedings of the National Conference on Artificial Intelligence*, pp. 516–20. San Mateo, CA: Morgan-Kaufmann

Tambe, M., Newell, A. 1988. Some chunks are expensive. In *Proceedings of the Fifth International Conference on Machine Learning*, pp. 451–58. San Mateo, CA: Morgan-Kaufmann.

Utgoff, P. E. 1988a. ID5: An incremental ID3. In *Proceedings of the Fifth International Conference on Machine Learning*, pp. 107–20. San Mateo, CA: Morgan-Kaufmann

Utgoff, P. E. 1988b. Perceptron trees: A case study in hybrid concept representation. In *AAAI-88: Proceedings of the National Conference on Artificial Intelligence*, pp. 601–6. San Mateo, CA: Morgan-Kauffman

Utgoff, P. E. 1989 Incremental induction of decision trees. *Machine Learning*. In press

Valiant, L. G. 1984. A theory of the learnable. *Communications of the ACM* 27:1134–42

van Harmelen, F., Bundy, A. 1988. Explanation-based generalization = partial evaluation. *Artificial Intelligence* 36(3): 401–12

VanLehn, K. 1987. Learning one subprocedure per lesson. *Artificial Intelligence* 31(1):1–40

Weiss, S., Kapouless, I. 1989. An empirical comparison of pattern recognition, neural nets, and machine learning classification methods. In *IJCAI-89: Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*. In press

Wilkins, D. C. 1988. Knowledge base refinement using apprenticeship learning techniques. In *AAAI-88: Proceedings of the National Conference on Artificial Intelligence*, pp. 646–51. San Mateo, CA: Morgan-Kaufmann