Machine Learning, 32, 245–295 (1998)

Learning from Examples and Membership Queries with Structured Determinations

PRASAD TADEPALLI tadepall@cs.orst.edu
Department of Computer Science, Oregon State University, Corvallis, OR 97331

STUART RUSSELL russell@cs.berkeley.edu
Computer Science Division, University of California, Berkeley, CA 94720

Editors: Thomas Dietterich and David Haussler

Abstract. It is well known that prior knowledge or bias can speed up learning, at least in theory. It has proved difficult to make *constructive* use of prior knowledge, so that approximately correct hypotheses can be learned efficiently. In this paper, we consider a particular form of bias which consists of a set of "determinations." A set of attributes is said to *determine* a given attribute if the latter is purely a function of the former. The bias is tree-structured if there is a tree of attributes such that the attribute at any node is determined by its children, where the leaves correspond to input attributes and the root corresponds to the target attribute for the learning problem. The set of allowed functions at each node is called *the basis*. The tree-structured bias restricts the target functions to those representable by a read-once formula (a Boolean formula in which each variable occurs at most once) of a given structure over the basis functions. We show that efficient learning is possible using a given tree-structured bias from random examples and membership queries, provided that the basis class itself is learnable and obeys some mild closure conditions. The algorithm uses a form of *controlled experimentation* in order to learn each part of the overall function, fixing the inputs to the other parts of the function at appropriate values. We present empirical results showing that when a tree-structured bias is available, our method significantly improves upon knowledge-free induction. We also show that there are hard cryptographic limitations to generalizing these positive results to structured determinations in the form of a directed acyclic graph.

Keywords: determinations, tree-structured bias, declarative bias, prior knowledge, read-once formulas, queries, controlled experimentation, pac-learning

1. Introduction

Bias is the term used in machine learning to refer to any information, implicit or explicit, that a learning algorithm uses to select hypotheses, beyond mere consistency with the available observations. Without bias, inductive learning is impossible (Mitchell, 1980). Bias can take many forms, including restrictions on the space of hypotheses that the learning algorithm can consider. Although it is possible for the designer of a learning system to engineer a bias directly into the algorithm's structure, or to specify exogenously various syntactic restrictions on hypotheses, it is desirable that learning programs should, over time, be able to develop their own biases in order to operate autonomously.

The idea of *declarative bias* (Russell & Grosof, 1987, 1990) is to replace the exogenous bias, which is often only implicit, with background domain knowledge possessed by the learning system. Restriction bias, which specifies a restricted hypothesis space, is realized by first-order background knowledge that is logically equivalent to the disjunction of all the hypotheses in the restricted space, whereas preference bias, which specifies the relative likelihoods of different hypotheses in the hypothesis space, is realized by a prioritized

default theory. In many cases, biases can be represented by compact logical theories. Conversely, *any* logical background knowledge is equivalent to a restriction bias that rules out any hypothesis that is inconsistent with the background knowledge. The advantage of such an approach is that such declarative background knowledge can be learned in a variety of ways, including inductive generalization from examples, automatic deduction from the system's other background knowledge, and by being directly told by other agents. As a result, it may be possible to design a learning agent whose cumulative learning performance far exceeds what can be achieved by taking each new learning problem *ab initio*.

Realizing the potential benefits of declarative bias is another matter. One obvious non-solution is to search in an unrestricted hypothesis space, filtering out hypotheses that are found to be inconsistent with the background knowledge. Such an approach is likely to be intractable. Another approach is to use the background knowledge to *re-express* the observations using a set of higher-level attributes, in addition to the original observational attributes, in the hope that it will be simple to find a correct hypothesis in the new space. This approach has proved effective in some cases. Implemented systems of this type include mEBG (Hirsh, 1990) and GOLEM (Muggleton & Feng, 1990). A third approach is to use the background knowledge to generate a "template" for the hypothesis, which can be filled in using the observations to constrain the functions that make up the template. In this way, the system need never consider hypotheses that are not consistent with the background knowledge. Implemented systems of this type include Odysseus (Wilkins, Clancey, & Buchanan, 1987), KBANN (Shavlik & Towell, 1989), and Grendel (Cohen, 1992), as well as the work of Getoor (1989) which we discuss below.

One of the earliest forms of declarative bias was based on the observation that the *vocabulary bias* for a propositional hypothesis space—that is, the set of features from which hypotheses are to be constructed—can be expressed concisely as a determination whose left-hand side contains the relevant features and whose right-hand side contains the target concept (Russell & Grosof, 1987). For example, we say that $\{A, B, C\} \succ D$, read the set $\{A, B, C\}$ *determines* D, if the values of A, B and C are sufficient to determine the value of D. Since determinations are a form of first-order knowledge (Davies, 1985), they can be derived using logical inference from suitable background knowledge, including other determinations. Thus, armed with an appropriate inference algorithm and a knowledge base, a learning system can derive a restricted hypothesis vocabulary for its current learning task. Since this restricted vocabulary may be much smaller than the total set of features available, the learning system may be able to learn from many fewer examples in much less time (Russell, 1989; Mahadevan & Tadepalli, 1994). For example, if the learner knows (or can deduce) that the model, make, optional extras, and year of a car determine its list price, then it need not worry about the color of the car, the day of the week, or the name of the current Pope.

In (Russell, 1988), it was shown that the knowledge used in the derivation of a suitable determination bias can impose additional constraints on the hypothesis space, beyond those implied by the determination itself. We assume that the bias is derived by a backward-chaining proof through a set of determinations, beginning with the target attribute as the root and terminating in a set of leaf attributes all of which are observable in the training data. (Section 2 gives an example of such a derivation. See (Russell, 1989) and (Getoor, 1989) for more examples.) By construction, the attributes at internal nodes of the derivation

tree will be unobservable. If in addition we assume that each attribute occurs only once in the derivation, then the derivation tree forces any hypothesized target function to conform to a *tree-structured bias* (TSB) in which the hypothesis is composed from a set of unknown functions that must combine in a tree structure that mirrors the derivation tree. We call this tree structure along with a labeling of the leaves of the tree with distinct input attributes a *determination tree* (see Figure 1). It was shown in (Russell, 1988) that the number of examples required to learn a function consistent with a given determination tree is linear in the number of leaves, provided the tree has bounded degree. We denote this assumption by k-TSB.[1] Unfortunately, results due to Pitt and Warmuth (1990) show that learning arbitrary Boolean formulas is reducible to learning with a fixed determination tree of bounded degree from classified random examples. Since learning arbitrary Boolean functions is as hard as inverting one-way functions (Kearns & Valiant, 1994), learning with tree-structured bias with random examples alone is believed to be intractable. Getoor's (1989) implementation of tree-structured learning, using a neural network pre-structured according to the bias derivation tree took a prohibitively large training time due to the depth of the neural network, suggesting that this intractability cannot be circumvented easily in practice.

Bshouty, Hancock, and Hellerstein (1992) and Tadepalli (1993) showed that the use of *membership queries* (Angluin, 1988) as well as random examples renders the learning problem with a k-TSB bias tractable. Tadepalli assumes that the determination tree is given, whereas Bshouty et al. infer the tree structure as well as the target function. The key to both of these algorithms is the use of *controlled experimentation*. The idea is to learn each internal function f_i at node i of the tree by finding a suitable setting (or *context*) for the leaf attributes (other than those that affect the inputs of f_i) so that the value of the target concept changes whenever the value of the unobservable internal node i is changed. If such settings are found for any node i and its children, the program can learn the internal function f_i, by varying its inputs and observing its output. When such a setting is not found for a node with a sufficiently large sample and a thorough search, it can, without the risk of a large error, conclude that that node has no effect on the output, and can ignore the subtree under that node.

In this paper, we replace the restriction of bounded degree with the restriction that the functions at the nodes of the tree be drawn from some learnable class called the "basis" which obeys some mild closure properties. We describe a method to convert almost any induction algorithm that learns the basis class to one that learns any function that is consistent with the tree-structured bias where the internal functions are chosen from the basis class. The environment provides random examples, and answers the membership queries, i.e., gives the output for any input queried by the learner. Using the probably approximately correct learning framework (Valiant, 1984), extended with membership queries (Angluin, 1988), we prove that if the determination tree is given, and the internal functions belong to a known learnable basis class with some mild closure conditions on them, then any function consistent with the determination tree can be learned by our algorithm. We present empirical results that show that our algorithm leads to fast and reliable convergence when two different induction programs are used to learn the internal functions: Quinlan's ID3 (Quinlan, 1986), and a program called k-DL that learns k-decision lists introduced by Rivest (1987).

Tree-structured determinations are closely related to read-once formulas or μ-formulas studied in the computational learning theory literature (Angluin, Hellerstein, & Karpinski,

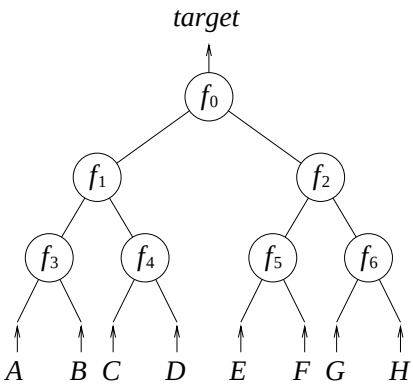

Figure 1. A determination tree that restricts the hypothesis space to functions of the form $f_0(f_1(f_3(A,B),f_4(C,D)),f_2(f_5(E,F),f_6(G,H)))$. The task is to find the functions $f_0, f_1, \ldots$ from examples and membership queries of the composite function.

1993; Hancock & Hellerstein, 1991; Bshouty, Hancock, & Hellerstein, 1992, 1995). In a read-once formula, each variable appears exactly once; hence the formula can be represented by a circuit in the form of a tree, where each leaf is distinctly labeled. This is also the defining property of a tree-structured bias. For example, compare the two formulas $BA + C$ and $BA + \overline{B}C$. While the first formula is read-once, the second is not, and this is reflected in the fact that B is repeated under two leaves in the tree corresponding to the second formula.[2] If we represent the function consistent with a determination tree using the functions $f_1, \ldots, f_n$ that correspond to the internal nodes of the tree as primitive symbols, each input feature appears exactly once in such a formula. The determination tree that corresponds to such a formula is called its "skeleton". Learning a read-once formula over a set of basis functions involves learning its skeleton as well as the internal functions at the nodes of the skeleton. A fair amount of work in this area has been done by investigating what restrictions on the basis functions allows for tractable learning of the skeleton and the internal functions (Angluin et al., 1993; Hancock & Hellerstein, 1991; Bshouty et al., 1992, 1995). This paper takes a different approach. Rather than restricting the basis functions, it considers a particular form of background knowledge, namely the determination tree, that is sufficiently strong to allow learning of read-once formulas for *any* learnable basis. In Section 9, we discuss the relationship of our work to the work on learning read-once formulas, especially to that of (Bshouty et al., 1995).

Our algorithm can be interpreted to mean that the unobservability of the internal attribute values does not pose unsurmountable difficulties in learning functions consistent with a given determination tree. However, using some negative results in computational learning theory, we also show that this positive result does not hold if the tree-structure of determinations is replaced by a directed acyclic graph (dag) structure. In particular, under some widely believed cryptographic assumptions such as the safety of the RSA cryptosystem, we show that it is, in general, not possible to learn functions that obey a given dag-structured bias, when the maximum fan-out (out-degree) of the dag that represents the determinations

is 2 and the maximum fan-in (in-degree) is 3, or vice versa. The negative results hold even when the internal functions are restricted to a very small set of basis functions.

The rest of the paper is organized as follows. Section 2 motivates our work using an example derivation of tree-structured bias. In Section 3, we will describe the pac-learning framework for learning from examples and membership queries. In Section 4, we define tree-structured bias and the associated learning problem. Section 5 introduces the concept of "distinguishing assignments" which is central to our algorithm. The learning algorithm and the proof of correctness are presented in Section 6. Section 7 presents experimental results on our system. Section 8 gives an analysis of dag-structured bias. Section 9 discusses related work, and Section 10 concludes with a summary and future work.

2. Declarative Bias: A Motivating Example

In this section, we motivate the problem of learning a function when a significant amount of prior knowledge is readily available in declarative form.

The example of this section is adapted from Getoor (1989). A credit card company may be interested in predicting the expected profits from offering a credit to an individual. The relevant knowledge base for this domain might contain several facts such as the following, expressed in the form of determinations.

$\{$ Loan, Repayments, InitialSchedule$\} \succ$ ExpectedProfit
$\{$ Responsibility, LiquidAssets, Ability, Commitments $\} \succ$ Repayments
$\{$ InitialBalance, IncomeHistory, MonthlyExpenditure $\} \succ$ LiquidAssets

Using the above knowledge base and an inference engine that can compose determinations, it is possible to construct a derivation tree that shows the functional dependency between the root attribute ExpectedProfit and the primitive input attributes such as Occupation, Education, and Age. Figure 2 shows a determination tree which is based on such a derivation.

The determination tree merely shows the functional dependencies in the domain and by itself is not sufficient to predict the ExpectedProfit from the primitive attributes. But it is an immensely useful bias. In the presence of such a strong bias, and when an oracle can answer queries about a few hypothetical persons, it is possible to learn a function that can make this prediction with only a small number of examples.

In the rest of the paper we show how this might be done when all the attributes are binary and the derivation is tree-structured. The functions at each node of the determination tree should all belong to a learnable class and satisfy some mild closure conditions. On the other hand, it appears that it is necessary for the derivation to be tree-structured in that even mild deviations from the tree-structure could make the learning problem computationally intractable.

3. Learning from Examples and Membership Queries

In this paper, we will be using a version of probably approximately correct (PAC) learning (Valiant, 1984), extended to include membership queries (Angluin, 1988). We restrict ourselves here to learning Boolean functions.

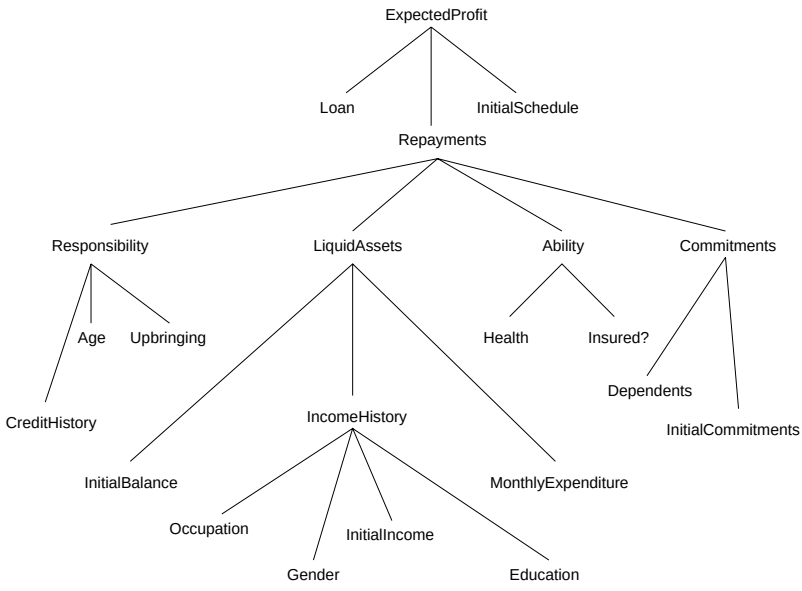

Figure 2. A determination tree derived from a knowledge-base of determinations in the credit card domain. The determination tree provides a strong bias to the learning algorithm.

An *assignment* is a mapping from attributes $B_1, \ldots, B_n$ to their values in $\Sigma = \{0, 1\}$. Σ^n denotes the set of binary strings of length n. Each string in Σ^n represents an assignment, where the i^{th} bit of the string is the value of B_i. Let $\mathcal{F}_n$ be the set of functions from Σ^n to Σ that we are interested in learning. The parameterized sequence $\mathcal{F} = \{\mathcal{F}_1, \ldots, \mathcal{F}_n, \ldots\}$ is called a *function space.* $\mathcal{F}_1, \ldots, \mathcal{F}_n, \ldots$ are called the *subspaces* of $\mathcal{F}$.

We call representations of functions "formulas." Formulas are assumed to be composed of the input attributes $B_1, \ldots, B_n$ as well as some other fixed symbols such as AND, OR, etc. The length of a formula r is the number of symbols in the formula and is denoted by $|r|$. We fix a representation language $\mathcal{R}$ for functions in $\mathcal{F}$. Let $\mathcal{R}_n$ be a set of formulas that represent the functions in $\mathcal{F}_n$. We define the *size* $|f|_{\mathcal{R}}$ of a function $f \in \mathcal{F}_n$ in $\mathcal{R}$ to be the length of a minimum-length formula in $\mathcal{R}_n$ that represents f. We will omit the subscript where no confusion can arise.

We assume an oracle called EXAMPLE that takes no arguments and generates a pair $\langle x, f(x)\rangle$ where x is an assignment in Σ^n chosen according to some fixed, but unknown, probability distribution P, and $f \in \mathcal{F}_n$ is the target function.

Definition 1. Given a target function f, the error of a hypothesis g according to a probability distribution P is the probability that f and g disagree on an instance randomly chosen using the distribution P, and is given by $\sum_{\{x \in \Sigma^n | f(x) \neq g(x)\}} P(x)$.

Following Angluin (1988), we also assume another oracle called MEMBER that takes any assignment $x \in \Sigma^n$ as input and outputs $f(x)$, where $f \in \mathcal{F}_n$ is the target function.

We now formally define the predictability model with membership queries that we will be using.

Definition 2. For a function space $\mathcal{F}$ and its representation language $\mathcal{R}$, if there is an algorithm $\mathcal{M}$ that, given an error parameter ϵ, a confidence parameter δ, and the problem size n, for all target functions $f \in \mathcal{F}_n$ and all probability distributions P,

(i) makes a number of calls to EXAMPLE and MEMBER that is polynomial in n, $|f|_{\mathcal{R}}$, $\frac{1}{\epsilon}$, and $\frac{1}{\delta}$,

(ii) runs for time polynomial in n, $|f|_{\mathcal{R}}$, $\frac{1}{\epsilon}$, $\frac{1}{\delta}$, and

(iii) with probability at least $1 - \delta$ outputs a a representation of function g that has error at most ϵ according to P and is computable in polynomial time,

then $\mathcal{M}$ is said to *pwm-predict* (pac-predict with membership queries) $\mathcal{F}$ under the representation $\mathcal{R}$. $\mathcal{F}$, represented in $\mathcal{R}$, is said to be *pwm-predictable* under these conditions.

Definition 3. If a function class $\mathcal{F}$ represented in $\mathcal{R}$ is pwm-predictable by an algorithm without making any calls to the MEMBER oracle, then $\mathcal{F}$ is said to be *pac-predictable*.

If $\mathcal{F}$ is the set of all functions represented by the sentences in $\mathcal{R}$, then we say that $\mathcal{R}$ is pwm- (or pac-) predictable to mean that $\mathcal{F}$ represented in $\mathcal{R}$ is pwm- (or pac-) predictable. Note that the output of the learning algorithm g need not be in the language $\mathcal{R}_n$, but $g(x)$ must be computable in polynomial-time for any x. This is what distinguishes *prediction* from *learning*, and is sometimes called representation-independent learning. In this paper, we are interested in prediction algorithms, and by "learning algorithm," we actually mean algorithms for pwm- or pac-prediction. The words "learnable" and "predictable" are used synonymously from now on.

Definition 4. A function g is *consistent* with a set of examples S if $\forall \langle x, o \rangle \in S, g(x) = o$.

Definition 5. A prediction algorithm is *consistent* if it always outputs a representation of function g that is consistent with the set of input training examples.

Typically, prediction algorithms work by finding a small formula that is consistent with a sufficiently large set of random examples.

Definition 6. A *compression* algorithm for a function space $\mathcal{F}$ represented in $\mathcal{R}$ is any algorithm that takes as input n, the size $|f|_{\mathcal{R}}$ of any target function $f \in \mathcal{F}_n$, the confidence parameter δ such that $0 < \delta \leq 1$, and a set of m distinct examples of f, runs in time polynomial in n, $|f|_{\mathcal{R}}$, $\frac{1}{\delta}$ and m, and with probability at least $1 - \delta$ outputs a hypothesis g represented in some language $\mathcal{R}'$, where, g is consistent with the input examples, $|g|_{\mathcal{R}'}$ is bounded by a polynomial "expansion" function $p(n, |f|_{\mathcal{R}})$ and is independent of m, and $g(x)$ can be evaluated on any instance x in time polynomial in n and $|g|_{\mathcal{R}'}$.

A compression algorithm can optionally call the MEMBER oracle and is charged 1 unit of time for each such call. In that case, it is called a *cwm-algorithm* (compression with membership queries).

If $\mathcal{F}$ is the set of all functions represented by $\mathcal{R}$, then we call a compression (or cwm-) algorithm for $\mathcal{F}$ represented in $\mathcal{R}$, a compression (or cwm-) algorithm for $\mathcal{R}$.

Note that the size of the hypothesis output by the compression algorithm should grow slowly with the size of the target function and should be independent of the number of examples m. More general definitions allow it to grow slowly with the number of examples, e.g., according to $O(m^\alpha)$, where $0 \leq \alpha < 1$ (Blumer, Ehrenfeucht, Haussler, & Warmuth, 1989). Thus, compression algorithms compress the training sample into a reasonably small hypothesis and are sometimes called "Ockham algorithms," named after William of Ockham, the original proponent of "Ockham's Razor."

The next theorem follows from the results of (Blumer et al., 1989) and establishes that a compression algorithm can be easily converted into a prediction algorithm by making sure that it is called with a sufficiently large random sample.

THEOREM 1 (BLUMER ET AL.) *Let f be any target function in $\mathcal{F}_n$ representable in $\mathcal{R}$ with c distinct primitive symbols, and p be the polynomial expansion-function defined in Definition 6. Then, any algorithm that takes as inputs n, ϵ, δ, and the size $|f|_{\mathcal{R}}$, calls a compression (or cwm) algorithm with a random example set S of size $\frac{1}{\epsilon}(p(|f|, n) \ln c + \ln \frac{2}{\delta})$ and a confidence parameter $\frac{\delta}{2}$, and outputs the result g that is polynomial-time computable is a consistent pac- (or pwm-) prediction algorithm for $\mathcal{R}$.*

The above theorem suggests that we need a sample whose size grows linearly with $\frac{1}{\epsilon}$ and the size of the output hypothesis. The $\log$ factor is $\ln \frac{2}{\delta}$ instead of the usual $\ln \frac{1}{\delta}$ because the compression algorithm might itself fail to find a consistent hypothesis with a probability less than $\frac{\delta}{2}$. With the sample size given in the theorem, the probability that the returned hypothesis is not approximately correct, given that the compression algorithm successfully finds a consistent hypothesis, is less than $\frac{\delta}{2}$. Adding the two bounds, the probability that the algorithm finds a hypothesis which is not approximately correct is less than δ.

The following "Ockham-converse" result of Schapire (1990) suggests that prediction algorithms can also be converted into compression algorithms, albeit through a more complex algorithmic reduction that involves running the prediction algorithm several times using different example distributions.

THEOREM 2 (SCHAPIRE) *If $\mathcal{F}$ is a pac- (or pwm-) predictable discrete class of Boolean functions in some representation $\mathcal{R}$, then there exists an efficient, deterministic compression (or cwm-) algorithm for $\mathcal{F}$ in $\mathcal{R}$.*

"Efficient" in the above theorem means that it runs in time polynomial in $\log \frac{1}{\delta}$ (rather than in $\frac{1}{\delta}$), in addition to the sample size m, the number of inputs of the target function n, and the size of the target function $|f|$ in its representation language. The output hypothesis can be evaluated in time polynomial in n and the size of the original target function.

Having established the relationship between prediction and compression, we now define some specific properties of the functions and function spaces in order to characterize the conditions under which our algorithm works correctly. We begin with the following.

If x and y are two complete assignments, $x \leq y$ if and only if the value assigned to B_i by x_i is $\leq$ the value assigned to B_i by y_i, for each i. A Boolean function f is *monotone* if $f(x) \leq f(y)$, whenever $x \leq y$.

A Boolean formula over a set of input variables is a possibly nested expression formed from a set of primitive terms that represent "basis functions," each of which can take any number of arguments, which can be input variables or other Boolean formulas.

A Boolean formula is monotone if the basis functions are composed only from AND and OR.

It is easy to see that monotone Boolean formulas represent monotone Boolean functions. In addition, every monotone Boolean function is represented by a monotone Boolean formula.

Definition 7. Given a formula r, the complement of r is the formula $\overline{r}$, obtained by negating it. The length of the formula $\overline{r}$ is denoted by $|\overline{r}|$ and is equal to $1 + |r|$.

The complement of a function is the function obtained by complementing the output of the original function for every input, and is represented by the complement of any formula that represents the original function (as well as possibly other formulas).

Note that the above definition implies that the complement of the complement of a function f is f itself. This means that both $\overline{\overline{r}}$ and r represent the same function.

Whereas the *complement* negates the formula, a *variable negation* negates the input variables of the formula.

Definition 8. Given a formula r over a set of variables X and a subset W of X, the *variable negation* of r with respect to W is the formula obtained by negating all occurrences in r of the variables in W. Its length is equal to $|W| + |r|$.

The variable negation of a function f with respect to W is the function obtained by applying f after complementing the variables in W. The variable negation of any formula for f with respect to W represents the variable negation of f with respect to W.

A *projection* of a function treats some input variables as always fixed. Here we restrict ourselves to projection to 0 rather than to 0 or 1.

Definition 9. Given a formula r over a set of variables X and a subset W of X, the *projection* of r with respect to W is the formula r^W obtained by replacing all occurrences of the variables W in r with 0's. Its length is the same as that of r.

The projection of a function f with respect to W is the function obtained by applying f after fixing the input variables in W to 0's. The projection of a formula for f represents the projection of f.

Definition 10. A representation language $\mathcal{R}$ is *closed under* complement (or variable negation or projection), if, for every formula r in $\mathcal{R}_n$, the complement (or variable negation or projection with respect to any subset of its variables) of r is also in $\mathcal{R}_n$.

Definition 11. A function space $\mathcal{F}$ is *closed under* complement (or variable negation or projection), if, for every function f in $\mathcal{F}_n$, the complement (or variable negation or projection with respect to any subset of its variables) of f is also in $\mathcal{F}_n$.

Definition 12. The *closure of a representation language* $\mathcal{R}$ under complement (or variable negation or projection) is the minimal language $\mathcal{R}^*$ which includes $\mathcal{R}$ and is closed under complement (or variable negation or projection), in that every language which includes $\mathcal{R}$ and is closed under complement (or variable negation or projection) is a superset of $\mathcal{R}^*$.

Definition 13. Analogously, the *closure of a function space* $\mathcal{F}$ under complement (or variable negation or projection) is the minimal function space $\mathcal{F}^*$ which includes $\mathcal{F}$ and is closed under complement (or variable negation or projection).

Consider the space of k-DNF functions, which is known to be learnable. k-DNF is the space of Boolean functions that can be expressed as disjunctions of conjunctions of at most k literals (Valiant, 1984). For example, $A\overline{B} + \overline{B}C + \overline{D}E$ is a 2-DNF formula. It is easy to see that k-DNF is closed under projection. Setting an input to 0 causes all the terms that contain the positive literal of that input to disappear, and all the terms that contain its negated literal to drop that literal from the term, leaving a formula that still represents a function in k-DNF. It is also easy to see that it is closed under variable negation, because complementing any subset of the literals in a k-DNF function expression does not change its k-DNF property. However, k-DNF is not closed under complement. For example, the complement of the previous function is $\overline{A\overline{B} + \overline{B}C + \overline{D}E} = (\overline{A} + B)(B + \overline{C})(D + \overline{E}) = \overline{A}\,\overline{C}\,D + \overline{A}\,\overline{C}\,\overline{E} + BD + B\overline{E}$, which is not expressible in 2-DNF.

Let $\mathcal{F}$ be a function space represented in $\mathcal{R}$ and $\mathcal{F}^*$ be its closure under complement (or variable negation or projection). It is easy to see that the size of any function derived by complementation (or variable negation or projection) from a function $f \in \mathcal{F}_n$ in the corresponding closure $\mathcal{R}^*$ is at most linear in the size of f in $\mathcal{R}$.

LEMMA 1 *If a function space $\mathcal{F}$ represented in the language $\mathcal{R}$ is pac (pwm) predictable, then $\mathcal{R}$'s closure under complement, $\mathcal{R}^*$, has a compression (cwm) algorithm.*

Proof: By Schapire's Ockham-converse result (Theorem 2), since $\mathcal{R}$ is pac (pwm) predictable, it has a compression (cwm) algorithm, say $\mathcal{A}$. We show that we can construct a compression (cwm) algorithm for $\mathcal{R}^*$ using $\mathcal{A}$ as a subroutine.

Every target formula in $\mathcal{R}^*$ has a functionally equivalent formula, which is not bigger than the original formula, and is of the form f or $\overline{f}$, where f is a formula in $\mathcal{R}$. To learn the function space $\mathcal{F}^*$ represented by $\mathcal{R}^*$, we run two copies of $\mathcal{A}$ for time $p(n, |f|_{\mathcal{R}}, \log \frac{1}{\delta}, m)$ where n is the number of features of each example, m is the number of examples, δ is the confidence parameter, and p is the polynomial upper-bound on the running time of $\mathcal{A}$. The first copy assumes that the target function f is represented in $\mathcal{R}$ and tries to learn it directly. The second copy assumes that the target is represented as $\overline{f}$ where f is in $\mathcal{R}$. Hence, the second copy turns the examples of the target into the examples of f by complementing their output bits, runs $\mathcal{A}$ on them, and returns the complement of the learned formula. (If $\mathcal{A}$ is

a cwm-algorithm, all the answers to the membership queries are similarly complemented before being passed on to the second copy of $\mathcal{A}$.)

Since the target function is equivalent to f or $\overline{f}$ at least one of the two copies must terminate in the above time bound, and output a hypothesis consistent with the input example set with probability at least $1 - \delta$. Consistency can be checked easily by evaluating the hypotheses output by the two copies on the input examples for time determined by the upper bound on the evaluation time (which is a polynomial in the hypothesis and example sizes). This serves as a compression (cwm) algorithm for $\mathcal{R}^*$ since at least one copy returns a consistent hypothesis with probability $1 - \delta$, and it can be easily determined which one it is by checking it against each example. In case both of them are consistent, one can be returned arbitrarily. □

Unfortunately the corresponding results are not known to be true for closures under variable negation and projection.[3]

4. Tree-Structured Bias

In this section, we present formal definitions of determinations and tree-structured bias, in the propositional setting. A complete summary of notation is given in Table 1.

Let an object x be described by a set of input attributes $B_1, \ldots, B_n$, whose values are denoted by $B_1(x), \ldots, B_n(x)$. The task is to predict the value of the output attribute A_0 for x.

Out of the many types of determinations Russell defined, in this paper, we are concerned with what may be called a "functional" determination (Russell, 1989). This is defined as follows.

Definition 14. A set of attributes $\{P_1, \ldots, P_l\}$ *determines* Q, if there is an l-input function f such that $\forall x\ Q(x) = f(P_1(x), \ldots, P_l(x))$.

Intuitively, the above determination says that some subset of attributes $P_1, \ldots, P_l$ are sufficient to predict the value of Q for any object x. Thus, if there are two objects x and y that have exactly the same values for $P_1, \ldots, P_l$, then they will have the same value for Q. Note that this leads to a first-order characterization of the determination:

$$\forall x, y\ (P_1(x) = P_1(y)) \wedge \ldots \wedge (P_l(x) = P_l(y)) \Rightarrow Q(x) = Q(y)$$

There are many examples of such determinations: the nationality of an individual might determine her language; the model, year, and make of a car determine its list price; the inputs of a combinatorial Boolean circuit determine its output; and so on.

The most important property of determinations for our purposes is that they reduce the set of functions that the learner has to consider by identifying the relevant features. If the nationality of a person determines her language, then the learner need not consider functions that map different people of the same nationality to different languages, since they are not consistent with this determination. Since learning occurs by filtering out functions that are not consistent with the training examples, the more constraining the determination, the fewer the examples needed to filter the remaining functions. As we stated in the introduction, if the determination for the target concept is derived by a tree-structured derivation such that

the number of children of each node is bounded by k, then the number of examples needed for pac-learning is linear in the number of leaves (which may, in turn, be much less than the total number of observable features).

If i is a node in the tree structure T, let A_i represent the attribute at i, and let $i_1, \ldots, i_{l(i)}$ represent the children of i. Let A_0 denote the target attribute. T_i represents the subtree of T rooted at node i. Hence T_0 is the same as T. The n input attributes are all distinct and are denoted by $B_1, \ldots, B_n$. During learning, the input attributes $B_1, \ldots, B_n$ and the target attribute A_0 are observable, and the other internal attributes A_i are not. The set of all input attributes other than the leaves of T_i is called the *context* of T_i.

A *partial assignment* is a mapping from the input attributes $\{B_1, \ldots, B_n\}$ to $\{0, 1, *\}$. The attributes that are mapped to 0 or 1 by the partial assignment are called *defined*, and the rest *undefined*.

The *restriction* of an assignment x to a subtree T_i, denoted by x_i, is a partial assignment where the leaves of T_i are assigned to their values under x and the rest of the input attributes are assigned to $*$'s.[4]

If x and y are two (partial) assignments, 'x *extended by* y' denoted x/y, is a (partial) assignment that maps an attribute B_i to $B_i(x)$ when it is defined (not equal to '$*$') and to $B_i(y)$ otherwise.

Note that x/y is defined exactly for those attributes that are defined for either x or y. When an attribute is defined for both x and y, the value under x overrides that under y. Hence the operation '$/$' is associative, but not commutative in general.

Definition 15. A *determination tree* is a tree structure T of attributes such that the root of the tree is the target attribute A_0, the leaves of the tree correspond to the input attributes $B_1, \ldots, B_n$, and for every non-leaf node i in the tree the set of attributes at the children of i, $\{A_{i_1}, \ldots, A_{i_{l(i)}}\}$, determines A_i.

The function f_i from the attributes $\{A_{i_1}, \ldots, A_{i_{l(i)}}\}$ to A_i is called an "internal function." Given a tree T and a set of internal functions $f_0, f_1, \ldots$, at nodes $0, 1, \ldots$, there is a unique "external" function from the input attributes of the tree to any internal attribute A_i. We denote this external function by T_i^f, and drop the superscript f when it is clear from the context. Hence,

$$T_i^f(x) = T_i(x) = f_i(T_{i_1}^f(x), \ldots, T_{i_{l(i)}}^f(x))$$

for all inputs x. Note that T_0^f is the same as T^f, and for all leaf nodes i, $T_i(x)$ is the value of the only defined attribute of x_i.

The number of children of a node is called its *fan-in* and the number of parents of a node is called its *fan-out*. The *fan-in* of the tree is the maximum fan-in over all nodes. The *fan-out* of the tree is the maximum fan-out over all nodes, which is 1. To nontrivially constrain the set of allowed functions, the fan-in of the tree was originally restricted to a constant k (Russell, 1989; Bshouty et al., 1992; Tadepalli, 1993) — the k-TSB bias. In this paper, we consider a generalized tree-structured bias in which the nodes have arbitrarily large fan-ins. We restrict the set of allowed functions by assuming that the internal functions belong to a function space that is itself learnable from random examples and membership queries.

Table 1. A Summary of the notation used in the paper

Notation	Meaning
$\mathcal{F}$	A space of functions
$\mathcal{F}_n$	A subspace of $\mathcal{F}$, where each function has n inputs
$\mathcal{R}_n$	Formulas of functions over n inputs, represented in $\mathcal{R}$
A_i	The internal attribute at node i
A_0	The root (target) attribute
B_i	The input attribute at the leaf node i
x_i	A restriction of assignment x to T_i
x/y	An assignment where input attribute values under x (if not = '*') override that under y.
z	An assignment which is 0 everywhere
$\perp$	A special distinguished symbol to denote "unknown"
i_j	j^{th} child of node i
$l(i)$	Number of children of node i
n_I	Number of internal nodes of the determination tree
n	Number of leaves of the determination tree
f_i	Target internal function from the children of i to i
g_i	Learned internal function from the children of i to i
$d(i)$	Distinguishing assignment of node i
$sign(i)$	The output of the target function on $z_i/d(i)$; sign(0) = 0
D_i	Children of i with unknown distinguishing assignments.
$f_i^{D_i}$	Projection of f_i with respect to the inputs in D_i
$T_i = T_i^f$	The external function from the leaves of subtree rooted at i to A_i induced by the internal functions $f_1, f_2, \ldots$
$T^f = T_0^f$	The external function from the leaves of the tree to A_0 induced by the internal functions $f_1, f_2, \ldots$
$\langle x_i, o_i \rangle$	An example (input-output pair) of function T_i^f
S_i	Training examples (input-output pairs) for the function T_i^f
$T_i^f(x)$	The value of A_i on x, where internal functions are $f_1, f_2, \ldots$
$T^{\mathcal{I}}$	The set of target functions induced by the internal functions in $\mathcal{I}$
TSB-2^X	The TSB-2 algorithm that uses X to learn the internal functions

Let the internal functions f_i be chosen from a function space $\mathcal{I}$ represented in some language $\mathcal{R}$. Following the literature on read-once formulas, we call the function space $\mathcal{I}$ the "basis" (Hancock & Hellerstein, 1991; Bshouty et al., 1992, 1995).

The hypothesis space represented by the tree-structured bias T and the internal function space $\mathcal{I}$ is just the set of all external functions at the root node definable by choosing functions from $\mathcal{I}$ for each node i in T.

Definition 16. The set of functions *induced by* the function space $\mathcal{I}$ over the tree structure T, denoted $T^{\mathcal{I}}$, is given by $\{T^f | f_i \in \mathcal{I}_{l(i)}$ for all nodes i of $T\}$, where $\mathcal{I}_{l(i)}$ is the subspace of functions in $\mathcal{I}$ with $l(i)$ inputs.

We assume that the functions T_i^f are represented in the natural fashion; by the representation of f_i followed by the representations of functions $T_{i_1}^f, \ldots, T_{i_{l(i)}}^f$ in parentheses. We denote by $T^{\mathcal{R}}$ the set of representations of T_0^f, where the internal functions f_i are represented in the language $\mathcal{R}$.

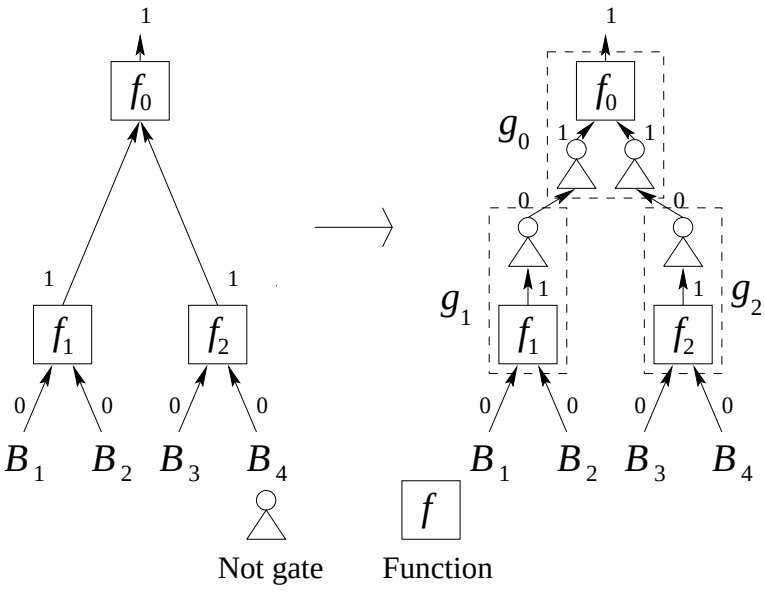

Figure 3. Transforming the target formula so that each of the intermediate nodes in the determination tree outputs 0 on the null assignment. The transformation increases the length of the formula at most by twice the number of intermediate nodes in the determination tree.

The size of a function T_i^f, represented in this fashion is the sum of the sizes of all internal functions in the subtree T_i. This measure of the function size is justified since we already know the tree structure and $|T_i^f|$ would be known if we know each of the internal functions. Note that we count all internal nodes, even if they compute a constant function. Since every internal function needs at least 1 bit to represent it, $|T_i^f|$ is greater than the number of internal nodes in T_i.

We are interested in learning the set of functions $T^{\mathcal{I}}$ represented by $T^{\mathcal{R}}$ from random examples and membership queries for any arbitrary tree-structured bias T. In contrast, to learn read-once formulas the algorithm has to learn the determination tree T, which is also called the "skeleton" of the read-once formula, and the functions $T^{\mathcal{I}}$. In other words, it has to learn the functions in $\bigcup_T T^{\mathcal{I}}$, where T varies over all possible determination trees. In this paper, we examine the effect on learning of having the skeleton T as prior knowledge.

Definition 17. A *null assignment* is a special assignment where every input attribute is set to 0. We denote this by the symbol z.

Note that z_i denotes the partial assignment where the leaves of the subtree T_i are assigned to 0's and the rest of the inputs are undefined. We call the non-root internal nodes of the determination tree the "intermediate nodes."

Definition 18. A representation of a target function T_0^f is *well-behaved* if for all intermediate nodes i, $T_i^f(z_i) = 0$.

We now prove a lemma that lets us assume, under some conditions, that the representation of the target is well-behaved, i.e., all intermediate nodes evaluate to 0's when the leaves of the subtrees under them are set to 0's.

LEMMA 2 *Let $\mathcal{R}$ be the language of the basis functions of the target hypothesis space $T^{\mathcal{R}}$, and let $\mathcal{R}^*$ include $\mathcal{R}$ and be closed under complement and variable negation. For any target formula $T_0^f \in T^{\mathcal{R}}$, there exists a functionally equivalent formula $T_0^g \in T^{\mathcal{R}^*}$, such that (a) T_0^g is well-behaved, and (b) for any internal node i, $|g_i| \leq |f_i| + l(i) + 1$, where $l(i)$ is the number of children of i.*

Proof: The proof is by construction. Starting with the bottom of the determination tree, we replace each internal function in the tree until the conditions of the lemma are satisfied.

Assume that i_j is one of the bottom-most intermediates nodes in the determination tree such that $T_{i_j}^f(z_{i_j}) = 1$, which violates condition (a). We can introduce two successive NOT gates at the output of node i_j without changing the target function T_0^f computed by the original formula (see Figure 3). If we define g_{i_j} as $\overline{f}_{i_j}$, then $T_{i_j}^g(z_{i_j}) = 0$. We repeat this for each child of i which outputs 1 on null input. Let W be the children of i whose internal functions are thus replaced. We then replace f_i with its variable negation with respect to W. This ensures that the external functions computed at node i and at the root of the tree do not change. We repeat the above process until all the intermediate nodes satisfy this condition. This process terminates because nodes are replaced in a strictly bottom-up fashion. Since at each internal node i, we added at most one NOT gate to its output and one NOT gate to each of its inputs, the size of the formula increases at most by $l(i) + 1$.

Since we replaced the internal function formulas only by their complements and variable negations, the new internal functions are all represented in $\mathcal{R}^*$, and the new target formula is in $T^{\mathcal{R}^*}$. □

Figure 3 shows this transformation on an example tree. The dashed boxes in the right half of Figure 3 represent the new internal functions g_0, g_1, and g_2, which are derived from f_0, f_1, and f_2, by complementation and variable negation. It is easy to check that the transformed circuit is functionally equivalent to the original one and is well-behaved.

5. Distinguishing Assignments

In this section, we motivate and introduce the idea of "distinguishing assignments," which is closely related to controlled experimentation in science and forms the basis of our learning algorithm.

5.1. Definition and Properties

The main problem in learning with a tree-structured bias is that the non-root internal attributes in the determination tree are not observable. The basic idea behind our algorithm is that an internal node (e.g., A_{i_2} in Figure 4) can be made effectively observable by finding

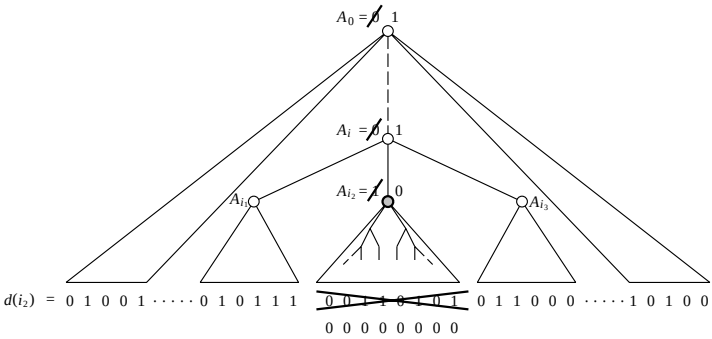

Figure 4. Distinguishing assignment for node i_2: changing the values of the leaves of the middle subtree T_{i_2} to 0's changes the values of all the attributes in the path from i_2 to the root.

a context, i.e., an assignment to all input attributes other than the leaves under that node, for which the value of the root attribute A_0 changes whenever the node's value changes. In Figure 4, assume that changing the values of the leaves of T_{i_2} to 0's while keeping the rest of the leaves at the specified values changes the value of the root attribute A_0 from 0 to 1. Since all the paths from the leaf nodes of T_{i_2} to the root pass through i_2, in order for the value of A_0 to change, the value of A_{i_2} must have changed. Under these conditions, we call the original assignment a *distinguishing assignment* for node i_2 and denote it with $d(i_2)$. Since the value of A_{i_2} is assumed to be 0 when the leaves of T_{i_2} are set to 0's, it must have been 1 originally. Since the value of A_{i_2} determines the value of A_0 under a fixed context, the values of A_0 and A_{i_2} are always *opposite* if the context of T_{i_2} is set according to $d(i_2)$. We denote this situation with $sign(i_2) = 1$. If, on the other hand, setting the leaves of T_{i_2} to 0's changes the value of the root A_0 from 1 to 0, then the values of A_0 and A_{i_2} are always *identical* in the context of $d(i_2)$, and $sign(i_2) = 0$.

Thus, if a distinguishing assignment and a sign for an internal node i_2 are known, then the value of A_0 can be used to "read off" the value of A_{i_2} for any input at the leaves of T_{i_2} by setting its context according to $d(i_2)$, making A_{i_2} effectively observable. For the distinguishing assignment $d(i_2)$ in Figure 4, since $sign(i_2) = 1$, if A_0 is 1, then A_{i_2} must be 0 and vice versa. A distinguishing assignment for a node i_2 also makes it possible to "control" its value. To set the value of A_{i_2} to 0, we can simply set all leaves of T_{i_2} to 0's. To set it to 1, we set them according to $d(i_2)$.

We are now ready to formally define distinguishing assignments and make the above intuitions more precise. Recall that z_i/x in the following definition refers to an assignment that maps the input attributes under the subtree T_i to 0's and all other input attributes to their values under x.

Definition 19. For a determination tree T, and an intermediate node i, an assignment x such that $T(x) \neq T(z_i/x)$ is called a *distinguishing assignment* for i, and is denoted by $d(i)$. If $d(i)$ is a distinguishing assignment for i, $T(z_i/d(i))$ is called the corresponding *sign* of node i and is denoted by $sign(i)$.

Note that the $sign$ of a node is always associated with a particular distinguishing assignment in that the same node might have different signs for different distinguishing assignments.

PROPOSITION 1 *Let i be any intermediate node in a determination tree T with a distinguishing assignment $d(i)$ and sign $sign(i)$. Then, for any assignments x and y,*

1.1. $T_i(x) = T_i(x_i) = T_i(x_i/y)$.
1.2. $T_i(d(i)) = 1$.
1.3. $T(x_i/d(i)) = T(y_i/d(i)) \Longleftrightarrow T_i(x) = T_i(y)$.
1.4. $T_i(x) = T(x_i/d(i)) \oplus sign(i)$.
1.5. $T(x_i/d(i)) = T_i(x) \oplus sign(i)$.

Proof:

1.1. This is true because only the leaves of the subtree T_i influence the value of node i, and the assignments x, x_i, and x_i/y all have the same values for these leaves.

1.2. Because of the tree structure of the determinations, for any intermediate node i, all paths in the tree from the leaves of T_i to the root go through the node i. Since $d(i)$ and $z_i/d(i)$ differ only in the values of the leaves of T_i, the change in the value of the root node of T can only be caused by a change in the value of node i. This implies that $T_i(d(i)) \neq T_i(z_i/d(i))$. Since $T_i(z_i/d(i)) = T_i(z_i) = 0$, $T_i(d(i)) = 1$ by Definition 19.

1.3. ($\Longleftarrow$) By Proposition 1.1, $T_i(x) = T_i(y)$ iff $T_i(x_i/d(i)) = T_i(y_i/d(i))$. Since $x_i/d(i)$ and $y_i/d(i)$ differ only in the leaves of T_i, and since all paths from the leaves of T_i must go through the node i to reach the root, if the node i has the same values for $x_i/d(i)$ and $y_i/d(i)$, then the root has the same values as well.

($\Longrightarrow$) Suppose that $T(x_i/d(i) = T(y_i/d(i))$ and $T_i(x_i) \neq T_i(y_i)$. Without loss of generality, let $T_i(x_i) = 0 = T_i(z_i)$ so that $T_i(y_i) = 1 = T_i(d(i)) = T_i(d(i)_i)$ by Propositions 1.1 and 1.2. By the proof of the $\Longleftarrow$ case, $T(x_i/d(i)) = T(z_i/d(i))$ and $T(y_i/d(i)) = T(d(i)_i/d(i)) = T(d(i))$. Since we assumed that $T(x_i/d(i)) = T(y_i/d(i))$, it follows that $T(z_i/d(i)) = T(d(i))$, which contradicts the definition of $d(i)$.

1.4. If $T(x_i/d(i))$ is the same as $sign(i) = T(z_i/d(i))$, then by the second part ($\Longrightarrow$) of Proposition 1.3, $T_i(x) = T_i(z_i) = 0$. Otherwise, by the first part ($\Longleftarrow$) of Proposition 1.3, $T_i(x) \neq T_i(z_i)$, which means $T_i(x) = 1$. Hence, in either case, $T_i(x) = T(x_i/d(i)) \oplus sign(i)$.

1.5. This is a straightforward consequence of Proposition 1.4.□

Proposition 1 suggests that a distinguishing assignment for an intermediate node i gives us the abilities to *control* and to *observe* the value of A_i. Control, i.e., setting A_i to 0 or 1, is possible by setting the leaves of T_i to 0's or as specified in $d(i)$ respectively. Observing the output of A_i for any arbitrary input, i.e., simulating the MEMBER oracle for A_i, is possible since switching the value of A_i switches the target attribute value when the context of T_i is

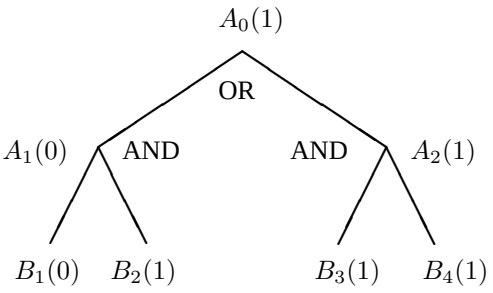

Figure 5. A determination tree and a target function: OR-OF-ANDS.

set according to $d(i)$. To observe $T_i(x)$, for any input x, we set the leaves of T_i according to x, and all other inputs according to $d(i)$. In other words, we ask a membership query on the assignment $x_i/d(i)$. This corresponds to asking the value of the root node for x_i, using the restriction of $d(i)$ onto the context of T_i as a "mask". We then exclusive-OR the result with $sign(i)$ as suggested by Proposition 1.4.

In Figure 5, the target function is OR-OF-ANDS = $B_1\,B_2 + B_3\,B_4$. Hence, $f_1 = f_2 =$ AND, and $f_0 =$ OR. The numbers in the parentheses represent the values of the attributes for a particular assignment $B_1 = 0, B_2 = 1, B_3 = 1, B_4 = 1$, which we will write as 0111 for convenience. The assignment 0111 is a distinguishing assignment for node 2, because $T(0111) \neq T(0100) = 0$. Hence, A_2 can be set to 1 by setting the leaves of its subtree, B_3B_4, to 11. Its value on any input can be observed because it is always the same as the value of A_0 when its context, B_1B_2, is set to 01.

Not all intermediate nodes i need have distinguishing assignments. For example, in Figure 5, A_1 would not have had a distinguishing assignment if f_2 were the constant function **1**. Then A_0 would always be 1, independent of the value of A_1. A node is called *live* for a given target function if it has a distinguishing assignment, and *dead* otherwise. If a node i is dead, then $T(x) = T(z_i/x)$ for all assignments x, which means that permanently replacing A_i's value with 0 would not affect the value of A_0 for any input.

In the computational learning theory literature, a similar notion of "justifying assignment" is used (Angluin et al., 1993; Hancock & Hellerstein, 1991; Bshouty et al., 1992, 1995). A justifying assignment is similar to the distinguishing assignment except it is defined only for input attributes, i.e., the leaf nodes of the determination tree. The justifying assignment x for an attribute B_i is a partial assignment defined on all other input attributes such that assigning B_i to 0 or 1 changes the value of the target function. If we use $\{B_i \leftarrow Y\}/x$ to represent the assignment where B_i is mapped to the value Y (which can be 0, 1, or $*$) and all other attributes are mapped according to x, then $\{B_i \leftarrow *\}/x$ is a justifying assignment if and only if $T(\{B_i \leftarrow 0\}/x) \neq T(\{B_i \leftarrow 1\}/x)$. Note that a change in the value of the leaf of a determination tree can change the value of the target function only by changing the values of all the internal nodes in the path from that leaf to the root. Hence if $\{B_i \leftarrow *\}/x$ is a justifying assignment for B_i, then for all internal nodes in the path from the root to B_i, either $\{B_i \leftarrow 0\}/x$ or $\{B_i \leftarrow 1\}/x$ is a distinguishing assignment.

5.2. Relationship to Controlled Experimentation

It is worth examining in more detail the analogy between distinguishing assignments and controlled experimentation in science. The goal of controlled experimentation is to discover the properties of an unobservable "theoretical entity." Examples of such theoretical entities range from fundamental particles in physics to disease states in medicine. Usually the experimenter has a number of parameters under direct control, such as throwing an electric switch in a physics experiment or administering a drug to a patient. These independent variables correspond to the input attributes of our determination tree. Similarly, the experimenter can also observe some final outcomes. For example, in the physics experiment one might be able to examine the exposure on a photographic plate. In the medical experiment, the patient's temperature and blood pressure may be observable as well as other physical symptoms. These are akin to the observable root nodes of the determination tree.

In controlled experimentation, the scientist relies on the differences of observations between the main "experimental setting" and the "control setting" of parameters to make a prediction of the theoretical entity. A distinguishing assignment of a node can be understood as a controlled experiment with two components of parameter values. One component is the set of values for the so-called "context" variables which should be the same for both the experimental and control settings. The other component is the values for the "control" variables that can causally influence the theoretical entity, i.e., the leaves of the determination subtree under the hidden node in question. The values for these variables are different for the experimental and the control settings. To the extent that there is a difference in the final observations between the two settings, the experimenter concludes that the theoretical entity has different values in these two settings, is controllable by the control variables, and has effects that are observable under the current context.

Viewing controlled experimentation in the framework of distinguishing assignments helps us identify the assumptions behind its use. One important assumption is that every causal path from the control variables to the observed variable is mediated through the theoretical entity (no "causal leak"). Otherwise, we are indeed not sure if the difference in observations is due to the difference in the states of the theoretical entity or is due to something else. Another important assumption is that there is no causal path from the context variables to the theoretical entity, or else we will not be able to isolate the effects of the control variables on the theoretical entity from those of the context variables (no "causal seep").

To make things concrete, let us suppose that throwing a switch in a physics experiment exposes a photographic film. The conditions of the experimental and control settings are exactly the same except that in the control setting, the switch is off and in the experimental setting, the switch is on. We are confident that the exposure of the film indicates the presence of a particle to the extent that there is no other causal mechanism that can expose the film when the switch is thrown. We can also isolate the settings of the context variables that can be used for observation from those of the control variables if none of the context variables can affect the *presence* of the particle, only its observation. This suggests that ideally the electric circuitry that helps bring about the particle's presence must be causally isolated from the photographic equipment that detects its presence, the only mediation between the two being through the particle.

Finally, the viewpoint of distinguishing assignments helps us understand the role of some important phenomena such as the "placebo effect" in experimental science and how they are usually avoided. The placebo effect can occur when a patient thinks that he is given a medicine. This mere belief can sometimes influence his own psychological and even physiological condition to the extent that he may be observed to be cured even when the placebo does not in fact contain any medicine and has no direct effect on the real disease. One way to understand this situation is to posit two causal pathways from taking the medicine to the observable symptoms. One pathway is through the physical disease that we would like to cure. Another is through some ill-understood psycho-physiological route that presumably affects the symptoms without affecting the disease. Thus, if we conduct a study where the experimental group[5] is given a medicine and the control group is given nothing, we have the problem of causal leak, i.e., a causal pathway from a control variable (of giving medicine) to the observable symptoms that bypasses the disease node. Thus, we cannot separate the effects of the medicine from the placebo effect. The usual way to avoid this problem is to give pills to both groups of patients, where the pills given to the experimental group contain the medicine and those given to the control group contain the placebo. The reason that this avoids the causal leak is that now we have split the control variable into two, one of giving the pill and the other of actually putting the medicine in the pill. Moreover, since giving the pill is common to both groups, it is now a context variable. Any observed differences between the two groups can now be attributed solely to the medicine, because it is the only control variable, and the leaky causal path from taking the pill to the observed symptoms is common to both groups and moved into the context.

6. The Learning Algorithm

In this section, we present an algorithm to pwm-predict the function space $T^{\mathcal{I}}$ for any given tree-structure T, assuming that $\mathcal{I}$ is (a) pac-predictable and closed under variable negation or (b) pwm-predictable and closed under projection and variable negation. In other words, we always require the basis class to be closed under variable negation and to have a prediction algorithm. If the basis class also needs the MEMBER oracle in addition to random examples to learn it, then we also require it to be closed under projection.

The learning algorithm works top-down on the determination tree, finding the distinguishing assignments for each intermediate node, and learning the internal function at each node. Note that if the distinguishing assignments and signs of any node i and its children $i_1, \ldots, i_l$ are known, then we have the ability to set each of the values of $A_{i_1}, \ldots, A_{i_l}$ to 0 or 1, and infer the value of A_i. We can also infer the values of $A_{i_1}, \ldots, A_{i_l}$, and that of A_i for any random example. Since $A_{i_1}, \ldots, A_{i_l}$ are the inputs of f_i, and A_i is its output, we can generate random examples of f_i using random examples of the target function. In other words, we can effectively simulate MEMBER and EXAMPLE oracles for the internal function f_i.[6] If f_i belongs to the basis $\mathcal{I}$ that is learnable from examples and membership queries, then we can learn it using the simulated oracles. We will see that if the algorithm is not successful in generating the distinguishing assignment for a node i_j using a sufficiently large sample, then, with high probability, A_{i_j} can be ignored without losing approximate correctness in predicting the target attribute A_0. Under these conditions, f_{i_j} is set to the constant function $\mathbf{0}$. On the other hand, if the algorithm can find a distinguishing assignment

```
01 procedure Learn-w-TSB
02   input
03     T_i /* subtree rooted at node i */
04     S_i /* set of restrictions of instances in S onto T_i and their outputs at A_i */
05     d(i) /* distinguishing assignment of node i ( arbitrary for the root) */
06     sign(i) /* = the sign of d(i) (0 for the root)*/;
07   /* Find distinguishing assignments for i's children */
08   ⟨S_i, d(i_1), sign(i_1), ..., d(i_l(i)), sign(i_l(i))⟩ :=
09         Distinguish-Children(T_i, S_i, d(i), sign(i))
10   /* Decompose examples of T_i^f into examples of f_i and T_i1^f, ..., T_il(i)^f */
11   ⟨Internal-Examples, S_i1, ..., S_il(i)⟩ :=
12         Decompose-Examples (T_i, S_i, d(i_1), sign(i_1), ..., d(i_l(i)), sign(i_l(i)))
13   for each child i_j of i
14     if i_j is a leaf, then T_ij^g = the identity function Id
15       else if d(i_j) = ⊥ then T_ij^g := the constant function 0
16         /* Recursively call nodes with known distinguishing assignments,
17            on restrictions of examples onto the corresponding subtrees */
18          else T_ij^g := Learn-w-TSB(T_ij, S_ij, d(i_j), sign(i_j));
19   end for;
20   /* Learn an approximation g_i for the internal function f_i */
21   g_i := Learn-Internal-Function (Internal-Examples)
22   output T_i^g := ⟨g_i, ⟨T_i1^g, ..., T_il(i)^g⟩⟩;
23 end Learn-w-TSB;
```

Figure 6. The top level algorithm for learning with a tree-structured bias.

for a node i_j, then it can, in effect, observe the value of A_{i_j} for any input. Hence, it can call itself recursively on such i_j. Figure 6 shows the top level algorithm.

The recursive algorithm Learn-w-TSB takes a subtree T_i of the determination tree, the restrictions S_i of the random example set S onto T_i, a distinguishing assignment and sign for node i as inputs (lines 02–06 of Figure 6). The top-level call is made on the full tree T_0 with the example set S. $d(0)$ is any arbitrary assignment and $sign(0) = 0$. The subroutine Distinguish-Children is called to find the distinguishing assignments and signs of all children of i (lines 07–09). Decompose-Examples then decomposes the examples of T_i into the examples of the external functions $T_{i_1}, \ldots, T_{i_{l(i)}}$ and the internal function f_i, by inferring the attribute values of all children of i using their distinguishing assignments and signs (lines 10–12). If a child i_j is a leaf, then Learn-w-TSB treats the corresponding function as the identity function Id (line 14). All non-leaf children without distinguishing assignments are treated as computing the constant function **0** (line 15). The algorithm recursively calls itself on all other children of i (lines 16–18). Learn-Internal-Function is used to generalize the examples of the internal function f_i to a consistent hypothesis (lines

20–21). Finally, the algorithm returns an approximation T_i^g to the target function T_i^f by putting together the results of Learn-Internal-Function and the recursive calls (line 22).

The next three sections describe the algorithm in more detail and prove its correctness.

6.1. Finding Distinguishing Assignments

In this section we describe the method for finding the distinguishing assignments for the internal nodes of the determination tree.

Assume that the internal functions in the target are representable in a language $\mathcal{R}$. Let $\mathcal{R}^*$ be a language that includes $\mathcal{R}$, is closed under complement and variable negation, and is pwm-predictable. Then, by Lemma 2, there is a formula with internal functions chosen from $\mathcal{R}^*$ that is functionally equivalent to the target, such that each intermediate node in the determination tree evaluates to 0 on the null input. Hence, from now on we assume $T_i(z_i) = 0$ for all intermediate nodes i, and that the internal functions are represented in $\mathcal{R}^*$.

Since the root node is observable during training, a distinguishing assignment is not needed for it and can be arbitrarily initialized. However, we find it convenient to initialize the $sign$ of the root node to 0. Note that for the root node, unlike for the intermediate nodes, $T(z_0)$ may not necessarily be 0 and $T(d(0))$ may equal $T(z_0/d(0)) = T(z_0)$.

Given a distinguishing assignment $d(i)$ for the node i, and some random example set S, Distinguish-Children attempts to find the distinguishing assignments and signs for i's children, $i_1, \ldots, i_l$, in that order (see Figure 7). It does this by "marching" all examples $\langle e_i, o_i \rangle$ from $e_i/d(i)$ toward $z_i/d(i)$, successively turning the values of leaf nodes of subtrees whose distinguishing assignments are not found to 0's. For each child i_j of i, it compares the values of A_0 on the instances $e_i/d(i)$ and $z_{i_j}/e_i/d(i)$ for each example e (lines 14–15 of Figure 7). (Note that $o(i) \oplus sign(i) = T(e_i/d(i)$ in line 15 by Proposition 1.5, since $o(i) = T_i(e)$.) If the value of A_0 is different for these two instances for *any* example, then a distinguishing assignment $d(i_j)$ is found (line 16). When a distinguishing assignment for i_j is found, the value of the target attribute A_0 on $z_{i_j}/e_i/d(i)$ is stored as $sign(i_j)$ (line 17). For any assignment x, the value of $T_{i_j}(x)$ can be obtained as $T(x_{i_j}/d(i_j)) \oplus sign(i_j)$ by Proposition 1.4.

If a distinguishing assignment is not found by this method, then it must be the case that the value of A_0 is *unaffected* by the replacement of e_{i_j} by z_{i_j} in *any* training example $\langle e_i, o(i) \rangle$. Therefore, we can assume without any risk of error on the training examples that $T_{i_j}^f$ is the constant function **0**, i.e., that the node i_j is dead. (We will show this more formally in Section 6.3.) In that case, $d(i_j)$ is left to be $\perp$, each example $\langle e_i, o_i \rangle$ in the set S_i is changed to $\langle z_{i_j}/e_i, o_i \rangle$ (line 19), and the search is repeated for the distinguishing assignment for the next child of i.

For example, assume that the routine Learn-w-TSB is called with the random training example set $\{\langle 0111, 1\rangle, \langle 1011, 1\rangle, \langle 0101, 0\rangle\}$ for the target function OR-OF-ANDS of Figure 5. The routine Distinguish-Children gets called with the same training set, with $d(0)$ initialized to some arbitrary instance and $sign(0)$ initialized to 0. It then compares the outputs of the target function for the three examples with those obtained by setting the leaves of the subtrees under A_1 and A_2 to 0's. Doing this for the subtree under A_1 for the first example, it asks a membership query on 0011 which is answered with a 1. Since this

```
01 procedure Distinguish-Children
02   input
03     T_i /* subtree rooted at node i */
04     S_i /* restrictions of instances in S onto T_i with their outputs at A_i) */
05     d(i) /* distinguishing assignment of node i (arbitrary for the root) */
06     sign(i) /* the sign of d(i) (0 for the root)*/;
07   /* Initialization */
08   for each child node i_j of i
09     d(i_j) := ⊥; sign(i_j) := 0;
10   end for;
11   for each non-leaf child node i_j of i
12     for each ⟨e_i, o(i)⟩ ∈ S_i until d(i_j) is found
13       /* Use the MEMBER oracle to read the output after setting e_{i_j} to 0's */
14       let Temp = MEMBER(z_{i_j}/e_i/d(i))
15       if o(i) ⊕ sign(i) ≠ Temp then { /* Note: o(i) ⊕ sign(i) = T(e_i/d(i)
16             d(i_j) := e_i/d(i);          by Proposition 1.5, since o(i) = T_i(e) */
17             sign(i_j) := Temp; }
18     end for;
19     if d(i_j) = ⊥ then replace each ⟨e_i, o_i⟩ ∈ S_i with ⟨z_{i_j}/e_i, o_i⟩
20   end for;
21   output ⟨S_i, d(i_1), sign(i_1), ..., d(i_{l(i)}), sign(i_{l(i)})⟩
22 end Distinguish-Children;
```

Figure 7. Algorithm to find distinguishing assignments and signs for i's children.

is the same as the output of the original assignment (which is $1 \oplus sign(0) = 1$), it is not a distinguishing assignment. The query is repeated for the second example with exactly the same result. For the third example $\langle 0101, 0\rangle$, the query 0001 also produces the same result as the output of the original assignment $0 \oplus sign(0) = 0$. Hence, the node 1 is assumed dead, and the example set is permanently changed to $\{\langle 0011, 1\rangle, \langle 0001, 0\rangle\}$. The search continues for the distinguishing assignments for node 2 with a membership query on 0000. This query is answered with a 0, and since it is not the same as the output of the first assignment, 0011 is made a distinguishing assignment for node 2. Its $sign$ is $T(0000) = 0$. Hence, Distinguish-Children returns the example set $\{\langle 0011, 1\rangle, \langle 0001, 0\rangle\}$, $d(1) = \perp$, $sign(1) = 1$ (which is not used), $d(2) = 0011$, and $sign(2) = 0$.

6.2. Learning Internal and External Functions

In this section we describe how the distinguishing assignments for the children of node i are used to learn the internal function f_i and the external functions of the children of i. The subroutine Decompose-Examples, shown in Figure 8, decomposes each example of the external function at node i into an example of f_i and an example of the external

01 **procedure** Decompose-Examples
02 **input**
03 T_i /* subtree rooted at node i */
04 S_i /* restrictions of instances in S onto T_i with their outputs at A_i */
05 $d(i_1)$ /* = distinguishing assignment of node i_1 */
06 $sign(i_1)$ /* = the sign of $d(i_1)$ */
07 ...
08 ...
09 $d(i_{l(i)})$ /* = distinguishing assignment of node $i_{l(i)}$ */
10 $sign(i_{l(i)})$ /* = the sign of $d(i_{l(i)})$ */;
11 /* Initialize the example sets with null instances */
12 **for** each child $i_j = i_1, \ldots, i_{l(i)}$ of i
13 $S_{i_j} := \{\langle z_{i_j}, 0\rangle\}$;
14 **end for;**
15 **for** each example $\langle e_i, o(i)\rangle \in S_i$
16 **for** each child $i_j = i_1, \ldots, i_{l(i)}$ of i
17 /* Find the output $o(i_j)$ of the example at node i_j */
18 **if** i_j is a leaf, **then** $o(i_j) := e_{i_j}$
19 **else** if $d(i_j) = \perp$ **then** $o(i_j) := 0$
20 **else** $o(i_j)$:= MEMBER$(e_{i_j}/d(i_j)) \oplus sign(i_j)$;
21 $S_{i_j} := S_{i_j} \bigcup \{\langle e_{i_j}, o(i_j)\rangle\}$;
22 **end for;**
23 $InternalExamples := InternalExamples \bigcup \{\langle o(i_1) \ldots o(i_{l(i)}), o(i)\rangle\}$
24 **end for;**
25 **output** $\langle InternalExamples, S_{i_1}, \ldots, S_{i_{l(i)}}\rangle$;
26 **end** Decompose-Examples

27 /* MEMBER$_i$ is a simulated MEMBER oracle for f_i^D. */
28 MEMBER$_i(b_1, \ldots, b_{l(i)})$ = MEMBER$(c(i_1)/\ldots/c(i_{l(i)})/d(i)) \oplus sign(i)$, where,
29 **if** $b_j = 0$ **or** $d(i_j) = \perp$ **then** $c(i_j) = z_{i_j}$
30 **else** $c(i_j) = d(i_j)$

Figure 8. Algorithm for finding the example sets for the internal function at node i and for the external functions for the children of i.

function at each child of i. All the examples of f_i are collected and passed to the prediction algorithm for $\mathcal{R}^*$, called Learn-Internal-Function. The external functions that correspond to the children of i are learned by a recursive call to Learn-w-TSB with the corresponding examples. In what follows, we elaborate on how this is done.

6.2.1. Generating examples for the internal and external functions As we said before, the examples for the internal function at the current node and the external functions at its

children are generated by decomposing the examples of the external functions of the current node.

The procedure Decompose-Examples of Figure 8 takes the subtree T_i, the restrictions of instances onto T_i and their outputs at A_i, and the distinguishing assignments and their signs of children of i as inputs (lines 02–10). It starts by initializing the external example set of each child of i with null instances with 0 output (lines 11–14). The key to decomposing an example is the ability to infer the outputs of the examples at the children of the current node (lines 15–24 of Figure 8). Assume that the current node is i, and that $\langle e_i, o(i)\rangle$ is the current example for the external function at i. We want to infer the values at the child nodes i_1, i_2, etc. There are three cases to consider:

1. If i_j is a leaf node, then we can directly observe its value (line 18 of Figure 8).

2. If a distinguishing assignment for i_j cannot be found, then i_j is treated as dead. This means that the value at i_j for e_i can be assumed to be the same as the value for z_i, i.e., 0 (line 19).

3. If a distinguishing assignment $d(i_j)$ for i_j is known, then we can compute the value $o(i_j) = T_{i_j}(e)$ as follows: first, we set the leaves of T_{i_j} according to e_i and its context according to $d(i_j)$. Then we call the MEMBER oracle for the output and exclusive-OR the result with $sign(i_j)$ (line 20). This is justified by Proposition 1.4.

Once the value $o(i_j)$ for each of the children of i is inferred for an example $\langle e_i, o(i)\rangle$, the corresponding example of the internal function f_i is generated as $\langle o(i_1) \ldots o(i_{l(i)}),\ o(i)\rangle$ (line 23). Similarly, at each child i_j of i, an example $\langle e_{i_j}, o(i_j)\rangle$ is generated for the external function T_{i_j} and these examples are collected in the set S_{i_j} (line 21).

In our running example of Figure 5, Decompose-Examples gets the example set $\{\langle 0011, 1\rangle, \langle 0001, 0\rangle\}$, and $d(1) = \perp$, $sign(1) = 1$, $d(2) = 0011$, and $sign(2) = 0$ as inputs. It first initializes the external example sets of A_1 and A_2 with $\{\langle 00, 0\rangle\}$. When processing the example $\langle 0011, 1\rangle$, since $d(1) = \perp$, the value $o(1)$ of node 1 is assumed to be 0. To compute the value of A_2 for this example, the system asks a membership query on an instance where the leaves of T_2, i.e., B_3 and B_4, are set to 1 each according to the current example, and the context of T_2, i.e., B_1B_2, is set to 00 according to the distinguishing assignment $d(2)$. In this case, the query is answered with a 1. Since $sign(2) = 0$, the value of A_2 is determined to be $1 \oplus 0 = 1$, and the example $\langle 11, 1\rangle$ is added to the external example set of node 2. Since $A_1 = 0$, $A_2 = 1$, and $A_0 = 1$, the example $\langle 01, 1\rangle$ is added to $InternalExamples$ of node 0.

Repeating the same for the second example $\langle 0001, 0\rangle$, the system determines that $A_1 = 0$ and $A_2 = 0$. Hence, the example $\langle 01, 0\rangle$ is added to the set S_2, and $\langle 00, 0\rangle$ is added to $InternalExamples$ of node 0. Thus, Decompose-Examples returns with $InternalExamples = \{\langle 01, 1\rangle, \langle 00, 0\rangle\}$, $S_1 = \{\langle 00, 0\rangle\}$, and $S_2 = \{\langle 00, 0\rangle, \langle 11, 1\rangle, \langle 01, 0\rangle\}$.

6.2.2. Learning the internal function To learn the internal function, the main routine calls the learning algorithm for the basis class with random examples.

In general, the learning algorithm for the basis class might also have to make membership queries. Hence, we need to simulate a MEMBER$_i$ oracle for f_i using the MEMBER

oracle for the target function T^f, and the distinguishing assignments for the node i and its children. Unfortunately, this cannot always be done because some children D_i of i may not have known distinguishing assignments, and hence we cannot set these children to 1. Fortunately, to learn a function that is consistent with the training examples it suffices to learn the projection of f_i with respect to D_i, and that is what the algorithm does.

A membership query would ask the output of f_i for some settings of i's children. The value of a child i_j can be set to 0 by setting the leaves of T_{i_j} to 0's. The value of i_j can be set to 1 by setting the leaves of T_{i_j} according to the distinguishing assignment $d(i_j)$, if it is known, which is justified by Proposition 1.2. That leaves the case where $d(i_j)$ is not known, and the membership query requires the node i_j to be set to 1. In general, this is not possible to do without exhaustively searching for a distinguishing assignment for i_j. As hinted before, we resolve this problem by setting ourselves a less ambitious goal. Instead of learning the function f_i, we try to learn its projection with respect to D_i, the children of i with unknown distinguishing assignments. To learn $f_i^{D_i}$, we never need to generate a 1 at the nodes in D_i, because, by definition, $f_i^{D_i}$ ignores the inputs in D_i. For example, if f_i is a 4-input function and the first 2 input nodes are indistinguishable, $f_i^{D_i}(x_1, x_2, x_3, x_4) = f_i^{D_i}(0, 0, x_3, x_4) = f_i(0, 0, x_3, x_4)$. So we are justified in fixing the first two inputs to 0's, even though the membership query might want them to be 1s. This is shown in Figure 8, lines 27-30. Note that this approach assumes that the basis class is closed under projection, because otherwise it may not be possible to learn $f_i^{D_i}$ using the learning algorithm for the basis class.

Thus, to simulate the oracle MEMBER$_i$ for the projection of f_i, the program sets the context of T_i according to $d(i)$. To set A_{i_j} to 0, it sets the leaves of T_{i_j} to 0's; to set it to 1, it sets them according to $d(i_j)$ if $d(i_j) \neq \perp$, otherwise it sets them to 0's (lines 29-30 of Figure 8). After setting the leaves of each T_{i_j} as above and the context of T_i according to $d(i)$, it finds the value of A_i, by asking a membership query on the target function, and finding the exclusive-OR with $sign(i)$ (line 28).

In our running example, since $d(1) = \perp$, the external function at A_1 is set to the constant function **0**. Then, Learn-w-TSB is recursively called on T_2 with the example set $\{S_2 = \langle 00, 0\rangle, \langle 11, 1\rangle, \langle 01, 0\rangle\}$. Since both the children of node 2 are leaves, no distinguishing assignments are needed for them. Eventually, Learn-Internal-Function is called on the example set S_2 at node 2, which learns some function which is consistent with it, say, AND.

Learn-Internal-Function is then called with the example set $\{\langle 01, 1\rangle, \langle 00, 0\rangle\}$ for the internal function at node 0. In general, Learn-Internal-Function gets many examples and generalizes them using some learning algorithm that possibly asks membership queries. Any membership queries asked by this algorithm would be reduced to membership queries on the target using the null assignment for the leaves of T_1 and either the null assignment or the distinguishing assignment for node 2. Suppose that the learning algorithm asks a membership query for the internal function f_0 on the input 11. Since the learning algorithm only needs to learn the projection of f_0 with respect to A_1, the output of the projection would be the same for 11 as it is for 01. Hence it sets B_1B_2 to 00 and sets B_3B_4 to 11 (as dictated by $d(2)$). The membership query on the target is now answered with 1, which is passed on to the learning algorithm after taking an exclusive-OR with $sign(0) = 0$. From the internal example set and the membership queries, assume that Learn-Internal-Function learns the function f_0 to be simply A_2 (which is consistent with the input examples $\langle 01, 1\rangle$

and $\langle 00, 0\rangle$). Learn-w-TSB puts this together with the constant function **0** learned at node 1 and the AND function learned at node 2, and returns the function B_3B_4. Note that this is consistent with all the input examples $\langle 0111, 1\rangle$, $\langle 1011, 1\rangle$, and $\langle 0101, 0\rangle$, although it is not the target function.

6.3. Proof of Correctness

In this section we prove that our algorithm learns the space of functions $T^{\mathcal{I}}$ induced by the basis class $\mathcal{I}$ over the determination tree T if a few technical conditions hold. The plan of the proof is as follows. We first prove an important lemma that shows that given a consistent compression algorithm for $\mathcal{R}^*$, Learn-w-TSB outputs a hypothesis that is consistent with its training set and the determination tree, and is at most polynomially longer than any well-behaved target formula. Using this lemma and Lemmas 1 and 2, we then show that if a closure $\mathcal{R}^*$ of the language $\mathcal{R}$ of the basis class under some operations is predictable, then $T^{\mathcal{R}}$ has a cwm-algorithm. We then appeal to the general result of Blumer et al. (1989) to show that given a reasonably small sample, this algorithm pwm-predicts the space of Boolean functions induced by the basis class over any given determination tree in time polynomial in various parameters including the size of the target function.

LEMMA 3 *Let T be a determination tree and T_0^f be a well-behaved target formula represented in $T^{\mathcal{R}}$. Let* Learn-Internal-Function *be a deterministic, consistent, compression algorithm for $\mathcal{R}$. If* Learn-Internal-Function *needs to ask membership queries, we require $\mathcal{R}$ to be closed under projection. If* Learn-w-TSB *is called on a subtree T_i with a set of training examples S_i of T_i^f, then the formula it returns, T_i^g, is consistent with S_i with probability at least $1 - Size(T_i)\delta'$, where $Size(T_i)$ is the number of internal nodes in T_i and $1 - \delta'$ is the lower bound on the confidence that* Learn-Internal-Function *outputs a formula which is consistent with its input training set.*

Proof: We prove the result by structural induction on i. Note that the examples in S_i consist of the restrictions of the input instances to T_i, and their outputs at node i.

The lemma is trivially true if i is a leaf node, since Learn-w-TSB always returns the identity function, which is the correct function for the leaf nodes. Assume that the lemma is true for all subtrees of T_i. That is, for all subtrees T_{i_k} of T_i assume that the hypothesis generated ($T_{i_k}^g$) is consistent with S_{i_k} (with the given probability). We prove that the hypothesis generated by Learn-Internal-Function for T_i (namely, T_i^g) is consistent with S_i, again with the given probability. In other words, we will show that for each e_i in S_i, the target and the hypothesis external functions, T_i^f and T_i^g, agree on the classification of e_i, i.e., $T_i^f(e_i) = T_i^g(e_i)$, with a high probability.

We do this by appealing to the fact that T_i can be decomposed into an internal function of the external functions of its children, since for any example e_i: $T_i(e_i) = f_i(T_{i_1}(e_{i_1}), ..T_{i_{l(i)}}(e_{i_{l(i)}}))$.

Let $D_i = \{i_{j1}, \ldots, i_{jk}\}$, be the children of i in left to right order for which the distinguishing assignments have not been found. Then, the learned external function for these nodes is the constant function **0**. We begin by showing that for x_i= $z_{i_{jk}}/\ldots/z_{i_{j1}}/e_i$, the target classifies e_i and x_i in the same way, i.e., $T_i^f(e_i) = T_i^f(x_i)$ for all e_i in S_i.

Notice that the algorithm finds a distinguishing assignment $d(i_j)$ for a node i_j only if $T(d(i_j)) \neq T(z_{i_j}/d(i_j))$. This is so because in the algorithm Distinguish-Children of Figure 7 (line 15), $Temp = T(z_{i_j}/e_i/d(i))$, and $o(i) \oplus sign(i) = T_i(e) \oplus sign(i) = T(e_i/d(i))$ by Proposition 1.5. Hence any distinguishing assignment found by the algorithm must be correct.

Recall that Distinguish-Children replaces the variables under any child of i for which a distinguishing assignment has not been found with 0's (Figure 7, line 19) for all examples before processing the subsequent children of i. Since Distinguish-Children had failed to find distinguishing assignments for nodes in D_i when called on the subtree T_i, it would have successively replaced the variables under the nodes in D_i with 0's for all examples. Since the test in line 15 of Figure 7 had failed for all nodes in D_i, for all examples $\langle e_i, o \rangle$ in S_i,

$$\begin{aligned} T^f(e_i/d(i)) &= T^f(z_{i_{j1}}/e_i/d(i)) = T^f(z_{i_{j2}}/z_{i_{j1}}/e_i/d(i)) \\ &= \ldots = T^f(z_{i_{jk}}/\ldots/z_{i_{j1}}/e_i/d(i)). \end{aligned}$$

By substituting $z_{jk}/\ldots/z_{j1}/e_i$ for x_i and e_i for y_i in Proposition 1.3, we have

$$T^f(z_{i_{jk}}/\ldots/z_{i_{j1}}/e_i/d(i)) = T^f(e_i/d(i)) \Longrightarrow T_i^f(z_{i_{jk}}/\ldots/z_{i_{j1}}/e_i) = T_i^f(e_i)$$

Since we have just showed that the left hand side of the above implication is true, it follows that

$$T_i^f(e_i) = T_i^f(z_{i_{jk}}/\ldots/z_{i_{j1}}/e_i).$$

Since $z_{i_{jk}}/\ldots/z_{i_{j1}}/e_i = x_i$,

$$T_i^f(e_i) = T_i^f(x_i) = f_i(T_{i_1}^f(x_{i_1}), \ldots, T_{i_{l(i)}}^f(x_{i_{l(i)}})).$$

Since for any node $i_j \in D_i$, $x_{i_j} = z_{i_j}$ and $T_{i_j}^g$ is the constant function $\mathbf{0}$ (see line 15 of Figure 6), we have

$$T_{i_j}^f(x_{i_j}) = 0 = T_{i_j}^g(e_{i_j}) \text{ for all } i_j \in D_i \tag{1}$$

Let us define a "proper run" to be one where the learned external functions of *all* children of i on which Learn-w-TSB was called are consistent with their training examples. By the inductive hypothesis, the probability of this occurrence is at least $1 - \delta' \sum_{i_j} Size(T_{i_j})$. Hence, in a proper run,

$$T_{i_j}^f(x_{i_j}) = T_{i_j}^f(e_{i_j}) = T_{i_j}^g(e_{i_j}) \text{ for all } i_j \notin D_i \tag{2}$$

Combining the equations 1 and 2, we conclude that in a proper run, for all children i_j of i, $T_{i_j}^f(x_{i_j}) = T_{i_j}^g(e_{i_j})$.

From the example e_i, if $l(i)$ is the number of children of i, Learn-w-TSB generates the example $\langle T_{i_1}^f(x_{i_1}) \ldots T_{i_{l(i)}}^f(x_{i_{l(i)}}), T_i^f(e_i) \rangle$ for Learn-Internal-Function. If Learn-Internal-Function is called with a confidence parameter δ', it finds a consistent function

g_i with probability at least $1 - \delta'$. Hence, when there was a proper run, with probability at least $1 - \delta'$,

$$\begin{aligned} T_i^f(e_i) &= f_i(T_{i_1}^f(x_{i_1}), \ldots, T_{i_{l(i)}}^f(x_{i_{l(i)}})) = g_i(T_{i_1}^f(x_{i_1}), \ldots, T_{i_{l(i)}}^f(x_{i_{l(i)}})) \\ &= g_i(T_{i_1}^g(e_{i_1}), \ldots, T_{i_{l(i)}}^g(e_{i_{l(i)}})) = T_i^g(e_i) \end{aligned} \quad (3)$$

To complete the induction step, we have to compute the total probability that the above might fail. It is the sum of the probability of the run being improper, and the probability of Learn-Internal-Function failing to find a consistent hypothesis. This is at most $\delta'(1 + \sum_{i_j} Size(T_{i_j})) = \delta' Size(T_i)$. Hence with a probability at least $1 - Size(T_i)\delta'$, T_i^g is consistent with the examples in S_i.

Now, let us consider the case where Learn-Internal-Function asks membership queries. Answering membership queries for a function f_i requires the program to set the inputs of f_i to required values, and read the output at node i. When a distinguishing assignment and sign of node i are known, its *output* on any assignment x is $T(x_i/d(i)) \oplus sign(i)$ by Proposition 1.4. Since $T(x_i/d(i))$ and $sign(i) = T(z_i/d(i))$ are both known by membership queries to the target function, the output at node i is easy to infer (Figure 8, lines 27–28). Setting the *inputs* of f_i appropriately is more of a problem. When a distinguishing assignment for a node i_j is known, then it can be set to 0 or 1 by setting the leaves of T_{i_j} according to z or $d(i_j)$ respectively. However, when a distinguishing assignment for i_j is not known, it cannot be set to 1. Hence, when $\mathcal{R}^*$ is only pwm-predictable, instead of learning the function f_i, Learn-Internal-Function is called to learn its projection with respect to nodes D_i for which the distinguishing assignments have not been found. If $\mathcal{R}$ is closed under projection, the projection $f_i^{D_i}$ of f_i is in $\mathcal{R}$ and has the same size as f_i. Thus Learn-Internal-Function can efficiently find a function g_i that is consistent with $f_i^{D_i}$ on the input examples. Since $f_i^{D_i}$ and f_i behave the same way when the inputs in D_i are set to 0, we can still conclude the following for any $x_i = z_{i_{jk}}/\ldots/z_{i_{j1}}/e_i$, with probability $1 - \delta'$, as in Equation 3:

$$\begin{aligned} T_i^f(e_i) &= f_i(T_{i_1}^f(x_{i_1}), \ldots, T_{i_{l(i)}}^f(x_{i_{l(i)}})) = f_i^{D_i}(T_{i_1}^f(x_{i_1}), \ldots, T_{i_{l(i)}}^f(x_{i_{l(i)}})) \\ &= g_i(T_{i_1}^f(x_{i_1}), \ldots, T_{i_{l(i)}}^f(x_{i_{l(i)}})) = g_i(T_{i_1}^g(e_{i_1}), \ldots, T_{i_{l(i)}}^g(e_{i_{l(i)}})) = T_i^g(e_i) \end{aligned}$$

Note that any membership query for $f_i^{D_i}$ asked by Learn-Internal-Function is answered correctly by setting the children of i in D_i to 0's. This is so because, by the definition of projection, $f_i^{D_i}$ treats all inputs in D_i as 0's. All other children i_j of i are set to 0 or 1 by setting the leaves under the corresponding subtrees according to z_{i_j} or $d(i_j)$ respectively (Figure 8, lines 29-30).

The rest of the proof is similar to the case without membership queries. □

We are now ready to prove the existence of compression algorithms for function spaces induced by determination trees under some conditions.

THEOREM 3 *Let T be any determination tree and $\mathcal{R}$ be a representation language for some internal function space. If (a) the closure of $\mathcal{R}$ under variable negation is pac-predictable, or (b) the closure of $\mathcal{R}$ under projection and variable negation is pwm-predictable, then there is a cwm-algorithm for $T^{\mathcal{R}}$ that accepts T and the labeling of its leaf nodes with attributes as input.*

Proof: We prove that Learn-w-TSB is a cwm-algorithm for $T^{\mathcal{R}}$.

Let $\mathcal{R}'$ be the closure of $\mathcal{R}$ defined as in the theorem statement. By Lemma 1, since $\mathcal{R}'$ is pac (pwm) predictable, its closure under complement, say $\mathcal{R}^*$, has a compression (cwm) algorithm. Let Learn-Internal-Function be such an algorithm. Note that the projection of the complement of a formula (or a function) with respect to a set of variables is the complement of the projection of that formula (or the function) with respect to the same variables. Similarly, a variable negation of the complement of a formula (or a function) is the complement of a variable negation of the formula (or the function). This means that $\mathcal{R}^*$ is also closed under projection and variable negation, since $\mathcal{R}'$ itself is closed under these operations.

Let f_i denote the internal target function at node i represented in $\mathcal{R}$, and let T_i^f denote the corresponding external target function. From Lemma 2, without loss of generality, we may assume that the internal function formulas f_i are represented in $\mathcal{R}^*$, and that T_0^f is well-behaved.

Since $\mathcal{R}^*$ has a cwm-algorithm and T_0^f is well-behaved, by Lemma 3, the output of Learn-w-TSB is consistent with its training example set with probability at least $1 - \delta' Size(T)$, where δ' is the confidence parameter of Learn-Internal-Function. To find a function consistent with examples at the root node with probability at least $1 - \delta$, we can set $\delta' = \delta / Size(T)$.

To complete the proof that Learn-w-TSB is a cwm-algorithm for $T^{\mathcal{R}}$ we have to show that (a) it outputs a polynomially evaluatable hypothesis whose size is polynomial in the number of its inputs n and the size of the target function $|T^f|$, and (b) it runs in time polynomial in n, $|T^f|$, $1/\delta$ and the number of input examples m.

We first show that it outputs a hypothesis of required size and complexity of evaluation. Recall that from Lemma 2 when the original internal target functions f_i are replaced with their complements and variable-negations, their sizes in $\mathcal{R}^*$ are increased by at most $l(i)+1$, where $l(i)$ is the number of children of i. Replacing a function by its projection in $\mathcal{R}^*$ does not increase its length. Let g_i be the internal function returned by Learn-Internal-Function at node i, and let T_i^g denote the corresponding external function. To ensure that $T_i^g(z_i) = 0$ also holds for all *learned* non-root internal functions g, Learn-w-TSB always includes a null instance in their examples (see the procedure Decompose-Examples in Figure 8, lines 12–14). By Schapire's theorem, $|g_i|$ must be a polynomial function p of the size of the new target function $|f_i|+l(i)+1$ and the number of its inputs $l(i)$, i.e., $p(|f_i|+l(i)+1, l(i))$. The size of the output hypothesis is the sum of the sizes of g_i at each node. If n_I is the number of the internal nodes and n is the number of leaves of the determination tree,

$$
\begin{aligned}
|T^g| = \sum_{i=0}^{n_I-1} |g_i| &= \sum_{i=0}^{n_I-1} p(|f_i| + l(i) + 1, l(i)) \\
&\leq n_I p(\max_i(|f_i| + l(i) + 1), \max_i l(i))
\end{aligned}
$$

Since n_I, the number of internal nodes of the determination tree, $\max_i(|f_i| + l(i) + 1)$, and $\max_i l(i)$ are all smaller than $|T^f| + n + 1$, the above bound is polynomial in the size of the target function and the number of inputs.

To show that the output hypothesis T^g can be evaluated in time polynomial in n and $|T^f|$, it suffices to notice that it forms a tree circuit where the nodes i contain functions g_i, which, by Schapire's Ockham-converse result, can be evaluated in time polynomial in $|f_i| + l(i) + 1$ and $l(i)$. Hence the output hypothesis T^g can be evaluated in the required time in bottom-up fashion.

We now show that Learn-w-TSB makes at most a polynomial number of queries in the size of the target function, the number of its inputs n, and the number of examples, and runs in time polynomial in these parameters.

In the worst case, Distinguish-Children generates one query for each example and each non-root internal node being considered (lines 11–14 of Figure 7). If n_I is the number of internal nodes of the determination tree, this totals to $m(n_I - 1)$ queries. Decompose-Examples infers the value of each child i_j of the current node i for each example e, which is done using one query on $e_{i_j}/d(i_j)$ (lines 15–20 of Figure 8). Since this is done on each non-root internal node and each example, in the worst case, it makes another set of $m(n_I - 1)$ queries, giving a total of $2m(n_I - 1)$ queries.

By Schapire's result, there exist polynomials r and q such that Learn-Internal-Function, when called at a node i with $l(i)$ children and given m distinct examples of a function f_i, runs for at most $r(m, |f_i| + l(i) + 1, l(i), \log \frac{1}{\delta'})$ time and asks at most $q(m, |f_i| + l(i) + 1, l(i), \log \frac{1}{\delta'})$ membership queries. Each query of Learn-Internal-Function is simulated by Learn-w-TSB with one membership query. δ' can be set to δ/n_I for our purpose. Hence, the total number of queries by Learn-w-TSB is bounded by $2m(n_I-1)+\sum_{i=0}^{n_I-1} q(m,|f_i|+ l(i)+1, l(i), \log \frac{n_I}{\delta})$. This can be easily seen to be a polynomial in the required parameters.

Since the run time of Learn-Internal-Function includes the time for its queries, the total computation done by Learn-w-TSB can be divided into asking $2m(n_I - 1)$ queries and running Learn-Internal-Function at each internal node. Since each query is on an assignment of size n, we have to multiply the number of queries by n to get the time complexity for asking queries. To this, we have to add the time complexity of running Learn-Internal-Function at all internal nodes.

Hence the time complexity of Learn-w-TSB is

$$O(mn(n_I - 1) + \sum_{i=0}^{n_I-1} r(m, |f_i| + l(i) + 1, l(i), \log \frac{n_I}{\delta})),$$

which is polynomially bounded.

This completes the proof of all requirements for Learn-w-TSB to be a cwm-algorithm for $T^{\mathcal{R}}$. □

We can consider the following parallel version of our algorithm. Observe that the distinguishing assignments for any node can be found after finding the distinguishing assignments for its parent and its previous siblings. Applying this rule recursively to the node's parent and siblings, we notice that we can find the distinguishing assignment of any node after finding the distinguishing assignments for all earlier siblings of that node, and for all its ancestors and their earlier siblings. All these nodes must be processed strictly sequentially. If d is the depth of the determination tree, the above argument implies that, we have to

find the distinguishing assignments for at most $d \max_{i=0}^{n_I-1} l(i)$ nodes sequentially. After finding the distinguishing assignments for all nodes, Learn-Internal-Function can be run in parallel for all nodes. Hence the parallel complexity of our algorithm is:

$$O(mnd \max_{i=0}^{n_I-1} l(i) + \max_{i=0}^{n_I-1} r(m, |f_i| + l(i) + 1, l(i), \log \frac{n_I}{\delta})).$$

We are now ready to state and prove our main result on pwm-predictability of function spaces induced by a determination tree.

THEOREM 4 *Let $\mathcal{R}$ be a representation language for some function space and T be any determination tree with a fixed mapping of its leaves to the input attributes given to the learner. If (a) the closure of $\mathcal{R}$ under variable negation is pac-predictable, or (b) the closure of $\mathcal{R}$ under projection and variable negation is pwm-predictable, then, there is a pwm-algorithm for $T^{\mathcal{R}}$.*

Proof: The proof of this theorem is by the direct application of Theorem 1. We already showed that Learn-w-TSB is a cwm-algorithm for $T^{\mathcal{R}}$. To convert it into a pwm-algorithm, call it with a sample of size $\frac{1}{\epsilon}(|T^g| \ln C + \ln \frac{2}{\delta})$, where $|T^g|$ is the size of its output hypothesis, and C is the number of distinct primitive symbols used in the language of g. Since $|T^g|$ is upper-bounded by a polynomial function of the size of the target function $|T^f|$ and the number of nodes in the determination tree, we can substitute this bound in the above formula, generate a sample of that size, and then call Learn-w-TSB on it. The resulting algorithm is a pwm-prediction algorithm for the class $T^{\mathcal{R}}$. □

A variety of basis functions satisfy the requirements of our theorem, including linear threshold functions, k-DNF (disjunctions of conjunctions of at most k literals), and k-CNF (conjunctions of disjunctions of at most k literals), which are closed under variable negation and are pac-learnable. In the next section, we apply our program to Rivest's k-decision lists or k-DL, i.e., lists of k-length clauses, each of which is an IF-THEN statement with 0 or 1 outcome (Rivest, 1987). The class of k-decision lists is closed under variable negation, pac-learnable, and properly includes k-DNF and k-CNF (Kohavi & Benson, 1993). Monotone DNF-formulas are pwm-predictable (Angluin et al., 1993), closed under projection, but not under variable negation; however, they do satisfy our condition because unate formulas (the closure of monotone DNF-formulas under variable negation) are pwm-predictable (Angluin et al., 1993). Similarly, we can use Bshouty's (1993) algorithm for learning decision trees. However, there are some interesting pwm-predictable classes such as conjunctions of Horn clauses for which the closure under variable negation does not hold, and hence our theorem does not apply.

It appears that both the oracles used by our algorithm are necessary to learn with tree-structured determinations. As shown by Pitt and Warmuth (1990), learning arbitrary Boolean formulas is reducible to learning with tree-structured bias from classified random examples. Since learning arbitrary Boolean functions is as hard as inverting one-way functions (Kearns & Valiant, 1994), learning with tree-structured bias with random examples alone is believed to be intractable. To see that tree-structured bias cannot be implemented with membership queries alone, consider the class of Boolean functions that output 1 on at most one input and output 0 on all others. There are $2^n + 1$ possible Boolean functions over n variables in this class, and they can be represented with a tree-structured bias.

However, this class cannot be learned with membership queries alone, because the program does not get any information unless it chances upon the single input that gives rise to an output of 1. In the worst case, this involves trying each possible input, which goes beyond the polynomial bound. Given the EXAMPLE oracle, however, the learning algorithm that generates a small sample of examples and returns a Boolean function which outputs 1 only on the positive example in the sample (if any) and 0 on all other inputs can be shown to be a successful PAC-learner. In particular, if the sample size is at least $\frac{1}{\epsilon} \ln \frac{1}{\delta}$, it is highly likely that either we are going to see the single positive example, or it has a probability less than ϵ so that the constant function **0** is a good approximation to the target. In other words, the EXAMPLE oracle helps the learner by providing information about the example distribution which enables it to focus on the commonly occurring instances and ignore the infrequent instances.

7. Experimental Results

The above algorithm was implemented as a program called TSB-2. TSB-2 needs an induction program to learn the internal functions of the determination tree. We created two versions:

- TSB-$2^{k-\mathrm{DL}}$ uses a version of Rivest's (1987) algorithm k-DL to learn the internal functions as k-decision lists. A k-decision list is a list of clauses, where each clause is of the form (t_i, b_i). t_i is a conjunction of at most k literals, and b_i is the output bit. For the last clause, t_i is always true. The k-decision list acts like a COND-statement in LISP. The value of the k-decision list for an instance is the value of b_i for the first clause for which t_i evaluates to true for that instance. Since the final t_i is always true, the k-decision list uniquely evaluates to 0 or 1 on any instance.

  k-DL is in fact a family of algorithms generated by varying k. The set of examples for which the term t_i evaluates to 1 are said to be "covered" by the clause (t_i, b_i). Starting with clauses of length 1, k-DL considers increasingly longer clauses until k-length clauses to find a clause that correctly classifies all examples covered by it. It selects that clause, removes all covered examples from the set and iterates to select the next clause. k-DL can be shown to be a consistent learning algorithm for k-decision lists.

- TSB-2^{ID3} uses ID3 to learn the internal functions as decision trees (Quinlan, 1986). A decision tree is a rooted binary tree in which each internal node in the tree tests for the value of a single input attribute of the input example. If its value is true, the right subtree is processed and if the value is false, the left subtree is processed. Each leaf of the tree is labeled with the class of the example that reach that leaf. Decision trees are strictly more general than k-decision lists for any fixed k and are also closed under variable negation and projection. ID3 is a top-down algorithm that uses information gain as a heuristic evaluation function to decide which attribute to select at a given point to split the example set into two depending on its value (Quinlan, 1986). It then calls itself recursively on the two mutually disjoint sets of examples thus created and builds the left and right subtrees. Unlike the k-DL algorithm, which is guaranteed to learn the space of k-DL functions, ID3 is not guaranteed to efficiently learn the space of decision trees, although it works well in practice.

7.1. Comparison with knowledge-free induction

In our first set of experiments, TSB-2$^{k-\mathrm{DL}}$ and TSB-2$^{\mathrm{ID3}}$ were applied to learn functions that are consistent with a determination tree of depth 2. The root node has 5 children, each of which has 5 children, which form the leaves of the tree. Hence there are a total of 25 input attributes. For concreteness, we can think of the target concept as a combinatorial Boolean circuit, with depth $= 2$, fan-in $= 5$, and fan-out $= 1$. The target functions were constructed using the above tree with the internal functions drawn from 3-decision lists of 12 clauses, where each clause has a term t_i of exactly 3 randomly chosen literals and a randomly chosen output bit b_i.

Our experiments compare TSB-2 with a knowledge-free induction program on the same set of examples. We used ID3 (Quinlan, 1986) and k-DL (Rivest, 1987) as our knowledge-free induction systems, because of their simplicity and availability. They also worked better than any other system we tried on this domain including a more sophisticated version of ID3 called C4.5 and a propositional version of FOIL (Quinlan, 1990). Of course, the comparison is not intended to decide which algorithm is better "in general" but rather to show that when the relevant background knowledge is available, TSB-2 can exploit it more effectively to design useful experiments and to yield higher accuracy after seeing fewer examples.

Each training episode consisted of the following steps:

- Generate a target function consistent with the determination tree.
- Generate and label 200 examples, drawn from a uniform distribution on the input space.
- Divide the examples into two equally-sized sets, reserving one for testing.
- Apply the learning algorithm to training sets of sizes from 0 to 100 in increments of 10, testing the result on the test set after each application.

Results were averaged over 50 training episodes.

We ran two experiments using the above determination tree — the first one with TSB-2$^{\mathrm{ID3}}$ and the second with TSB-2$^{3-\mathrm{DL}}$. In each experiment, for each target concept and training sample, we stored the queries asked by TSB-2 and the responses. We compared TSB-2$^{\mathrm{ID3}}$ with ID3. Note that the target functions in T^{3-DL} are not representable by 3-decision lists. Since the determination tree is of depth 2, they are, however, representable by 6-decision lists. So we compared TSB-2$^{3-\mathrm{DL}}$ with 6-DL.

To provide a fair comparison, we provided both the random examples and the membership query examples to the knowledge-free programs. The versions of the programs that use the query examples are called ID3-Q and 6DL-Q respectively. As control, we also ran ID3 and 6-DL with the same number of training examples chosen randomly. These versions of ID3 and 6-DL are called ID3-R and 6DL-R respectively. Both versions of ID3 were run without pruning, because pruning worsened the performance. There is no pruning step for k-DL.

Figures 9 and 10 show the results of the first experiment with TSB-2$^{\mathrm{ID3}}$. Figure 9 shows the relationship between the size of the random training sample and the number of queries generated by TSB-2. The total number of queries rose from 0, with no random examples, to about 547, with 100 random examples. As the theoretical analysis predicts (see the proof of Theorem 3), the number of queries grows linearly with the number of

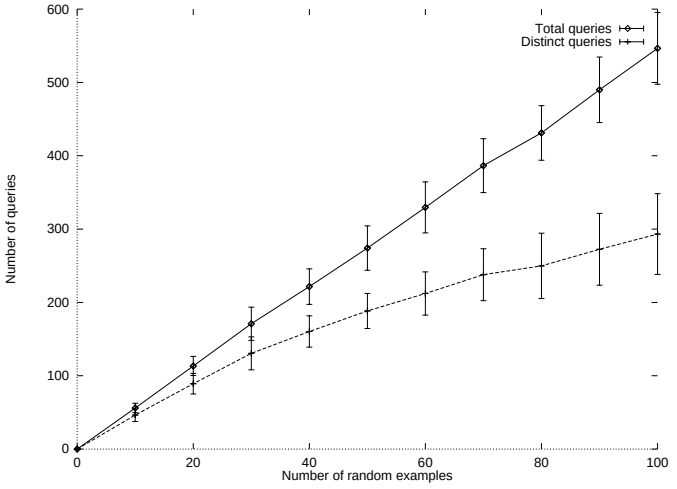

Figure 9. Growth of queries with random examples for TSB-2^{ID3}. The top curve shows the total number of queries asked as the number of random examples is varied. The bottom curve shows the total number of distinct queries asked for the same number of random examples.

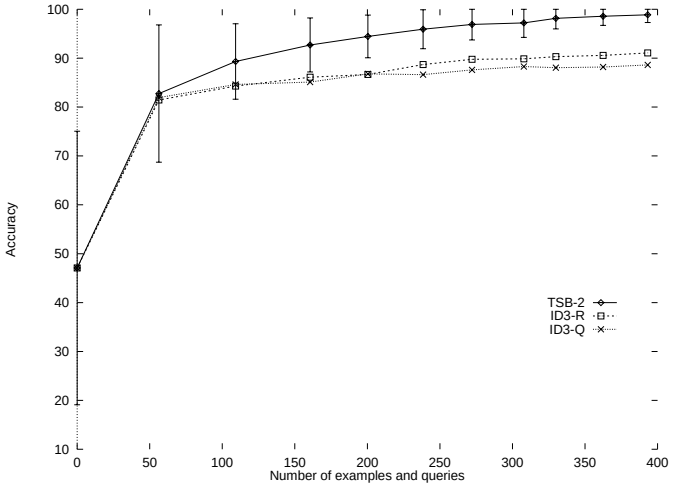

Figure 10. Learning curves for TSB-2^{ID3} and ID3. On the X-axis is the number of training examples and queries, and on the Y-axis is the percentage accuracy. ID3-R is the learning curve of ID3 when all its training examples are randomly chosen. ID3-Q is the learning curve of ID3 when it is given the same training examples and answers to queries as used by TSB-2^{ID3}.

examples, and is approximately equal to the number of non-root internal nodes (5) times the number of random examples. This is a little more than 50% of the worst-case bound, which is $2m(n_I - 1)$. However, using a hash table that caches the results of previous queries reduced the number of queries considerably. The number of distinct queries rose

from 0 to 293 with 100 random examples. More interestingly, this curve appears to flatten out as the number of random examples approaches 100, suggesting that perhaps there is an upper bound on the total number of distinct queries asked. (Since the instance space has 2^{25} examples, the flattening is not due to exhausting the instances.) The error bars denote 1 standard deviation error on either side of the average for 50 random trials.

Figure 10 shows the accuracy on the test sample for TSB-2^{ID3} and ID3 plotted against the sum of the number of random examples and the average number of distinct queries made by TSB-2 for that many random examples. Each successive point on the two curves of Figure 10 represents a random training sample of size 10 more than the previous point. The error bars are only shown for TSB-2^{ID3}, to avoid clutter, and denote 1 standard deviation error on either side. ID3's error bars are much larger, even at the end of training. During the training, ID3-Q's average performance increased from 47.1% to 88.62% with 393 training examples. In contrast, TSB-2's performance increased from 47.1% to 98.9% with the same training, showing that TSB-2 exploits its tree-structured bias well. Interestingly, ID3-Q did not perform better than ID3-R, which chose all its examples randomly. ID3-R achieved an accuracy of 91.1 % at the end of learning, which is slightly higher than ID3-Q's performance.

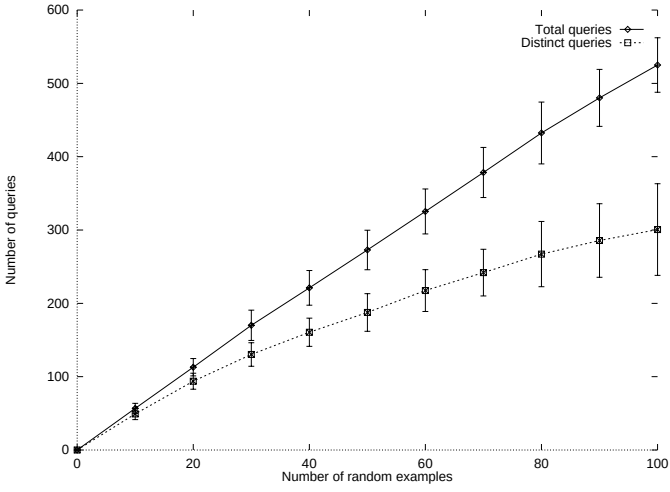

Figure 11. Growth of queries with random examples for TSB-$2^{3-\mathrm{DL}}$. The top curve shows the total number of queries asked as the number of random examples is varied. The bottom curve shows the total number of distinct queries asked.

Figures 11 and 12 show the learning curves for the same target function space for TSB-$2^{3-\mathrm{DL}}$ and 6-DL. The results here were similar to those of the first experiment. The average number of queries reached 525 with 100 examples, and the average number of distinct queries reached 301. TSB-$2^{3-\mathrm{DL}}$'s accuracy rose from 41% to 99.4% for a total of 401 examples and distinct queries. 6DL-Q and 6DL-R also started from an initial performance of 41%. 6DL-Q's final performance on the same set of training examples and distinct queries as used by TSB-2 was 86.6%, and 6DL-R's final performance on the same number of random examples was 93.2%.

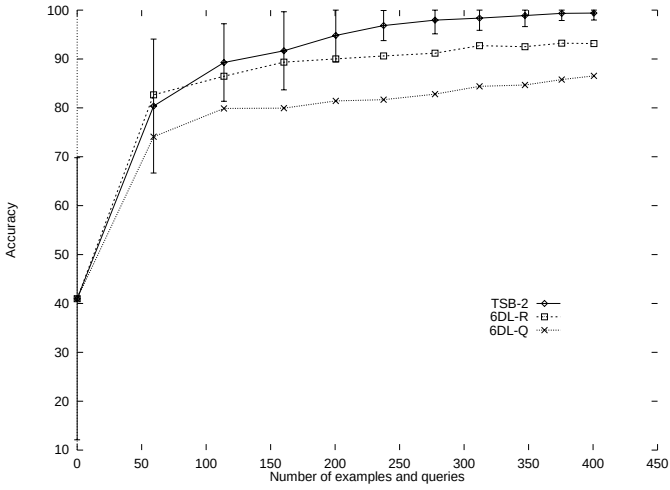

Figure 12. Learning curves for TSB-$2^{3-\mathrm{DL}}$ and 6-DL. On the X-axis is the number of training examples and queries, and on the Y-axis is the percentage accuracy. 6DL-R is the learning curve of 6-DL when all its training examples are randomly chosen. 6DL-Q is the learning curve of 6-DL when it is given the same training examples and answers to queries as used by TSB-$2^{3-\mathrm{DL}}$.

As before, we can see that TSB-$2^{3-\mathrm{DL}}$ performs significantly better than 6DL-R. What is perhaps more prominent than in the case of ID3 is the significantly superior performance of 6DL-R compared to 6DL-Q. This difference is due to the fact that for 6DL-Q (unlike 6DL-R), the training distribution is significantly different from the test distribution. Thus, 6DL-R's advantage comes from the fact that it is trained on examples that are more similar to the test examples. This also suggests that an example is "good" only with respect to the learning algorithm, and good performance involves more than simply knowing what queries to ask.

Our experiments clearly illustrate the usefulness of queries in learning certain types of functions, and the effectiveness of tree-structured determination knowledge in allowing the learner to decide which queries are most useful. The differences between the performances of ID3-Q and ID3-R and between 6DL-Q and 6DL-R suggest that the queries themselves did not provide an advantage to TSB-2—the advantage comes from being able to use them in the context of background knowledge to answer specific questions about the true function.

As can be expected, the above experimental results of TSB-2 were also much better than the worst-case theoretical bounds. For the above internal function space of 3-decision lists over 5 variables, there are a total of $\sum_{i=0}^{3}\binom{5}{i}2^i = 131$ different conjunctive terms. Our target functions actually consist of only 12 such terms, out of which the last one must be true. Hence, there are $131!/120! < 131^{11}$ possible left-hand sides of clauses. Since the right hand side of each clause can be either 0 or 1, there are at most $131^{11}2^{12}$ different internal target functions. Since 3-DL does not necessarily find a minimal decision list, the actual number of hypotheses we should consider is actually higher. Even if we ignore this factor, which considerably increases the number of training examples needed, the total number of target functions is $(2 \times 262^{11})^6$ since there are 6 internal functions. Since both

decision lists and k-decision lists are closed under complement and variable negation, there is not going to be any further increase in the size of the output hypothesis space. Using the bound in the proof of Theorem 4, and using $\epsilon = \delta = 0.1$, we get $4,003$ random examples and, in the worst case, $2 \times 4,003 \times 5 = 40,030$ queries. Both of these values are at least two orders of magnitude larger than the experimentally determined values for the same accuracy.

7.2. Scaling with the depth of determination tree

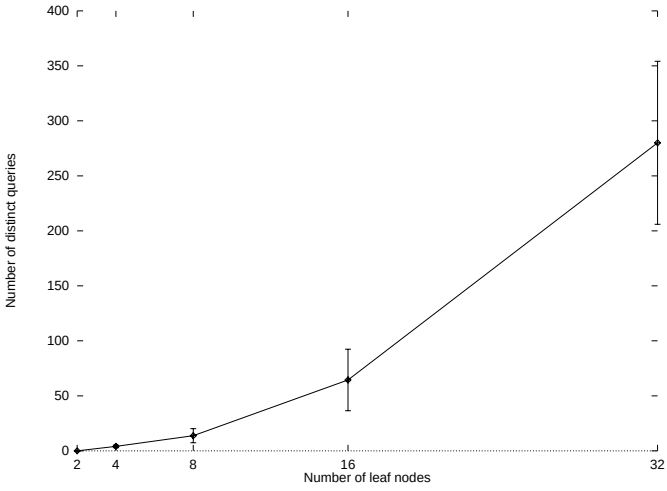

Figure 13. Number of queries asked by TSB-2 as a function of the tree size. The X-axis shows the number of random input examples, which is equal to the number of leaves of the tree. The Y-axis shows the total number of distinct queries asked. The error bars denote 1 standard deviation on either side.

The second set of experiments examines how the performance of TSB-2 scales with the depth of the determination tree when compared to ID3 and k-DL. For this set of experiments, we chose to use full, binary determination trees of depths varying from 1 through 5. The number of random training examples for TSB-2 varied in proportion to the number of nodes in the tree — from 2 in the case of depth–1 tree to 32 in the case of depth–5 tree. The target internal functions were generated uniformly randomly from 10 of the 16 2-input functions. (The other 6 functions ignore one or both inputs.) TSB-2 used k-DL with $k = 2$ to learn the internal functions. We ran 30 trials on each tree, randomly varying the target function. Since the internal functions are representable as 2-decision lists, the target functions are representable by $2d$-decision lists, where d is the depth of the determination tree. Hence, we compared TSB-2 with ID3 and with $2d$-DL. Both of these programs were given the same number of random training examples as the number of examples and queries used by TSB-2 on the same target function. We measured the performance of the learned functions on 100 test examples which were generated independently randomly.

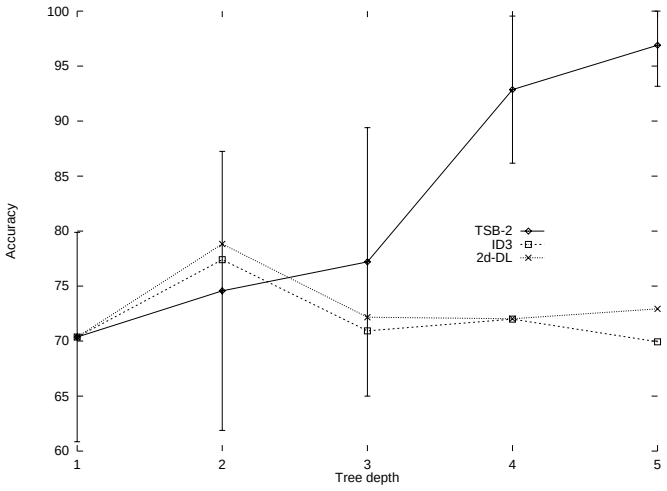

Figure 14. Scaling of performance with the depth of the determination tree for TSB-$2^{2-\mathrm{DL}}$, ID3 and $2d$-DL. On the X-axis is the depth of the determination tree, and on the Y-axis is the percentage accuracy. The error bars are only shown for TSB-2 and represent 1 standard deviation on either side. TSB-$2^{2-\mathrm{DL}}$ improves its performance with depth and corresponding increase in the training sample size, while ID3 and $2d$-DL deteriorate.

Figure 13 shows the number of distinct queries asked by TSB-2 as a function of the number of leaves of the determination tree. Recall that the number of random training examples is increased linearly with the number of leaves. Since the number of queries is proportional to the product of the number of random examples and the number of internal nodes, it varies quadratically with the number of leaves. This quadratic nature of the relationship is apparent in Figure 13. In our experiment, the number of distinct queries increased from 0 when the number of leaf nodes is 2 to 280 when the number of leaf nodes is 32.

Figure 14 shows the performances of TSB-$2^{2-\mathrm{DL}}$, ID3 and $2d$-DL as a function of the depth of the determination tree. On the X-axis is the depth, and on the Y-axis is the accuracy of the various learned functions on the test examples. It must be kept in mind that these are not "learning curves," but curves that show how the test set accuracy changes by an increase in the size of the target function and a corresponding increase in the training set size. Since the number of examples is increased in proportion to the tree size, we expect the asymptotic performance to be more or less constant for TSB-2. Hence, it is the relative performance that is most informative here. While they all perform similarly when the tree is of depth 1 and there are no hidden nodes, differences appear as the depth increases. While TSB-$2^{2-\mathrm{DL}}$ steadily improves its accuracy from 70.4% at depth 1 to 96.9% at depth 5, ID3 and $2d$-DL peak at depth 2 with accuracies of 77.4% and 78.8% respectively. Their accuracies drop to 69.9% and 72.9% respectively when the depth is increased to 5.

The results clearly illustrate that deeper trees are harder to learn for both ID3 and $2d$-DL. For $2d$-DL, since the length of the clauses required to represent the target function increases linearly with the depth of the tree, there are exponentially many clauses to consider, and in the worst case the number of clauses in the hypothesis increases exponentially with depth. Since the sample complexity is proportional to the hypothesis size, it is also going to

increase exponentially with depth. A decision tree representation may be even larger than the corresponding decision list representation since it suffers from the so called "replication problem," which is due to the fact that the decision tree can only test a single attribute at any node (Pagallo & Haussler, 1990). As a result, even ID3 performs poorly as the depth of the determination tree is increased.

In summary, the experiments of this section suggest that when there are such deep relationships between the variables in a domain as exhibited by the determination tree, knowledge-free induction is hopelessly difficult, and it is much better to make these relationships explicit and exploit this knowledge in the learning algorithm.

8. Determination Graphs

In the previous sections, we considered the problem of learning from prior knowledge in the form of tree-structured determinations. In this section, we consider the case where the determinations are structured as a directed acyclic graph (dag). We call this graph a *d-dag* (for determination dag) and this bias a dag-structured bias. The d-dag is a directed acyclic graph over a set of attributes, where each attribute may participate in directly determining more than one attribute. In contrast with the previous section, here we show a negative result, namely that the pwm-prediction with dag-structured bias is hard under standard cryptographic assumptions. Our result is representation-independent in that it holds for any language choice to represent the final function as long as it is efficient to evaluate the function on any example.

Formally, a d-dag is a directed acyclic graph $G = (A, Pred, L, R)$, where A, the set of nodes of the d-dag, represents attributes, and $Pred$ represents the directed edges of the d-dag as a set of predecessor relationships between nodes. $Pred(i)$ is the set of predecessor nodes of i, and the attributes represented by them *determine* the attribute A_i that corresponds to node i. The nodes $L \subseteq A$ are the input nodes that have no predecessors, and the node $R \in A$ is the root node, which represents the target attribute.[7]

While the fan-out of a determination tree is 1, it is at least 2 for a d-dag. In other words, there is at least one node with more than one parent in a d-dag. Given a set of basis functions $\mathcal{I}$, the set of functions induced by $\mathcal{I}$ over the d-dag G is defined similarly to that induced by $\mathcal{I}$ over the tree structure, and is denoted by $G^{\mathcal{I}}$. The task is to learn these functions from examples and membership queries. To keep our results representation-independent, we assume that the learning algorithms do not output a function, but simply predict the output on an arbitrary test example in polynomial time.

For our proofs, we will be using the framework of prediction-preserving reducibility with membership queries introduced by Angluin and Kharitonov (1995). We proceed as follows. We reduce the problem of learning arbitrary Boolean functions in n variables from examples and membership queries to one of learning with a fixed dag-structured bias, where only the leaf nodes may have multiple parents. Since learning arbitrary Boolean functions with examples and membership queries is cryptographically hard, our results imply that learning with d-dags can also be as hard. Our results hold when the fan-out of the d-dag is at least 2 and fan-in is at least 3 or vice versa. In both cases, the internal functions are limited to $\{\mathrm{AND}, \mathrm{NOT}, \equiv\}$, where $\equiv$ gate outputs 1 if and only if all its inputs are the same.

In the framework of prediction-preserving reducibility with membership queries, we start with two function spaces $\mathcal{F}$ and $\mathcal{F}'$ and define a reduction that will allow us to convert a polynomial-time pwm-prediction algorithm M' for $\mathcal{F}'$ into a polynomial-time pwm-prediction algorithm M for $\mathcal{F}$ (Angluin & Kharitonov, 1995). If $\mathcal{F}$ is known to be hard to predict, so is $\mathcal{F}'$. In our case, $\mathcal{F}$ is the set of all Boolean functions over n variables, and $\mathcal{F}'$ is the set of functions induced by $\{\text{AND}, \text{NOT}, \equiv\}$ over some d-dag.

Let D be the domain of functions in $\mathcal{F}$ and D' be the domain of functions in $\mathcal{F}'$.

In order to carry out a prediction-preserving reduction with membership queries from the function space $\mathcal{F}$ to $\mathcal{F}'$, we have to show the existence of the following three mappings.

- A mapping g of representations of functions in $\mathcal{F}$ to representations of functions in $\mathcal{F}'$.
- A mapping r of training or test examples in D to training or test examples in D'.
- A mapping h of examples in D' queried by M' to examples in D queried by M.

The rationale for these mappings is as follows. To learn a function $y \in \mathcal{F}$, M mimics the steps of M' learning the function $g(y) \in \mathcal{F}'$. Whenever M' requests a random example, M requests a random example from D, transforms it using r and gives it to M'. The mappings g and r should be such that the value of y on any input $x \in D$ must be the same as the value of $g(y)$ on $r(x)$. h maps the examples in D' queried by M' to examples in D, queried by M. h could be an inverse of r; however it might so happen that some elements in D' have no inverses. In all such cases, h outputs a fixed answer ($\perp$ in our case). The trick is to make sure that this answer is consistent with $g(y)$ for all target functions y. While h and r must be computable in polynomial time, g need not be, but we must ensure that it maps to functions whose representation sizes are bounded by a polynomial in the sizes of the original functions.

The reader is referred to (Angluin & Kharitonov, 1995) for a more formal description of prediction-preserving reducibility. We now state our main theorem which is based on a prediction preserving transformation of learning arbitrary boolean functions to functions induced by fixed dags.

THEOREM 5 *Pwm-prediction of Boolean formulas over the basis* $\{$*AND, NOT,* $\equiv\}$ *consistent with a given d-dag is as hard as pwm-prediction of arbitrary Boolean formulas, even for d-dags with fan-in* $= 3$ *and fan-out* $= 2$ *or fan-in* $= 2$ *and fan-out* $= 3$.

Proof: Please see the Appendix.

THEOREM 6 *Pwm-prediction of Boolean formulas over the basis* $\{$AND, NOT, $\equiv\}$ consistent with a given d-dag is as hard as inverting one-way functions, even for d-dags with fan-in $= 3$ and fan-out $= 2$ or fan-in $= 2$ and fan-out $= 3$.

Proof: Follows immediately from Theorem 5 and the result of Kearns and Valiant (1994), which shows that learning Boolean functions is as hard as inverting certain one-way functions, e.g., breaking the RSA cryptosystem or factoring Blum integers. □

It was shown in (Bshouty et al., 1995) that exact learning of read-twice formulas with equivalence and membership queries is hard when the internal functions include $\{\equiv$, OR, NOT, AND$\}$. Our result is stronger because it shows that even pwm-prediction, which is sometimes easier than exact learning, is also as hard *even* when the underlying determination-dag (or skeleton) of the target formula is given.

9. Related Work

Our work is most closely related to the work on Boolean read-once formulas in computational learning theory (Angluin et al., 1993; Hancock, 1991; Bshouty et al., 1992, 1995). The more theoretically inclined reader is referred to these excellent papers, particularly the last one, which subsumes the others. All of this work is based on exact learning using equivalence queries as well as membership queries (Angluin, 1988). We call this model the "exact learning model" from now on. Equivalence queries are in the form of "is the target function the same as g?" If g is the target function, the teacher answers 'yes'; otherwise the teacher responds with a counterexample, i.e., an example where g and the target function disagree. Bshouty et al. (1992, 1995) describe a learning algorithm that exactly identifies the determination tree (skeleton) and the target function from membership and equivalence queries for constant fan-in or symmetric internal functions. A symmetric Boolean function gives the same output for any permutation of its inputs. The determination tree is identified by using the queries and justifying assignments to partition the set of variables into those that are descendants of some internal node and those that are not. Once such a nontrivial partition is found, the algorithm can be called recursively to induce trees on the two sets of variables. After the tree T is found, the functions are identified by running multiple copies of Learn-Internal-Function on the internal nodes of the tree until all of them request counterexamples. When this happens, their corresponding hypotheses g_i are plugged into the determination tree T and the composite hypothesis T^g is used to ask an equivalence query. Either the hypothesis is correct or a counterexample to this composite function is returned. If there is a counterexample to this composite hypothesis, then it is used to generate a counterexample to at least one of the internal functions g_i, and the cycle repeats. Since the space of internal functions is assumed to be exactly learnable with a polynomial number of queries, this process terminates in polynomial time under conditions similar to our Theorem 4.

In summary, unlike our algorithm, the algorithm of Bshouty et al. also learns the skeleton of the read-once formulas, but is restricted to the union of symmetric and constant fan-in basis functions (Bshouty et al., 1995). For example, arbitrary k-decision lists cannot be represented as compositions of symmetric or constant fan-in functions, and hence their algorithm cannot be used when the basis includes k-decision lists. It is an open question whether the skeletons of read-once formulas over such bases can be efficiently learned from examples.

The model of exact learning with equivalence and membership queries is theoretically attractive because it sidesteps the need for probability distributions. Positive results of this model directly yield positive results under the pwm-predictability model using standard transformations (Angluin, 1988). However, they are not equivalent because a negative result in the equivalence plus membership query model does not imply that pwm-learning is not possible. More specifically, Lemma 5 of (Bshouty et al., 1995) is the exact-model counterpart of our Theorem 4. Although our theorems are based on similar insights, neither of them implies the other. Their lemma assumes that the internal functions are exactly learnable, a strictly stronger assumption than ours that they are pwm-predictable. Our theorem also does not require closure under projection for the internal function class if it is

pac-predictable. On the other hand, our theorem only guarantees pwm-predictability, while theirs guarantees exact learning, which is strictly stronger.

While the exact learning model is theoretically convenient, the pwm-predictability model yields more natural and efficient implementations, which is the main concern of our paper. Our emphasis is on building a bridge between the theoretical work on learning with membership queries and the practical concerns of effectively exploiting prior knowledge in learning. We are able to achieve this by presenting a theoretically justified algorithm that effectively solves the problem of learning from tree-structured prior determination knowledge. We also provide empirical support for the work on read-once formulas by showing that our implementation achieves smaller error rates from the same data when compared to knowledge-free induction algorithms. Our results and those in (Bshouty et al., 1995) suggest that the experimental learning community needs to pay attention to theoretical work on learning from membership queries; the performance of practical induction systems can perhaps be improved significantly compared to an approach based purely on random examples.

The ideas of Bshouty et al. (1994) were adapted to the PAC learning framework by Hancock, Golea and Merchand (1994). They proposed an algorithm that learns multi-layer nets of perceptrons from queries and examples assuming that there is no overlap between the receptive fields of the input units and that the output of each unit is connected to at most one other unit. These assumptions impose a tree-structure on the network and constrain the basis functions to be linear threshold functions. Their algorithm learns the network structure using recursive partitioning of the input variables as in (Bshouty et al., 1995), and learns the linearly separable internal functions using linear programming. The algorithms of (Bshouty et al., 1995) for learning the skeletons of the read-once formulas have worst-case complexities of the order of $O(n^{k+1})$ for k-TSB and $O(n^6)$ for TSB with symmetric basis functions. Improving the complexities of these algorithms is essential to yield practical implementations. Such improved programs could be used in concert with ours to provide a more complete learning system.

TSB-2 is a successor of TSB-1, which was restricted to constant fan-in determination trees (Tadepalli, 1993). Other than TSB-1, there were two previous implementations of tree-structured bias (Getoor, 1989) (see also (Russell, 1989)). However, both of them have only had limited success. One of them assumes that the Boolean functions at each node are monotone, yielding a very simple polytime algorithm with membership queries. The other uses the determination tree to prestructure a neural network, using a "mini-network" to represent each of the internal functions. Unfortunately, using backpropagation in such a network is too slow to handle trees with more than a handful of inputs. On the other hand, this problem should not arise with TSB-2 because of its ability to learn each of the internal functions locally.

There has been a considerable amount of research on using prior knowledge in learning. For example, FOCL (Pazzani & Kibler, 1992), and Grendel (Cohen, 1992) learn general relational concepts and are capable of making use of background knowledge expressed as first-order Horn clauses. As suggested by Cohen, determinations can be expressed as overgeneral Horn clauses (Cohen, 1991). For example, the fact that A and B determine C can be represented by a set of 8 Horn-clauses of the form $value(A, V_A) \wedge value(B, V_B) \rightarrow value(C, V_C)$, where V_A, V_B, and V_C take values from $\{0, 1\}$ in all combinations. The

learning problem can be seen as finding a correct subset of these clauses, using examples to guide the process. However, neither FOCL nor Grendel can exploit the tree structure because they always learn clauses for the root of the determination tree in terms of the observable leaf nodes. Such rules collapse the tree structure to two levels, thus destroying the possibility of fast convergence. Consider, for example, learning a 2-CNF function, $G = (A_0 + B_0)(A_1 + B_1)(A_2 + B_2)\ldots$. This function can be represented succinctly as a determination tree with three levels. The root node computes AND, the middle nodes compute OR, and the leaf nodes are observable inputs. However, both Grendel and FOCL learn G as a set of Horn clauses of the form $A_0A_1A_2\ldots \rightarrow G; B_0A_1A_2\ldots \rightarrow G; \ldots B_0B_1B_2\ldots \rightarrow G$. Unfortunately, this representation is exponentially larger than the target's original representation and hence both FOCL and Grendel converge very slowly when learning concepts of this kind.

Work in theory refinement seeks to address the above issue by discouraging radical revisions of the theory and by requiring that the learning program makes only *minimal* changes to the input theory. We ran a theory revision program, called NEITHER (Baffes & Mooney, 1993), on our problem. NEITHER is a propositional theory refinement program that accepts theories described as Horn clauses, and refines them until all the training examples are correctly classified. We gave NEITHER a domain theory which is the equivalent of the tree-structured bias used in our experiments of Section 7, while constraining the basis functions to monotone 3-DNF. The monotone DNF constraint allowed us to easily convert the determination theory into an overgeneral, propositional Horn theory. For each determination in the tree of the form "$\{A_1, \ldots, A_l\}$ determines A," all possible Horn-clauses of the form "$t_i \rightarrow 1$" were generated, where t_i is a conjunction of at most 3 positive literals in $A_1, \ldots, A_l$, and these were given to the system as prior knowledge. The task of the system was simply to select a subset of the clauses that is consistent with the input examples. We gave the random examples and the queries used by TSB-2 as input to NEITHER.

Somewhat surprisingly, NEITHER worked better without the Horn theory than with the theory![8] This illustrates the difficulty of building robust knowledge-based learning systems—simply adding prior knowledge seems not to be enough.

10. Conclusions and Future Work

This paper applies some of the theoretical tools developed in computational learning theory to inductive learning problems addressed in experimental machine learning. Our algorithm is inspired by the methodology of controlled experimentation in experimental science. It is based on the notion of distinguishing assignments, which simultaneously play the roles of making a hidden node (theoretical variable) observable by setting up an appropriate context, as well as controlling that node (variable) by changing its causal antecedents appropriately.

We also saw that membership queries can be effectively utilized in learning when the function space is constrained by declarative knowledge in the form of a tree of determinations. This illustrates the power of tree-structured bias in constraining learning, and the effectiveness of special-purpose learning programs like TSB-2 in exploiting this bias. Our results with ID3 and k-DL show that merely knowing which queries to ask does not improve performance. Using queried examples in the way that random examples are used can in fact worsen the performance due to the differences in the two distributions. Our results with

NEITHER also show that having the necessary prior knowledge and using it in a uniform way may not be sufficient to guarantee successful learning.

The determinations we considered are deterministic in that the value of the output feature is a fixed function of the values of the input features. We can generalize this to stochastic domains by assuming that the values of the input features only determine the *probability* of the output feature. A stochastic determination graph can thus be interpreted as the graph structure of a Bayesian network (Pearl, 1988). Deterministic versions of Bayesian networks, called "symbolic causal networks" by Darwiche and Pearl (1994), are similar to determination graphs except that they allow some "exogenous" propositions to introduce noise into the functional relationship defined by the determination. Although there are a few results in learning Bayesian networks from data when all the variables are observable (Cooper & Herskovits, 1992), learning with hidden variables is much harder (see (Russell, Binder, Koller, & Kanazawa, 1995) and (Friedman, 1997) for recent algorithms). As yet, no work has been done on using membership queries in such algorithms, nor on ascertaining the computational complexity of the problem. It may be possible to extend our algorithms to learn tree-structured Bayesian networks with hidden variables. Our negative result on learning with determination-graphs immediately implies that learning the parameters of a graph-structured Bayesian network with hidden variables is hard even when the network structure is known and the teacher can answer membership queries.

It would be interesting to extend our results to include noisy and incomplete oracles, since these often provide a better model for real-world learning problems; any of several recent noise models might be used. We also foresee that real-world learning might require the introduction of restricted classes of non-Boolean functions at the nodes of the tree. There has been some work in computational learning theory on learning arithmetic read-once formulas that may be useful here (Bshouty, Hancock, & Hellerstein, 1995). We expect that the basic insights used in TSB-2, including distinguishing assignments, will carry over into the more general setting.

We can also consider applying TSB-2 to *diagnosis* tasks, which are not normally thought of as learning tasks since prior knowledge provides such a strong constraint. Here, the structure of the system maps to the tree or dag structure of the target function, while the components of the system map to the internal functions. In some cases, the problem may have some decomposability so that we can test some components of the device separately. This corresponds to the availability of a MEMBER oracle for those components. Knowing the correct functionality of the device might also make the learning problem easier by constraining the hypothesis space so that only minor deviations from the correct function are considered, e.g., the so-called "single fault" assumption. Components that are known to be working can be modeled as internal function spaces containing only a single function. In general, there is no reason why the internal learning function cannot also use any prior knowledge that is available.

On the theoretical side, the most interesting open problem is to learn the tree structure under the weakest possible assumptions about the internal (basis) functions. The positive results so far have been restricted to the union of constant fan-in and symmetric basis functions (Bshouty et al., 1995). The question of whether this class can be extended to include all learnable functions that obey our closure properties is still open. It is possible that there are basis functions for which learning the tree-structure is hard under the conditions

of Theorem 4. If that turns out to be the case, it implies that the prior knowledge in the form of determination tree improves not only the sample complexity of learning, but also the computational complexity.

Recently, it has been shown that read-twice DNF and CNF formulas are learnable from random examples and membership queries (Aizenstein & Pitt, 1991; Hancock, 1991; Pillaipakkamnatt & Raghavan, 1995), while the status of general read-twice formulas over the basis {AND, OR} is still open. These results might be useful in closing the gap between our negative and positive results, for example, by studying the learnability of determination graphs of fan-out = fan-in $= 2$ for various basis functions.

Acknowledgments

This work was supported in part by National Science Foundation grants IRI-9111231 (to the first author) and IRI-9058427 and IRI-9634215 (to the second author). We thank Hussein Almuallim, Tom Dietterich, Tom Hancock, Sridhar Mahadevan, Balas Natarajan, Vijay Raghavan, Chandra Reddy, and Manfred Warmuth for interesting discussions on this topic. We are grateful to Paul Baffes and Ray Mooney for providing the code for NEITHER and to Ross Quinlan for making C4.5 available. We thank David Haussler, Lisa Hellerstein, Leonard Pitt, and the anonymous reviewers of this paper for their thorough and thoughtful comments, which significantly improved the paper.

Appendix

THEOREM 5: *Pwm-prediction of Boolean formulas over the basis {AND, NOT, $\equiv$} consistent with a given d-dag is as hard as pwm-prediction of arbitrary Boolean formulas, even for d-dags with fan-in $= 3$ and fan-out $= 2$ or fan-in $= 2$ and fan-out $= 3$.*

Proof: The proof is based on a two-part prediction-preserving transformation of arbitrary Boolean functions to functions induced by $\{\mathrm{AND}, \mathrm{NOT}, \equiv\}$ over a fixed d-dag.

First, using a transformation similar to that introduced by Pitt and Warmuth (1990), we reduce learning arbitrary Boolean functions to learning functions over a fixed determination tree with the basis $\{\mathrm{AND}, \mathrm{NOT}, \equiv\}$, where each input attribute of the original function is repeated many times in the leaves of the tree. We then transform that tree to a d-dag with fan-out > 1 at the leaf nodes, so that the target attribute value is 0 if all the copies of the same input attribute do not have the same value. We now describe this transformation in detail (see Figure A.1).

Let y be the target Boolean formula over the input attributes $x_1, \ldots, x_n$ that uses NOT, AND, and OR. We first describe a mapping g that converts y into a target function representation for a fixed d-dag. y can be viewed as a tree. Let $|y|$ be the number of nodes in that tree. Results in circuit complexity (Wegener, 1987) imply that y can be computed by a Boolean circuit made-up of any sufficiently powerful 2-input gate types with a circuit of depth at most $d = \lfloor b \log |y| \rfloor$, where b is a constant independent of y. We use the AND gate and the "consensus gate" which computes the function $\equiv$. Given a constant 0 as an optional input, these two gates are sufficiently powerful to implement any other Boolean function over two inputs. This is because $\mathrm{NOT(x)}$ can be implemented as $\equiv (x, 0)$ and OR can be implemented as $\mathrm{NOT(AND(NOT(x), NOT(y)))}$ by DeMorgan's law. Hence the above

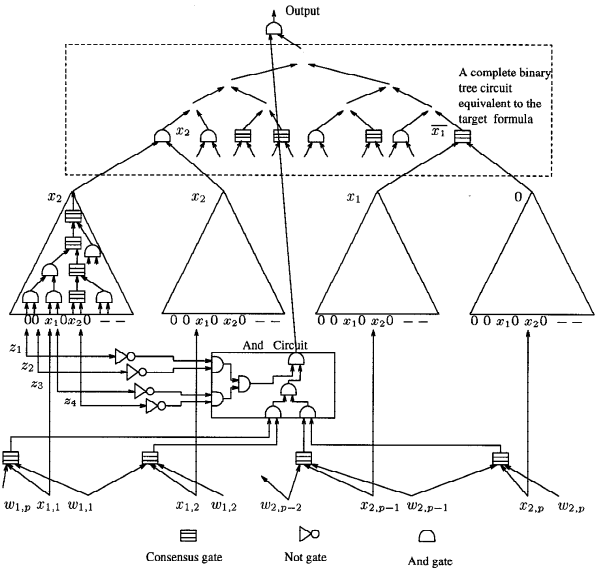

Figure A.1. Reduction of an arbitrary Boolean formula to a fixed d-dag of fan-out = 2 and fan-in = 3. All nodes except the leaves have a fan-out of 1. All gates other than those in the bottom layer have a fan-in of 2.

result holds for this set. The tree can be made into a complete tree of depth exactly d by iteratively replacing each of the shallow leaves labeled x with $\mathrm{AND(x, x)}$. In Figure A.1, the dashed rectangle on the top shows the complete binary tree which is equivalent to the target formula.

The inputs of the resulting tree T are from the set $\{x_0 = 0, x_1, \ldots, x_n\}$. T is not yet a determination tree, because the leaves of T do not occur in any fixed order and each input is repeated many times. Let n' be the smallest power of 4 that is as large as $2(n+1)$. Replace each leaf l of the tree T with a complete binary tree T_l of height $\log n' = 2h'$, whose leaves are labeled $\langle x_0 = 0, 0, x_1, 0, x_2, 0, \ldots, x_n, 0\rangle\langle 0\rangle^{n'-2(n+1)}$. If l is labeled with x_i in the tree T, then the internal functions of T_l can be chosen from $\{\mathrm{AND}, \equiv\}$ such that its root evaluates to x_i. This can be done by having the consensus gates at all the internal nodes in the path to the x_i, and AND gates at all other nodes. Each consensus gate works like NOT in this case, and since there are an even number of such gates in any such path of height $2h'$, the root computes x_i. This is shown by the triangles in the middle of Figure A.1.

The new tree T' is a complete binary tree of depth $d + 2h'$ whose leaves are labeled with fixed input attributes including 0. However, each attribute is repeated many times, and many input attributes are fixed as 0. To make all the attributes distinct, we first replace each jth occurrence of 0 by a new variable z_j. We then replace each j^{th} occurrence of each x_i by a variable $x_{i,j}$. Now, all nodes are distinctly labeled; but all "legal" inputs for the original Boolean formula y should have $z_j = 0$ for all j, and $x_{i,1} = x_{i,2} = \ldots = x_{i,p}$ for all i. We want to make the new circuit output the same value as the original formula y for all legal

inputs of y, but output 0 for all illegal inputs. The part of the circuit below the triangles in Figure A.1 does this by checking that the inputs to the circuit are legal.

To check that all z_js are 0's, we first negate them using NOT gates, and then AND the results with AND gates of fan-in =2, as shown in the bottom rectangle of Figure A.1. This can be done without increasing the depth of the circuit. To check that $x_{i,1} = x_{i,2} = \ldots = x_{i,p}$ for all i, we borrow an idea from (Bshouty et al., 1992) and introduce an auxiliary variable $w_{i,j}$ for each $x_{i,j}$. We use 3-input consensus gates to compute $\equiv (w_{i,p}, x_{i,1}, w_{i,1})$, $\equiv (w_{i,1}, x_{i,2}, w_{i,2})$, etc., and then AND the results with a binary tree of 2-input AND gates. Since the $w_{i,j}$'s for successive j's are shared between successive consensus gates, this circuit outputs 1 if and only if $x_{i,1} = x_{i,2} = \ldots = x_{i,p}$ for all i. We then AND the outputs of the above two sub-circuits, getting a circuit which outputs 1 on all legal inputs for the original formula y and 0 on all illegal inputs (shown as the output of the bottom rectangle of Figure A.1). Finally, we AND this output with the output of T' (shown at the top of Figure A.1). This circuit outputs 0 for all illegal inputs of the target formula y and computes y on all legal inputs. Call the d-dag that corresponds to this circuit, G.

It is easy to see that the fan-out of all internal nodes of G is at most 1. The fan-out of each input z_j is 2 because each such node is connected to a NOT gate and to a single gate in the tree T'. Each input $x_{i,j}$ is connected to one consensus gate and to another gate in the tree T', and hence has a fan-out of 2. Each input $w_{i,j}$ also has a fan-out of 2 since it is only connected to two successive consensus gates. The fan-in of all gates except the consensus gates in the bottom of Figure A.1 is at most 2. The consensus gates have a fan-in of 3. Thus, G is a d-dag of fan-out = 2 and fan-in = 3 that can be instantiated with AND, NOT, and consensus gates so that it computes the target formula y for all legal inputs and outputs 0 for all other inputs. The size of G, as measured by the number of gates, is polynomial in the size of the original function y, and the number of inputs n, because its height is $O(\log 2(n+1) + \log |y|)$, and it has a single output node and a fan-in of 3.

To create a circuit of fan-out = 3 and fan-in = 2, simply remove all the inputs $w_{i,1}, \ldots, w_{i,p}$ and implement $x_{i,1} = \ldots = x_{i,p}$ for all i by AND($\equiv (\mathrm{x_{i,1}}, \mathrm{x_{i,2}}), \equiv (\mathrm{x_{i,2}}, \mathrm{x_{i,3}}), \ldots \equiv (\mathrm{x_{i,p-1}}, \mathrm{x_{i,p}})$), using 2-input AND and consensus gates. Each $x_{i,j}$ is now input to two consensus gates and one gate in the tree T', resulting in a circuit with fan-out = 3 and fan-in = 2. Call the d-dag that corresponds to this circuit, H.

The rest of the proof applies to the d-dag G. The above mapping g described how to transform a target boolean function y into an instantiation of the d-dag G with specific gates. We define the remaining two mappings—r for the random instances, and h for the membership queries—necessary to complete the prediction-preserving transformation as follows:

- r maps any random instance $x = \langle x_1, \ldots, x_n \rangle$ to

$$\langle \langle 0, 0, x_1, x_1, 0, x_2, x_2, \ldots, x_n, x_n, 0 \rangle \langle 0 \rangle^{n'-2(n+1)} \rangle^{2^d}.$$

  Here the 0's correspond to the inputs z_1, z_2, and so on. The two successive x_1's correspond to $x_{1,j}$ and $w_{1,j}$ respectively. Similarly, the two successive x_2's correspond to $x_{2,j}$ and $w_{2,j}$, and so on.

- If a membership query x' is of the form

$$\langle\langle 0, 0, x_1, x_1, 0, x_2, x_2, \ldots, x_n, x_n, 0\rangle\langle 0\rangle^{n'-2(n+1)}\rangle^{2^d},$$

i.e., if there is an x such that $r(x) = x'$, then $h(x') = x = \langle x_1, \ldots, x_n\rangle$; else $h(x') = \perp$.

It is easy to see that a program M' that learns the function space induced by $\{\mathrm{AND}, \mathrm{NOT}, \equiv\}$ over G can be used to simulate a learner M for the Boolean function space $\mathcal{F}$. The target function $y \in \mathcal{F}$ is mapped to $g(y)$ by instantiating G with specific gates as described above. Whenever M' requests a training example, we generate a training example for M, and if it receives $\langle x_i, y_i\rangle$ as an example, we provide $\langle r(x_i), y_i\rangle$ to M'. By the way we defined the mapping g, $y_i = y(x_i) = g(y)(r(x_i))$. Any membership queries x' asked by M' will be answered with a 0 if $h(x') = \perp$, and otherwise by asking a membership query on $h(x')$ to M's MEMBER oracle. Note that $g(y)$ maps x' to 0 whenever $h(x') = \perp$, i.e., the circuit G outputs 0 on all "illegal" inputs. To predict the output of the target function on a random example x, x is transformed to $r(x)$ and the value of the learned function on $r(x)$ is returned. If M can learn an approximation of $g(y)$, then our simulation can learn an approximation to y, since the probability weights of their error regions are exactly the same.

The proof for H with fan-out = 3 and fan-in = 2 is similar. □

Notes

1. This assumption is fairly natural. It corresponds to assuming that each attribute in the domain is determined by at most k other attributes.
2. It should be noted that the read-once property depends on what "basis" functions (primitives) one is allowed to use. $\overline{A}B + A\overline{B}$ is not read-once, but is trivially read-once if one is allowed to use exclusive-or as a primitive.
3. We thank Lisa Hellerstein for providing a counter-example for the projection case.
4. Note that the projection of a function fixes the irrelevant inputs to 0, while the restriction of an assignment sets the irrelevant inputs to $*$.
5. The experiment needs to be done on a large group of patients rather than a single one in the medical domain to offset the effects of large variations among the different patients and study their aggregate behavior.
6. One obvious lemma is needed to use the example oracle in establishing learnability, namely, that if the original example distribution is stationary then the oracle for the subtree also has a stationary distribution.
7. Notice that here, as in determination trees, we assume that such graphs (or trees) to be nonredundant, in that determinations that are implied by other determinations are not explicitly represented in the graphs (or trees). For example, if P and Q determine R, which in turn determines S, P and Q are not connected to S, even though they do determine S by implication.
8. Since NEITHER's performance without the theory was similar to that of ID3, we only showed the results of comparisons with ID3 in Section 7.

References

Aizenstein, H., & Pitt, L. (1991). Exact learning of read-twice DNF formulas. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, (pp. 170–179). Washington, D.C.: IEEE Computer Society Press.

Angluin, D. (1988). Queries and concept learning. *Machine Learning*, *2*(4), 319–342.

Angluin, D., Hellerstein, L., & Karpinski, M. (1993). Learning read-once formulas with queries. *Journal of the ACM*, *40*(1), 185–210.

Angluin, D., & Kharitonov, M. (1995). When won't membership queries help?. *Journal of Computer and Systems Sciences*, *50*(2), 336–355.

Baffes, P., & Mooney, R. (1993). Extending theory refinement to m-of-n rules. *Informatica*, *17*, 387–397.

Blumer, A., Ehrenfeucht, A., Haussler, D., & Warmuth, M. (1989). Learnability and the Vapnik-Chervonenkis dimension. *Journal of the ACM*, *36*(4), 929–965.

Bshouty, N., Hancock, T., & Hellerstein, L. (1992). Learning Boolean read-once formulas with arbitrary symmetric and constant fan-in gates. In *Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory*, (pp. 1–15). Pittsburgh, PA: ACM Press.

Bshouty, N., Hancock, T., & Hellerstein, L. (1995). Learning Boolean read-once formulas over generalized bases. *Journal of Computer and Systems Sciences*, *50*(3), 521–542.

Bshouty, N., Hancock, T., & Hellerstein, L. (1995). Learning arithmetic read-once formulas. *SIAM Journal on Computing*, *24*(4), 706–35.

Cohen, W. (1991). The generality of overgenerality. In *Proceedings of the Eighth International Workshop on Machine Learning*, (pp. 490–494). Evanston, IL: Morgan Kaufmann.

Cohen, W. (1992). Compiling prior knowledge into an explicit bias. In *Proceedings of the Ninth International Conference on Machine Learning*, (pp. 102–110). Austin, TX: Morgan Kaufmann.

Cooper, G., & Herskovits, E. (1992). A Bayesian method for the induction of probabilistic networks from data. *Machine Learning*, *9*(4), 309–348.

Davies, T. (1985). Analogy. Informal note IN-CSLI-85-4, CSLI, Stanford University.

Friedman, N. (1997). Learning belief networks in the presence of missing values and hidden variables. In *Proceedings of International Conference on Machine Learning*, (pp. 125–133). Nashville, TN: Morgan Kaufmann.

Getoor, L. (1989). The instance description: How it can be derived and the use of its derivation. Unpublished ms report, Computer Science Division, University of California, Berkeley, CA. Later published as NASA Ames Research Center Technical Report IC-94-04.

Hancock, T. (1991). Learning 2μDNF formulas and $k\mu$ decision trees. In *Proceedings of the Fourth Annual Workshop on Computational Learning Theory*, (pp. 199–209). San Mateo, CA: Morgan Kaufmann.

Hancock, T., & Hellerstein, L. (1991). Learning read-once formulas over fields and extended bases. In *Proceedings of the Fourth Annual Workshop on Computational Learning Theory*, (pp. 326–336). Santa Cruz, CA: Morgan Kaufmann.

Hirsh, H. (1990). Knowledge as bias. In Benjamin, D. (Ed.), *Change of Representation and Inductive Bias*, pp. 209–221. Kluwer, Norwell, MA.

Kearns, M., & Valiant, L. (1994). Cryptographic limitations on learning Boolean formulae and finite automata. *Journal of the ACM*, *41*(1), 67–95.

Kohavi, R., & Benson, S. (1993). Research note on decision lists. *Machine Learning*, *13*(1), 131–134.

Mahadevan, S., & Tadepalli, P. (1994). Quantifying prior determination knowledge using PAC learning model. *Machine Learning*, *17*(1), 69–105.

Mitchell, T. (1980). The need for biases in learning generalizations. In Shavlik, J., & Dietterich, T. (Eds.), *Readings in Machine Learning*, pp. 184–191. Morgan Kaufmann, San Mateo, CA. Originally published as Rutgers University technical report.

Muggleton, S., & Feng, C. (1990). Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory*, (pp. 369–381). Tokyo, Japan: Ohmsha.

Pagallo, G., & Haussler, D. (1990). Boolean feature discovery in empirical learning. *Machine Learning*, *5*, 71–99.

Pazzani, M., & Kibler, D. (1992). The utility of knowledge in inductive learning. *Machine Learning*, *9*(1), 57–94.

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Mateo, CA.

Pillaipakkamnatt, K., & Raghavan, V. (1995). Read-twice DNF formulas are properly learnable. *Information and Computation*, *122*(2), 236–267.

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

Quinlan, R. (1990). Learning logical definitions from relations. *Machine Learning*, *5*, 239–266.

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

Russell, S. (1988). Tree-structured bias. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, (pp. 641–645). Saint Paul, MN: Morgan Kaufmann.

Russell, S. (1989). *The Use of Knowledge in Analogy and Induction*. Morgan Kaufmann, San Mateo, CA.

Russell, S., Binder, J., Koller, D., & Kanazawa, K. (1995). Local learning in probabilistic networks with hidden variables. In *Proceedings of the* 14^{th} *International Conference on Artificial Intelligence*, (pp. 1146–1152). Montreal, Canada: Morgan Kaufmann.

Russell, S., & Grosof, B. (1987). A declarative approach to bias in concept learning. In *Proceedings of the National Conference on Artificial Intelligence*, (pp. 505–510). Seattle, WA: Morgan Kaufmann.

Russell, S., & Grosof, B. (1990). Declarative bias: An overview. In Benjamin, D. (Ed.), *Change of Representation and Inductive Bias*, pp. 267–308. Kluwer, Norwell, MA.

Shavlik, J., & Towell, G. (1989). An approach to combining explanation-based and neural learning algorithms. *Connection Science*, *3*(1), 231–53.

Tadepalli, P. (1993). Learning from queries and examples with tree-structured bias. In *Proceedings of the Tenth International Conference on Machine Learning*, (pp. 322–329). Amherst, MA: Morgan Kaufmann.

Valiant, L. (1984). A theory of the learnable. *Communications of the ACM*, *27*(11), 1134–1142.

Wegener, I. (1987). *The Complexity of Boolean Functions*. John Wiley & Sons, New York, NY.

Wilkins, D., Clancey, W., & Buchanan, B. (1987). Knowledge base refinement by monitoring abstract control knowledge. *International Journal of Man-Machine Studies*, *27*(3), 281–293.

Received September 22, 1994
Accepted August 29, 1996
Final Manuscript March 3, 1998