A THEORY OF THE LEARNABLE

L.G. Valiant

Aiken Computation Laboratory
Harvard University, Cambridge, Massachusetts

ABSTRACT. Humans appear to be able to learn new concepts without needing to be programmed explicitly in any conventional sense. In this paper we regard learning as the phenomenon of knowledge acquisition in the absence of explicit programming. We give a precise methodology for studying this phenomenon from a computational viewpoint. It consists of choosing an appropriate information gathering mechanism, the learning protocol, and exploring the class of concepts that can be learnt using it in a reasonable (polynomial) number of steps. We find that inherent algorithmic complexity appears to set serious limits to the range of concepts that can be so learnt. The methodology and results suggest concrete principles for designing realistic learning systems.

1. INTRODUCTION

Computability theory became possible once precise models became available for modelling the commonplance phenomenon of mechanical calculation. The theory that evolved has been used to explain human experience and to suggest how artificial computing devices should be built. It is also worth studying for its own sake.

The commonplace phenomenon of learning surely merits similar attention. The problem is to discover good models that are interesting to study for their own sake and promise to be relevant both to explaining human experience and to building devices that can learn. The models should also shed light on the limits of what can be learnt, just as computability does on what can be computed.

In this paper we shall say that a program for performing a task has been acquired by *learning* if it has been acquired by any means other than explicit programming. Among human skills some clearly appear to have a genetically preprogrammed element while some others consist of executing an explicit sequence of instructions that has been memorized. There remains a large area of skill acquisition where no such explicit programming is identifiable. It is this area that we describe here as learning. The recognition of familiar objects, such as tables, provide such examples. These skills often have the additional property that, although we have learnt them, we find it difficult to articulate what algorithm we are really using. In these cases it would be especially significant if machines could be made to acquire them by learning.

This paper is concerned with precise computational models of the learning phenomenon. We shall restrict ourselves to skills that consist of recognizing whether a *concept* (or predicate) is true or not for given data. We shall say that a concept Q has been learnt if a program for recognizing it has been deduced (i.e. by some method other than the acquisition from the outside of the explicit program).

The main contribution of this paper is that it shows that it is possible to design *learning machines* that have all three of the following properties.

(A) The machines can provably learn whole classes of concepts. Furthermore these classes can be characterized.

(B) The classes of concepts are appropriate and nontrivial for general purpose knowledge, and

(C) The computational process by which the machines deduce the desired programs requires a feasible (i.e. polynomial) number of steps.

A learning machine consists of a *learning protocol* together with a *deduction procedure*. The former specifies the manner in which information is obtained from the outside. The latter is the mechanism by which a correct recognition algorithm for the concept to be learnt is deduced. At the broadest level the suggested *methodology* for studying learning is the following: define a plausible learning protocol and investigate the class of concepts for which recognition programs can be deduced in polynomial time using the protocol.

*This research was supported in part by National Science Foundation Grant MCS-83-02385.

The specific protocols considered in this paper allow for two kinds of information supply. First the learner has access to a supply of typical data that positively exemplify the concept. More precisely, it is assumed that these positive examples have a probabilistic distribution determined arbitrarily by nature. A call of a routine EXAMPLES produces one such positive example. The relative probability of producing different examples is determined by the distribution. The second source of information that may be available is a routine ORACLE. In its most basic version, when presented with data it will tell the learner whether or not the data positively exemplifies the concept.

The major remaining design choice that has to be made is that of knowledge representation. Since our declared aim is to represent general knowledge, it seems almost unavoidable that we use some kind of logic rather than, for example, formal grammars or geometrical constructs. In this paper we shall represent concepts as Boolean functions of a set of propositional variables. The recognition algorithms that we attempt to deduce will be therefore Boolean circuits or expressions.

The adequacy of the propositional calculus for representing knowledge in practical learning systems is clearly a crucial question. Few would argue that this much power is not necessary. The question is whether it is enough. There are several arguments suggesting that such a system would, at least, be a very good start. First, when one examines the most famous examples of systems that embody preprogrammed knowledge, namely expert systems such as DENDRAL and MYCIN, essentially no logical notation beyond the propositional calculus is used. Surely it would be over-ambitious to try to base learning systems on representations that are more powerful than those that have been successfully managed in programmed systems. Second, the results in this paper can be negatively interpreted to suggest that the class of learnable concepts even within the propositional calculus is severely circumscribed. This suggests that the search for extensions to the propositional calculus that have substantially larger learnable classes may be a difficult one.

The positive conclusions of this paper are that there are specific classes of concepts that are learnable in polynomial time using learning protocols of the kind described. These classes can all be characterized by defining the class of programs that recognize them. In each case the programs are special kinds of Boolean expressions. The three classes are: (i) conjunctive normal form expressions with a bounded number of literals in each clause, (ii) monotone disjunctive normal form expressions, and (iii) arbitrary expressions in which each variable occurs just once. In the first of these no calls of the oracle are necessary. In the last no access to typical examples is necessary but the oracles need to be more powerful than the one described above.

The deduction procedure will in each case output an expression that with high likelihood closely approximates the expression to be learnt. Such an approximate expression never says yes when it should not, but may say no on a small fraction of the probability space of positive examples. This fraction can be made arbitrarily small by increasing the runtime of the deduction procedure. Perhaps the main technical discovery contained in the paper is that with this probabilistic notion of learning highly convergent learning is possible for whole classes of Boolean functions. This appears to distinguish this approach from more traditional ones where learning is seen as a process of "inducing" some general rule from information that is insufficient for a reliable deduction to be made. The power of this probabilistic viewpoint is illustrated, for example, by the fact that an arbitrary set of polynomially many positive examples cannot be relied on to determine expressions consisting of even just a single monomial in any reliable way.

There is another aspect of our formulation that is worth emphasizing. In a learning system that is about to learn a new concept there may be an enormous number of propositional variables available. These may be primitive inputs, the values of preprogrammed concepts, or the values of concepts that have been learnt previously. We want the complexity of learning the new concept to be related only to the number of variables that may be set in natural examples of it, and not on the cardinality of the universe of available variables. Hence the questions asked of ORACLE and the values given by EXAMPLES will not be truth assignments to all the variables. In the archetypal case they will specify an assignment to a subset of the variables that is still sufficient to guarantee the truth of the function.

Whether the classes of learnable Boolean concepts can be extended significantly beyond the three classes given is an interesting question. There is circumstantial evidence from cryptography, however, that the whole class of functions computable by polynomial size circuits is not learnable. Consider a cryptographic scheme that encodes messages by evaluating the function E_k where k specifies the key. Suppose that this scheme is immune to chosen plaintext attack in the sense that even if the values of E_k are known for polynomially many different inputs, it is computationally infeasible to deduce an algorithm for E_k or for an approximation to it. This is equivalent to saying, however, that E_k is not learnable. The conjectured existence of good cryptographic functions that are easy to compute therefore implies that some easy to compute functions are not learnable.

If the class of learnable concepts is as severely limited as suggested by our results then it would follow that the only way of teaching more complicated concepts is to build them up from such simple ones. Thus a good teacher would have to identify, name and sequence these intermediate concepts in the manner of a programmer. The results of *learnability theory* would then indicate the maximum granularity of the single concepts that can be acquired without programming.

In summary, this paper attempts to explore the limits of what is learnable as allowed by algorithmic complexity. The results are distinguishable from the diverse body of previous work on learning because they attempt to reconcile the three properties (A)-(C) mentioned earlier. Closest in rigour to our approach is the inductive inference literature (see Angluin and Smith [1] for a survey) that deals with inducing such things as recursive functions or formal grammars (but not Boolean functions) from examples. There is a large body of work on pattern recognition and classification, using statistical and other tools, (e.g. [4]) but the question of general knowledge representation is not addressed there directly. Learning, in various less formal senses, has been studied widely as a branch of artificial intelligence. A survey and bibliography can be found in [2,6]. In their terminology the subject of this paper is concept learning.

2. A LEARNING PROTOCOL FOR BOOLEAN FUNCTIONS

We consider t Boolean variables $p_1,\ldots,p_t$ each of which can take the value 1 or 0 to indicate whether the associated proposition is true or false. There is no assumption about the independence of the variables. Indeed, they may be functions of each other.

A *vector* is an assignment to each of the t variables of a value from $\{0,1,*\}$. The symbol $*$ denotes that a variable is *undetermined*. A vector is *total* if every variable is *determined* (i.e. is assigned a value from $\{0,1\}$.) For example, the assignment $p_1=p_3=1$, $p_4=0$ and $p_2=*$ is a vector that is not total.

A Boolean function F is a mapping from the set of 2^t total vectors to $\{0,1\}$. A Boolean function F becomes a *concept* F if its domain is extended to the set of all vectors as follows: For a vector v $F(v)=1$ if and only if $F(w)=1$ for all total vectors w that agree with v on all variables for which v is determined. The purpose of this extension is that it permits us not to mention the variables on which F does not depend.

Given a concept F we consider an *arbitrary* probability distribution D over the set of all vectors v such that $F(v)=1$. In other words for each v such that $F(v)=1$ it is the case that $D(v)\geq 0$. Also $\Sigma D(v)=1$ when summation is over this set of vectors. There are no other assumed restrictions on D, which is intended to describe the relative frequency with which the positive examples of F occur in nature.

What are reasonable learning protocols to consider? First we must avoid giving the teacher too much power, namely the power to communicate a program instruction by instruction. For example, if a premeditated sequence of vectors with repetitions could be communicated then this could be used to encode the description of the program even if just two such vectors were used for binary notation. Secondly we must avoid giving the teacher what is evidently too little power. In particular, the protocol must provide some typical examples of vectors for which F is true, for otherwise, if F is true for just one vector which is total, only an exponential search or more powerful oracles would be able to find it.

Such considerations led us to consider the following learning protocol as a reasonable one. It gives the learner access to the following two routines:

(i) EXAMPLES: This has no input. It gives as output a vector v such that $F(v)=1$. For each such v the probability that v is output on any single call is $D(v)$.

(ii) ORACLE( ): On vector v as input it outputs 1 or 0 according to whether $F(v)=1$ or 0.

The first of these gives randomly chosen positive examples of the concept being learnt. The second provides a test for whether a vector which the deduction procedure considers to be critical is an example of the concept or not. In a real system the oracle may be a human expert, a data base of past observations, some deduction system, or a combination of these.

Finally we observe that our particular choice of learning protocol was strongly influenced by the earlier decision to deal with concepts rather than raw Boolean functions. If we were to deal with the latter then many other alternatives are equally natural. For example, given a vector v instead of asking, as we do, whether all completions of it makes the function true, we could ask whether there exist any completions that make it true. This suggests natural alternative semantics for EXAMPLES and ORACLE. In Section 7 we shall use such more elaborate oracles.

3. LEARNABILITY

We shall consider various classes of programs having $p_1,\ldots,p_t$ as inputs and show that in each case any program in the class can be deduced, with only small probability of error, in polynomial time using the protocol previously described. We assume that programs can take the values 1, 0 and undetermined.

More precisely we say that a class X of programs is *learnable* with respect to a given learning protocol if and only if there exists an algorithm A (the deduction procedure) invoking the protocol with the following properties:

(a) The algorithm runs in time polynomial both in an adjustable parameter h and in the various parameters that quantify the size of the program to be learnt, and

(b) For all programs $f\in X$ and all distributions D over vectors v on which f outputs 1 the algorithm will deduce with probability at least $(1-h^{-1})$ a program $g\in X$ that never outputs one when it should not, but outputs one almost always when it should. In particular (i) for all vectors v $g(v)=1$ implies $f(v)=1$, and (ii) the sum of $D(v)$ over all v such that $f(v)=1$ but $g(v)\neq 1$ is at most h^{-1}.

In our definition we have disallowed positive answers from g to be wrong, only because the particular classes X in this paper do not require it. This we call *one-sided error* learning. In other circumstances it may be efficacious to allow two-sided errors. In that more general case we would need to put a probability distribution $\bar{D}$ on the set of vectors such that $f(v) \neq 1$. Condition (i) in (b) is then replaced by: $\Sigma \bar{D}(v)$ over all v such that $f(v) \neq 1$ but $g(v) = 1$ is at most h^{-1}.

A second aspect of our definition is that the parameter h is used in two independent probabilistic bounds. This simplifies the statement of the results. It would be more precise, however, to work explicitly with two independent parameters h_1 and h_2.

Thirdly, we should note that programs that compute concepts should be distinguished from those that compute merely the Boolean function. This distinction has little consequence for the three classes of expressions that we consider since in these cases programs for the concepts follow directly from the specification of the expressions. For the sake of generality our definitions do allow the value of a program to be undefined since a non-total vector will not, in general, determine the value of an expressions as 0 or 1.

It would be interesting to obtain negative results other than the cryptographic evidence mentioned in the introduction, indicating that the class of unrestricted Boolean circuits is not learnable. In the contrary direction the reader can verify that the difficulty of this problem can be upper bounded in various ways. It can be shown that the assumption that P equals NP would imply that the class of Boolean circuits is **learnable** with two sided error with respect to the protocol that provides random examples from both D and $\bar{D}$.

4. A COMBINATORIAL BOUND

The probabilistic analysis needed for our **later** results can be abstracted into a single lemma. The proof of the lemma is independent of the rest of the paper.

We define the function L(h,S) for any **real** number h greater than one and any positive integer S as follows: Let L(h,S) be the **smallest** integer such that in L(h,S) independent Bernoulli trials each with probability h of success, the probability of having fewer than S successes is less than h^{-1}.

The following simple upper bound holds for the whole range of values of S and h and shows that L(h,S) is essentially linear both in h and in S.

PROPOSITION: *For all integers* $S \geq 1$ *and all* **real** $h > 1$.

$$L(h,S) \leq 2h(S + \log_e h) .$$

Proof. We use the following three well known inequalities of which the last is due to Chernoff (see [5], p. 18).

(a) For **all** $x > 0$ $(1 + x^{-1})^x < e$.

(b) For **all** $x > 0$ $(1 - x^{-1})^x < e^{-1}$.

(c) In m independent trials each with probability p of success the probability that there are at most k success, where $k < mp$, is at most

$$\left(\frac{m-mp}{m-k}\right)^{m-k} \left(\frac{mp}{k}\right)^{k} .$$

The first factor in the expression in (c) above can be rewritten as

$$\left(1 - \frac{mp-k}{m-k}\right)^{((m-k)/(mp-k))(mp-k)} .$$

using (b) with $x = (m-k)/(mp-k)$ we can upper bound the product by

$$e^{-mp+k} (mp/k)^k .$$

Substituting $m = 2h(S+\log_e h)$, $p = h^{-1}$ and $k=S$ and using logarithms to the base e gives the bound

$$e^{-2S - 2\log h + S} \cdot [2(1+(\log h)/S)]^{(S/\log h)\log h} .$$

Rewriting this using (a) with $x = (\log_e h)/S$ gives

$$e^{-S - 2\log h} \cdot 2^S e^{\log h} \leq (2/e)^S \cdot e^{-\log h} \leq h^{-1}. \quad \square$$

As an illustration of an application suppose that we have access to EXAMPLES, a source of natural positive examples of vectors for concept F. Suppose that we want to find an approximation to the subset P of **variables** that are determined in at least one natural example of F. Consider the following procedure. Pick the first $L(h,|P|)$ vectors provided by EXAMPLES and let P' be the union of the sets of variables determined by these vectors. The proposition then implies that with probability at least $1 - h^{-1}$ the set P' will have the following property: if a vector is chosen with probability distribution D then the probability that it determines a variable in P -P' is less than h^{-1}. To see this observe that if P' does not have the claimed property then the following has happened: $L(h,|P|)$ **trials** have been made (i.e. calls of EXAMPLES) each with probability greater than h^{-1} of success (i.e. finding a vector that determines a variable in P-P') but there have been fewer than $|P|$ successes (i.e. discoveries of new additions to P'). The probability of such a happening is less than h^{-1} by the definition of L.

The above application shows that the set of **variables that are** determined in natural examples of F can be approximated by a procedure whose runtime is independent of **t**, the **total** number of **variables.**

5. BOUNDED CNF EXPRESSIONS

A *conjunctive normal form* (CNF) expression is any product $c_1c_2\ldots c_r$ of clauses where each clause c_i is a sum $q_1+q_2+\ldots+q_{j_i}$ of literals. A *literal* q is either a variable p or the negation $\bar{p}$ of a variable. For example, $(p_1+\bar{p}_2)(\bar{p}_1+\bar{p}_2+p_3)$ is a CNF expression. For a positive integer k a *k-CNF expression* is a CNF expression where each clause is the sum of at most k literals.

For CNF expressions in general we do not know of any polynomial time deduction procedure with respect to our learning protocol. The question of whether one exists is tantalizing because of its apparent simplicity.

In this section we shall prove that for any fixed integer k the class of k-CNF expressions is learnable. In fact it is learnable easily in the sense that calls of EXAMPLES suffice and no calls of ORACLE are required.

In this and subsequent sections we shall use the relation $\Rightarrow$ on concepts as follows: $F\Rightarrow G$ means that for all vectors v whenever $F(v)=1$ it is also the case that $G(v)=1$. (N.B. This is equivalent to the relation $F\Rightarrow G$ when F,G are regarded as functions.) For brevity we shall often denote both expressions and even vectors by the concepts they represent. Thus the concept represented by vector w is true for exactly those vectors that agree with w on all variables on which w is determined. The concept represented by expression f is obtained by considering the Boolean function computed by f and extending its domain to become a concept. In this section we regard a clause c_i to be a special kind of expression. In Sections 7 and 8 we consider monomials m, simple products of literals, as special forms of expressions. In all cases statements of the form $f\Rightarrow g$, $v\Rightarrow c_i$, $v\Rightarrow m$ or $m\Rightarrow f$ can be understood by interpreting the two sides as concepts or functions in the obvious way.

THEOREM A: *For any positive integer* k *the class of* k-CNF *expressions is learnable via an algorithm* A *that uses* $L=L(h,(2t)^{k+1})$ *calls of EXAMPLES and no calls of ORACLE, where* t *is the number of variables.*

Proof. The algorithm is initialized with formula g as the product of all possible clauses of up to k literals from $\{p_1,\bar{p}_1,p_2,\bar{p}_2,\ldots,p_t,\bar{p}_t\}$. Clearly the number of ways of choosing a clause of exactly k literals is at most $(2t)^k$ and hence the number of ways of choosing a clause with up to k literals is less than $2t+(2t)^2+\ldots+(2t)^k<(2t)^{k+1}$. This bounds the initial number of clauses in g.

The algorithm then calls EXAMPLES L times to give positive examples of the concept represented by k-CNF formula f. For each vector v so obtained it deletes all of the remaining clauses in g that do not contain a literal that is determined to be true in v. (I.e. in the clauses deleted each literal is either not determined or is negated in v.) More precisely the following is repeated L times:

```
begin v := EXAMPLES
      for each c_i in g delete c_i if v ⇏ c_i.
end
```

The claim is that with high probability the value of g after the L-th execution of this block will be the desired approximation to the formula f that is being learnt.

Initially the set of vectors $\{v|v\Rightarrow g\}$ is clearly empty. We first claim that as the algorithm proceeds the set $\{v|v\Rightarrow g\}$ will always be a subset of $\{v|v\Rightarrow f\}$. To prove this it is clearly sufficient to prove the same statement when both sets are restricted to only total vectors. Let B be the product of all the clauses c containing up to k literals with the property that "$\forall v$ if $v\Rightarrow f$ then $v\Rightarrow c$." It is clear that in the course of the algorithm no clause in B can be ever deleted. Hence it is certainly true that $\{v|v\Rightarrow g\}\subseteq\{v|v\Rightarrow B\}$. To establish the claim it remains only to prove that B computes the same Boolean function as f. (In fact B will be the maximal k-CNF formula equivalent to f.) It is easy to see that $f\Rightarrow B$ since by the definition of B, for every c in B it is the case that "for all v if $v\Rightarrow f$ then $v\Rightarrow c$." To verify the converse, that $B\Rightarrow f$, we first note that, by definition, f is a k-CNF formula. If some clause in f did not occur in B then this clause c' would have the property that "$\exists v$ such that $v\Rightarrow f$ but $v\not\Rightarrow c'$." But this is impossible since if c' is a clause of f and if $v\not\Rightarrow c'$ then $v\not\Rightarrow f$. We conclude that every k-CNF representation of the function that f represents consists of a set of clauses that is a subset of the clauses in B. Hence $B\Rightarrow f$.

Let $X=\Sigma D(v)$ with summation over all (not necessarily total) vectors v such that $v\Rightarrow f$ but $v\not\Rightarrow g$. This quantity is defined for each intermediate value of g in the course of the algorithm and, as is easily verified, it is monotone decreasing with time. Now clauses will be removed from g whenever EXAMPLES outputs a vector v such that $v\not\Rightarrow g$. The probability of this happening at any moment is exactly the current value of X. Also, the process of running the algorithm to completion can have one of just two possible outcomes: (i) At some point X becomes less than h^{-1}, in which case the g found will approximate to f exactly as required by the definition of learnability. (ii) The value of X never goes below h^{-1} and hence g is not an acceptable approximation. The probability of the latter eventuality is, however, at most h^{-1} since it corresponds to the situation of performing $L(h,(2t)^{k+1})$ Bernoulli experiments (i.e. calls of EXAMPLES) each with probability greater than h^{-1} of success (i.e. finding a v such that $v\not\Rightarrow g$) and obtaining fewer than $(2t)^{k+1}$ successes (a success being manifested by the removal of at least one clause from g). □

In conclusion we observe that a CNF expression g immediately yields a program for computing the associated concept. Given a vector v we substitute the determined truth values in g. The concept will be true for v if and only if all the clauses are made true by the substitution.

6. DNF EXPRESSIONS

A *disjunctive normal form* (DNF) expression is any sum $m_1 + m_2 + \ldots + m_r$ of monomials where each monomial m_i is a product of literals. For example $p_1\bar{p}_2 + p_1p_3\bar{p}_4$ is a DNF expression. Such expressions appear particularly easy for humans to comprehend. Hence we expect that any practical learning system would have to allow for them.

An expression is *monotone* if no variable is negated in it. We shall show that for monotone DNF expressions there exists a simple deduction procedure that uses both EXAMPLES and ORACLE. For unrestricted DNF a similar result can be proved with respect to a different size measure. In the unrestricted case there is the additional difficulty that we can guarantee to deduce a program only for the function, and not for the concept. This difficulty does not arise in the monotone case where given an expression we can always compute the value of the associated concept for a vector v by making the substitution and asking whether the resulting expression is identically true.

A monomial m is a *prime implicant* of a function F (or of an expression representing the function) if $m \Rightarrow F$ and if $m' \not\Rightarrow F$ for any m' obtained by deleting one literal from m. A DNF expression is *prime* if it consists of the sum of prime implicants none of which is redundant in the sense of being implied by the sum of the others. There is always a unique prime DNF expression in the monotone case, but not in general. We therefore define the *degree* of a DNF expression to be the largest number of monomials that a prime DNF expression equivalent to it can have. The unique prime DNF expression in the monotone case is simply the sum of all the prime implicants.

THEOREM B: *The class of monotone DNF expressions is learnable via an algorithm B that uses* $L = L(h,d)$ *calls of EXAMPLES and* dt *calls of ORACLE, where* d *is the degree of the DNF expression* f *to be learnt and* t *the number of variables.*

Proof. The algorithm is initialized with formula g identically equal to the constant zero. The algorithm then calls EXAMPLES L times to produce positive examples of f. Each time a vector v is produced such that $v \not\Rightarrow g$ a new monomial m is added to g. The monomial m is the product of those literals determined in v that are essential to make $v \Rightarrow f$. More precisely, the loop that is executed L times is the following:

```
begin  v := EXAMPLES
       if  v ⇏ g  then
       begin for  i := 1  to  t  do
             if  p_i  is determined in  v  then
             begin  set ṽ equal to v but with p_i := *;
                    if ORACLE(ṽ) = 1 then v := ṽ
             end
             set m equal to the product of all
                    literals q such that  v ⇒ q;
             g := g + m
       end
end
```

The test $v \not\Rightarrow g$ amounts to asking whether none of the monomials of g is made true by the values determined to be true in v. Every time EXAMPLES produces a value v such that $v \not\Rightarrow g$ the inner loop of the algorithm will find a prime implicant m to add to g. Each such m is different from any previously added (since the contrary would have implied that $v \Rightarrow g$). It follows that such a v will be found at most d times and hence ORACLE will be called at most dt times.

Let $X = \Sigma D(w)$ with summation over all (not necessarily total) vectors w such that $w \Rightarrow f$ but $w \not\Rightarrow g$. This quantity is defined for each intermediate value of g in the course of the algorithm, is initially unity, and decreases monotonically with time. Now a monomial will be added to g each time EXAMPLES outputs a vector v such that $v \not\Rightarrow g$. The probability of this occurring at any call of EXAMPLES is exactly X. The process of running the algorithm to completion can have just two possible outcomes: (i) At some time X has become less than h^{-1}, in which case the final expression g found will approximate to f as required by the definition of learnability. (ii) The value of X never goes below h^{-1} and hence g is not an acceptable approximation. The probability of this second eventuality is, however, at most h^{-1} since it corresponds to the situation of performing $L(h,d)$ Bernoulli experiments (i.e. calls of EXAMPLES) each with probability greater than h^{-1} of success (i.e. of finding a v such that $v \Rightarrow f$ but $v \not\Rightarrow g$) and obtaining fewer than d successes (each manifested by the addition of a new monomial). □

For unrestricted DNF expressions several problems arise. The main source of difficulty is the fact that the problem of determining whether a nontotal vector implies the function specified by a DNF formula is NP-hard. This is simply because the problem of determining whether the nowhere determined vector implies the function is the tautology question of Cook [3]. An immediate consequence of this is that it is unreasonable to assume that the program being learnt computes concepts rather than functions. Another consequence is that in any algorithm akin to Algorithm B the test $v \not\Rightarrow g$ may not be feasible if v is not total. Algorithm B is sufficient, however, to establish the following:

THEOREM B': *Suppose the notion of learnability is restricted to distributions* D *such that* $D(v) = 0$ *whenever* v *is not total. Then the class of DNF expressions is learnable, in the sense that programs for the functions (but not necessarily the concepts) can be deduced in* $L(h,d)$ *calls of EXAMPLES and* dt *calls of ORACLE where* d *is the degree of the DNF expression to be learnt.*

The reader should note that here there is the additional philosophical difficulty that availability of the expression does not in itself imply a polynomial time algorithm for ORACLE. On the other hand, the theorem does say that if an agent has some, may be *ad hoc*, black box for ORACLE whose workings are unknown to him, he can use it to teach

someone else a DNF expression that approximates the function.

Finally, it may be worth emphasizing that the monotone case is nonproblematic and the deduction procedure for it very simple-minded. It may be interesting to pursue more sophisticated deduction procedures for it if contexts can be found in which they can be proved advantageous. The question as to whether monotone DNF expressions can be learnt from EXAMPLES alone is open. A positive answer would be especially significant.

7. μ-EXPRESSIONS

We have already seen a class of expressions, namely the k-CNF expressions, that can be learnt from positive examples alone. Here we shall consider the other extreme, a class that can be learnt using oracles alone.

Deducing expressions on which there are less severe structural restrictions than DNF or CNF appears much more difficult. The aim of this section is to give a paradigmatic example of how far one can go in that direction if one is willing to pay the price of oracles that are more sophisticated than the one previously described.

A general *expression* over variable set $\{p_1,\ldots,p_t\}$ is defined recursively as follows:

(i) For each i $(1 \leqslant i \leqslant t)$ "p_i" and "$\bar{p}_i$" are expressions.

(ii) If $f_1,\ldots,f_r$ are expressions then "$(f_1 + f_2 + \ldots + f_r)$" is an expression (called a plus expression).

(iii) If $f_1,\ldots,f_r$ are expressions then "$(f_1 \times f_2 \times \ldots \times f_r)$" is an expression (called a times expression).

A μ-*expression* is an expression in which each variable appears at most once. We can assume that a μ-expression is monotone since we can always relabel negated variables with new names that denote their negation. In the recursive definition of any μ-expression there are clearly at most $2t-1$ intermediate expressions of which at most t are of type (i) and at most $t-1$ of types (ii) or (iii). Without loss of generality we shall assume that in the definition of a μ-expression rules (ii) and (iii) alternate. We shall regard two μ-expressions as identical if their definitions can be made formally the same by reordering sums and products and relabelling as necessary.

For learning μ-expressions we shall employ more powerful oracles. The boundary between reasonable and unreasonable oracles does not appear sharp. We make no claims about the reasonableness of these new oracles except that they may serve as vehicles for understanding learnability.

The definitions refer to the Boolean function F (not regarded as a concept here). The oracle of Section 2 will be renamed N-ORACLE since it is one of *necessity*: N-ORACLE$(v)=1$ if and only if for all total vectors w such that $w \Rightarrow v$ it is the case that $F(w)=1$. The dual of this would be a *possibility oracle*: P-ORACLE$(v)=1$ if and only if there exists a total vector w such that $w \Rightarrow v$ and $F(w)=1$. For brevity we shall also define the *prime implicant* oracle: PI$(v)=1$ if and only if N-ORACLE$(v)=1$ but N-ORACLE$(w)=0$ for any w obtained from v by making one variable determined in v undetermined.

Finally we define two further oracles, ones of *relevant possibility* RP and of *relevant accompaniment* RA. For convenience we define the first by how it behaves on vectors represented as monomials: RP$(m)=1$ iff from some monomial m' mm' is a prime implicant of f. For sets, V,W of variables we define RA$(V,W)=1$ iff every prime implicant of f that contains a variable from V also contains a variable from W.

THEOREM C: *The class of* μ-*expressions is learnable via a deduction procedure* C *that uses* $O(t^3)$ *calls of* N-ORACLE, RP *and* AP *altogether, where* t *is the number of variables, (and no calls of* EXAMPLES.*) The procedure always deduces exactly the correct expression.*

Proof. Let λ be the monomial that is the product of no literals. The algorithm will first compute RP(p_i) for each p_i to determine which of the variables occur in the prime implicants of the function that the expression f to be learnt represents. Let $g_1,\ldots,g_r$ be distinct single variable expressions, one for each p_i for which RP$(p_i)=1$. Note that these are exactly the variables that occur in f since f is monotone.

With each g_i we associate two monomials m_i and $\hat{m}_i$. The former will be the single variable p_j that g_i represents. The latter is defined as any monomial $\hat{m}$ having no variable in common with m_i such that $m_i\hat{m}$ is a prime implicant of f. The algorithm will construct each such $\hat{m}_i$ as the final value of m in the following procedure: Set $m=\lambda$; while PI$(mm_i)=0$ find a p_k not in mm_i such that RP$(p_k mm_i)=1$ and set $m := p_k m$. Hence in at most t^2 calls of PI (i.e. t^3 calls of N-ORACLE) and t^2 calls of RP values for every $\hat{m}_i$ will be found.

Once initialized the algorithm proceeds by alternately executing a *plus-phase* and a *times-phase*. At the start of each phase we have a set of expressions $g_1,\ldots,g_r$ where each g_i is associated with two monomials m_i and $\hat{m}_i$ (having no variables in common) where m_i is a prime implicant of g_i and $m_i\hat{m}_i$ is a prime implicant of f. We distinguish the g's as *plus* or *times expressions* according to whether the outermost construction rule was addition or multiplication. A plus-phase will first compute an equivalence relation S on the subset of $\{g_i\}$ that are times expressions. For each equivalence class G such that no $g_i \in G$ already occurs as a summand in a sum, we construct a new expression that is the sum of the members of G and call this sum g_k where k is a previously unused index. If some members of G already occur in a sum, say g_j, (N.B. they are never distributed in more than one sum) then we modify the sum expression

g_j to equal the sum of every expression in G. A times phase is exactly analogous except that it computes a different equivalence relation T, now on plus expressions, and will form new, or extend old, times expressions. In the above context single variable expressions will be regarded as both times and plus expressions. Also, it is immaterial which of the two kinds of phase is used to start the algorithm.

The intention of the algorithm is best expressed by the claim to follow which will be verified later. Suppose that the expressions occurring in the definition of f are $f_1,\ldots,f_q$ (where $q \leq 2t-1$). We shall say that $g \leq f_i$ iff the set of prime implicants of g is a subset of the set of prime implicants of $\tilde{f}_i$ where (i) if f_i is a plus expression then $\tilde{f}_i = f_i$ and (ii) if f_i is a times expression then $\tilde{f}_i$ is the product of some subset of the multiplicands of f_i.

Claim 1: After every phase of the algorithm for every g_i that has been constructed there is exactly one expression f_i such that (i) $g_i \leq f_i$ and (ii) f_i is of the same kind (i.e. plus or times) as g_i.

The procedure builds up the rooted tree of the expression as rooted subtrees starting from the leaves. One evident difficulty is that there is no *a priori* knowledge available about the shape of the tree. Hence in the grafting process a subtree may become attached to another at just about any node.

Whenever a plus or times expression g_i is created or enlarged its associated m_i and $\hat{m}_i$ is updated as follows. If g_i is a sum then we let $m_i = m_j$ and $\hat{m}_i = \hat{m}_j$ for any g_j that is a summand in g_i. If g_i is a product then m_i will be the product of all the m_j's that correspond to multiplicands in g_i. Finally $\hat{m}_i$ will be generated as the final value of m in the following procedure: Set $m := \lambda$; while $PI(mm_i) = 0$ find a p_k not in mm_i such that $RP(p_k mm_i) = 1$ and set $m := p_k m$. Since such an $\hat{m}_i$ has to be found at most t times in the overall algorithm the total cost is at most t^2 calls of PI (i.e., t^3 calls of N-ORACLE) and t^2 calls of RP.

In order to complete the description of the algorithm it remains only to define the equivalence relations S and T.

Definition: $g_i \, S \, g_j$ if and only if

(i) $PI(m_i\hat{m}_j) = PI(m_j\hat{m}_i) = 1$, and

(ii) $m_i, \hat{m}_j$ contains disjoint sets of variables, as do $m_j, \hat{m}_i$.

For defining T we shall denote by V_i the set of variables that occur in the expression g_i.

Definition: $g_i \, T \, g_j$ if and only if $RA(V_i,V_j) = RA(V_j,V_i) = 1$.

First we shall verify that S and T are indeed equivalence relations under the assumption that Claim 1 holds at the beginning of each phase. Clearly S and T are defined to be both reflexive and symmetric. Also T is transitive for supoose that $g_i \, T g_j$ and $g_j \, T g_k$. The former implies that every prime implicant of f containing some variable from g_i also contains some variable from g_j. The latter implies that every prime implicant of f containing some variable from g_j also contains a variable from g_k. Hence $g_i \, T g_k$ follows. In order to verify the transitivity of S we shall make a more general observation about any sub-expression of f.

Claim 2: If m_i, m_j are prime implicants of distinct times subexpressions f_i, f_j of f and if for some m, with no variable in common with either m_i or m_j, both $m_i m$ and $m_j m$ are prime implicants of f, then f_i, f_j must occur as summands in the same plus expression of f.

Proof. Most easily verified by representing expression f as a directed graph (e.g. [7]) with edges labelled by the Boolean variables. The sets of labels along directed paths between a distinguished source node and a distinguished sink node correspond exactly to prime implicants in the case of μ-expressions where all edges have different labels. □

Now to verify that S is transitive suppose that $g_i \, S g_j$ and $g_j \, S g_k$ where, by Claim 1, $g_i \leq f_i$, $g_j \leq f_j$ and $g_k \leq f_k$ for appropriate times expressions f_i, f_j, f_k. Then it follows that $m_i\hat{m}_j$, $m_j\hat{m}_j$ and $m_k\hat{m}_j$ are all prime implicants of f where m_i, m_j, m_k have no variable in common with $\hat{m}_j$. It follows from Claim 2 that f_i, f_j and f_k are addends in the same sum expression. Hence $g_i \, S g_k$ follows.

Claim 3: Suppose that after some phase of the algorithm Claim 1 holds and times expressions g_i, g_j have been formed where $g_i \leq f_i$, $g_j \leq f_j$ and f_i, f_j are times expressions. Then g_i and g_j will be placed in the same sum expression at the next plus phase if and only if $f_i = \tilde{f}_i$, $f_j = \tilde{f}_j$ and f_i, f_j are addends in the same sum expression in f.

Proof. ($\Rightarrow$) If $g_i \leq f_i = \tilde{f}_i$ and $g_j \leq f_j = \tilde{f}_j$ and f_i, f_j are in the same sum then m_i, m_j will be prime implicants of f_i, f_j. Also $m_i\hat{m}_j$ and $m_j\hat{m}_i$ will be prime implicants of f, and the variables in m_i will be disjoint from those in $\hat{m}_j$ as will be those in m_j from those in $\hat{m}_i$. Hence $g_i \, S g_j$ will hold and g_i, g_j will be in the same plus expression after the next plus phase.

($\Leftarrow$) Suppose that $g_i \leq f_i$, $g_j \leq f_j$ and $g_i \, S g_j$ holds. Then m_i, m_j will be prime implicants

of g_i, g_j respectively, $m_i \hat{m}_j, m_j \hat{m}_j$ will both be prime implicants of f, and the variables in $\hat{m}_j$ will be disjoint from those of both m_i and m_j. It follows from Claim 2 that $f_i = \tilde{f}_i$ and $f_j = \tilde{f}_j$ must occur as summands in the same plus expression of f. [N.B. Here we are using the fact that Claim 2 remains true if we allow a "times subexpression" to be the product of any subset of the multiplicands in a times expression in f.]

Claim 4: Suppose that after some phase of the algorithm Claim 1 holds and plus expressions $g_i \leqslant f_i$ and $g_j \leqslant f_j$ (f_i, f_j plus expressions) have been found. (i) If $g_i = f_i$ and $g_j = f_j$ and f_i, f_j are multiplicands in the same times expressions in f then g_i, g_j will be placed in the same times expression after the next times phase. (ii) If g_i, g_j are placed in the same times expression at any subsequent phase then f_i, f_j are in the same product in f.

Proof. (i) If the conditions of (i) hold then $g_i T g_j$ will be discovered at the next times phase and the claim follows. (ii) If f_i, f_j are not in the same product in f then f contains some prime implicant containing variables from one of g_i, g_j and not from the other. Hence $g_i T g_j$ will never hold. □

Proof of Claim 1. By induction on the number of phases on Claims 1, 3 and 4 simultaneously. □

We define the *depth* of a formula f_i to be the maximum number of alternations of sum and product required in its definition.

Claim 5: If f_i is an expression of depth k in f then after k phases a g_i identical to f_i will have been constructed by the algorithm.

Proof. By induction on Claims 3 and 4. □

To conclude the proof of the theorem it remains to analyze the runtime. This is dominated by the cost of computing m_i and $\hat{m}_i$, for which we have already accounted, plus the cost of computing the equivalence relations S and T. We first note that fewer than 2t expression names g_i are used in the course of the algorithm. When an expression is grafted into another at a point deep in the latter then the semantics of all the subexpressions above will change. By Claims 3 and 4 such grafting can occur when a g_i is added to a sum expression but not when added to a times expression. Hence the values m_i and $\hat{m}_i$ do not need to be changed for any expression when such grafting occurs. It follows that for computing S only 2t values of $m_i, \hat{m}_i$ need to be considered. Hence $O(t^2)$ calls of PI or $O(t^3)$ calls of N-ORACLE suffice overall. For computing T grafting may cause a ripple effect. On each occasion the value of V_j may have to be updated for up to t such sets and hence t^2 calls of RA will suffice. Hence $O(t^3)$ calls of RA in the overall algrorithm will be enough. □

8. REMARKS

In this paper we have considered learning as the process of deducing a program for performing a task, from information that does not provide an explicit description of such a program. We have given precise meaning to this notion of learning and have shown that in some restricted but nontrivial contexts it is computationally feasible.

Consider a world containing robots and elephants. Suppose that one of the robots has discovered a recognition algorithm for elephants that can be meaningfully expressed in k-conjunctive normal form. Our Theorem A implies that this robot can communicate its algorithm to the rest of the robot population by simply exclaiming "elephant" whenever one appears.

An important aspect of our approach, if cast in its greatest generality, is that we require the recognition algorithms of the teacher and learner to agree on an overwhelming fraction of only the natural inputs. Their behavior on unnatural inputs is irrelevant and hence descriptions of all possible worlds are not necessary. If followed to its conclusion this idea has considerable philosophical implications: A learnable concept is nothing more than a short program that distinguishes some natural inputs from some others. If such a concept is passed on among a population in a distributed manner substantial variations in meaning may arise. More importantly, what consensus there is will only be meaningful for natural inputs. The behavior of an individual's program for unnatural inputs has no relevance. Hence thought experiments and logical arguments involving unnatural hypothetical situations may be meaningless activities.

The second important aspects of the formulation is that the notion of oracles makes it possible to discuss a whole range of teacher-learner interactions beyond the mere identification of examples. This is significant in the context of artificial intelligence where a human may be willing to go to great lengths to convey his skills to a machine while being unable to articulate the algorithms he himself uses in the practice of the skills. We expect that some explicit programming does become essential for transmitting skills that are beyond certain limits of difficulty. The identification of these limits is a major goal of the line of work proposed in this paper.

9. REFERENCES

[1] D. Angluin and C.H. Smith. "A Survey of Inductive Inference: Theory and Methods." Yale University, Computer Science Department Tech. Report 250, October 1982.

[2] A. Barr and E.A. Feigenbaum. *The Handbook of Artificial Intelligence*, Vol. II, Wm. Kaufmann, Los Altos, Calif. (1982).

[3] S.A. Cook. "The Complexity of Theorem Proving Procedures." *Proc. of Third ACM Symposium on Theory of Computing* (1971), 151-158.

[4] R.O. Duda and P.F. Hart. *Pattern Classification and Scene Analysis*. Wiley, New York (1973).

[5] P. Erdös and J. Spencer. *Probabilistic Methods in Combinatorics*. Academic Press, New York (1974).

[6] R.S. Michalski, J.G. Carbonell and T.M. Mitchell. *Machine Learning: An Artificial Intelligence Approach*, Tioga Publishing Co., Palo Alto, Calif. (1983).

[7] S. Skyum and L.G. Valiant. "A Complexity Theory based on Boolean Algebra." *Proc. of 22nd IEEE Symp. on Foundations of Computer Science*, (1981), 244-253.