From "Proceedings of the Ninth Annual Conference on Computational Learning Theory" (COLT96)

The Dual DFA Learning Problem: Hardness Results for Programming by Demonstration and Learning First-Order Representations
(Extended Abstract)

William W. Cohen
AT&T Laboratories, 600 Mountain Avenue
Murray Hill, NJ 07974
wcohen@research.att.com

Abstract

Learning from "structured examples" is the goal of relational learning and inductive logic programming. Here we analyze a simple learning problem in which examples have non-trivial structure: specifically, a learning problem in which concepts are strings over a fixed alphabet, examples are DFAs, and a string represents the set of all DFAs that accept it. We show that solving this "dual" DFA learning problem is hard, under cryptographic assumptions. This result implies the hardness of several other more natural learning problems, including learning the description logic CLASSIC from subconcepts, and learning arity-two "determinate" function-free Prolog clauses from ground clauses. The result also implies the hardness of two formal problems related to the area of "programming by demonstration": learning straightline programs over a fixed operator set from input-output pairs, and learning straightline programs from input-output pairs, and "partial traces".

1 Introduction

A typical learning problem includes a set of possible concepts $C_1, \ldots, C_n$ and a set of possible instances $x_1, \ldots, x_m$. Conceptually, one can think of these sets as a large 0,1 matrix in which the columns correspond to instances, the rows correspond to concepts, and each row encodes the characteristic function for a concept:

$$\begin{array}{c} \\ C_1 \\ C_2 \\ \vdots \\ C_n \end{array} \begin{array}{c} \begin{array}{cccc} x_1 & x_2 & \ldots & x_m \end{array} \\ \left[\begin{array}{cccc} 0 & 0 & \ldots & 0 \\ 0 & 0 & \ldots & 1 \\ \vdots & \vdots & \ldots & \vdots \\ 1 & 1 & \ldots & 1 \end{array}\right] \end{array}$$

For every such learning problem there is a dual learning problem which is obtained by considering the transpose of this matrix.

In this paper we consider the dual version of the DFA learning problem, in which the concepts are strings over a fixed alphabet, the examples are DFAs, and a string s represents the set of all DFAs that accept it. We consider the *polynomial predictability* of this language—*i.e.*, the problem of finding a hypothesis which is probably approximately correct, but which may be encoded in any representation. We show that even if the class of example DFAs is severely constrained, solving this "dual DFA problem" is as hard as predicting log-depth boolean circuits.

This result leads immediately to a number of cryptographic hardness results for first-order representations, including the resolution of two previously open problems. As one corollary of the dual DFA result we show that the description logic CLASSIC is not polynomially predictable. Earlier, Cohen and Hirsh [1994a] showed that CLASSIC is not pac-learnable from examples, and Pitt and Frazier [1994] showed that CLASSIC is not pac-learnable from membership queries alone; however, the question of the learnability of CLASSIC (and related description logics) from random examples in a representation independent sense has until now remained open. In another application, we show that arity-two "determinate" function-free Prolog clauses are not polynomially predictable; again, although Kietz [1993] showed that this language is not pac-learnable, its learnability in the polynomial predictability model has remained open.

The dual DFA result also gives some insight into the problem of learning simple programs from examples and traces. We show that learning straightline code (without loops or branches) from input/output pairs is hard, even if there are only three possible actions to take at each step of the program. We then show that this problem becomes tractable if the learner also has access to a "trace" that reveals the intermediate values computed by the target program after performing each individual action. However, learning from traces is shown to be cryptographically hard if an adversary is allowed to hide even $O(\log\log n)$ bits of each intermediate value. Further, learning from traces is shown to be as hard as learning DNF if an adversary is allowed to hide only *two* bits of each intermediate value. These results are motivated by problems from the research area of "programming by demonstration" [Cypher, 1993].

2 Preliminaries

This paper uses the model of *pac-learnability*, as introduced by Valiant [1984]. Let X be a set, called the *domain*. Define a *concept* C over X to be a subset of X, and a *language* $\mathcal{L}$ to be a set of concepts. Associated with $\mathcal{L}$ is some scheme for representing the concepts in $\mathcal{L}$. In general, we will be casual about the distinction between a concept and its representation; when there is a risk of confusion we will write the set denoted by a representation C (*i.e.*, the extension of C) as $ext(C)$. We will assume a *size* or *complexity measure* on representations $C \in \mathcal{L}$, and also a size measure on instances $x \in X$. The size of $C \in \mathcal{L}$ (respectively $x \in X$) will be denoted $\|C\|$ (or $\|x\|$). Typically these measures will be polynomially related to the number of bits needed to encode a concept (or instance).

If P is a probability distribution, a *sample of C drawn from X according to P* is a pair of multisets S^+, S^- drawn according to P, S^+ containing only the positive examples of C, and S^- containing the negative ones. We define a sample to be *n_I-bounded* if it contains no example larger than n_I.[1]

Informally, pac-learnability requires that the learning algorithm PacLearn be "probably approximately correct"; *i.e.*, that PacLearn outputs an accurate hypothesis from a given language $\mathcal{L}$ most of the time, whenever the target concept is succinctly expressible in $\mathcal{L}$. Formally, we define a language $\mathcal{L}$ to be *pac-learnable* iff there is an algorithm PacLearn and a polynomial function $m(\frac{1}{\epsilon}, \frac{1}{\delta}, n_I, n_T)$ so that for every $n_t > 0$, every $C \in \mathcal{L}$ of size less than n_T, every $0 < \epsilon < 1$, every $0 < \delta < 1$, and every probability distribution P, PacLearn has the following behavior: when run on a n_I-bounded sample S^+, S^- of C drawn according to P of size $|S^+| + |S^-| > m(\frac{1}{\epsilon}, \frac{1}{\delta}, n_I, n_T)$, PacLearn outputs a hypothesis $H \in \mathcal{L}$ such that $Prob(P(H \triangle C) > \epsilon) < \delta$ where $\triangle$ denotes symmetric difference. The probability above is taken over the possible samples S^+ and S^- and (if PacLearn is a randomized algorithm) over any coin flips made by PacLearn; furthermore, PacLearn runs in time polynomial in $\frac{1}{\epsilon}$, $\frac{1}{\delta}$, n_I, n_T, and the size of the sample.

The function $m(\frac{1}{\epsilon}, \frac{1}{\delta}, n_I, n_T)$ is called the *sample complexity* of PacLearn. A hypothesis H such that $Prob(P(H \triangle C) > \epsilon) < \delta$ is called *ϵ-good with respect to the target C*.

We will say that $\mathcal{L}$ is *polynomially predictable* if there is an algorithm PacPredict that satisfies all the requirements for a pac-learning algorithm for $\mathcal{L}$, except that PacPredict outputs a polynomial time evaluable hypothesis H which is perhaps *not* in the target language $\mathcal{L}$.

One analytic tool used in this paper is *prediction preserving reducibility*, as described by Pitt and Warmuth [1990]. If $\mathcal{L}_1$ is a language over domain X_1 and $\mathcal{L}_2$ is a language over domain X_2, then we say that *predicting $\mathcal{L}_1$ reduces to predicting $\mathcal{L}_2$*, written $\mathcal{L}_1 \trianglelefteq \mathcal{L}_2$, if there is a function $f : X_1 \rightarrow X_2$, henceforth called the *instance mapping*, and a function $g : \mathcal{L}_1 \rightarrow \mathcal{L}_2$, henceforth called the *concept mapping*, so that the following all hold:

1. $x \in C$ if and only if $f(x) \in g(C)$—*i.e.*, concept membership is preserved by the mappings;
2. the size complexity of $g(C)$ is polynomial in the size complexity of C—*i.e.*, the size of concepts is preserved within a polynomial factor; and
3. $f(x)$ can be computed in polynomial time.

Intuitively, $g(C_1)$ returns a concept $C_2 \in \mathcal{L}_2$ that will "emulate" C_1 (*i.e.*, make the same decisions about concept membership) on examples that have been "preprocessed" with the function f. Pitt and Warmuth [1990] showed that if $\mathcal{L}_1 \trianglelefteq \mathcal{L}_2$ and $\mathcal{L}_2$ is polynomially predictable, then $\mathcal{L}_1$ is also polynomially predictable. Conversely, if $\mathcal{L}_1 \trianglelefteq \mathcal{L}_2$ and $\mathcal{L}_1$ is not polynomially predictable, then neither is $\mathcal{L}_2$.

3 The dual DFA problem

A standard problem in computational learning theory is the learnability of DFAs from strings [Angluin, 1987; Kearns and Valiant, 1989]. Here we will consider a dual version of this problem, in which the concepts are strings and the examples are DFAs. Let us consider the domain of DFAs over a fixed alphabet Σ, and the concept class $\mathcal{S}^{\mathrm{DFA}}$ of strings $s \in \Sigma^*$, with semantics defined as as follows: if M is a DFA and s is a string in Σ^*, then $M \in ext(s)$ iff s is accepted by M. In other words, a string denotes the class of DFAs that accept it.

Now let us consider the question of whether this representation is polynomially predictable—*i.e.*, the question of whether a pac hypothesis can be found efficiently if any representation for hypotheses is allowed. We have the following result.

Theorem 1 *If $|\Sigma| \geq 2$ then the language $\mathcal{S}^{\mathrm{DFA}}$ is not polynomially predictable under cryptographic*

[1] We will not assume that n_I is known to the learning algorithms.

assumptions.[2]

The remainder of this section is a proof of this result, based on a prediction-preserving reducibility from a certain class of boolean formulae, which we will now define.

Given the boolean variables $x_1, \ldots, x_n$, define $\mathcal{B}^d_{n,*}$ inductively as follows.

- $\mathcal{B}^0_{n,*}$ is the class of monomials over $x_1, \ldots, x_n$.
- if $d > 0$ and d is odd, then $\mathcal{B}^d_{n,*} \equiv \{b_1 \vee b_2 : b_1 \in \mathcal{B}^{d-1}_{n,*}$ and $b_2 \in \mathcal{B}^{d-1}_{n,*}\}$.
- if $d > 0$ and d is even, then $\mathcal{B}^d_{n,*} \equiv \{b_1 \wedge b_2 : b_1 \in \mathcal{B}^{d-1}_{n,*}$ and $b_2 \in \mathcal{B}^{d-1}_{n,*}\}$.

In other words, $\mathcal{B}^d_{n,*}$ is the class of balanced alternating depth-d boolean formulae over n variables with monomials as leaves.

We will now define some restrictions on DFAs. Let M be a DFA with start state q_0 and transition function δ. Define LEVEL(d, M) recursively as follows:

1. LEVEL$(0, M)$ is the singleton set $\{q_0\}$.
2. For $d > 0$, LEVEL(d, M) is the set $\{q_j : \exists q_i \in$ LEVEL$(d-1, M)$, $\exists a \in \Sigma$ so that $\delta(q_i, a) = q_j\}$

In other words, LEVEL(d, M) is the set of states in M that can be reached with input strings of length d. The *level width* of a DFA M is defined to be the maximum cardinality over all d of LEVEL(d, M). A DFA is *leveled* if LEVEL(d_1, M) and LEVEL(d_2, M) are disjoint for all $d_1 \neq d_2$.

We define LDFA(w) to be the set of leveled acyclic DFAs of level width at most w.

We will also adopt the following notation: if $\mathcal{L}$ is a language over the domain X, then $\mathcal{L}_{n_I,n_T}$ denotes the set of concepts $\{C \in \mathcal{L} : \|C\| \leq n_T\}$ over the restricted domain $\{x \in X : \|x\| \leq n_I\}$. $\mathcal{L}_{n_I,*}$ denotes the set $\bigcup_j \mathcal{L}_{n_I,j}$. Thus $\mathcal{S}^{\mathrm{LDFA}(w)}_{n_I,n_T}$ denotes the class of all strings of length at most n_T, with the semantics given above, over the domain of leveled acyclic DFAs of size at most n_I and level width at most w. The size measure we will use for instances (DFAs) is simply the number of states.

We will now show that

$$\mathcal{B}^d_{n,*} \trianglelefteq \mathcal{S}^{\mathrm{LDFA}(d+2)}_{p_I(n,d),p_T(n,d)}$$

where the functions $p_I(*,*)$ and $p_T(*,*)$ are both polynomials in n and 2^d. The lemma below establishes a slightly stronger result by induction on d.

Lemma 2 *Assume $|\Sigma| \geq 3$. Then for all $d > 0$ and all $n > 2$, $\mathcal{B}^d_{n,*} \trianglelefteq \mathcal{S}^{\mathrm{LDFA}(d+2)}_{n_I,n_T}$, where $n_I = (d+2)(n+2)2^d$ and $n_T = (n+2)2^d$. Further, in every example DFA there is exactly one accepting state, which appears at the maximal depth, and exactly two states total at the maximal depth.*

Proof of lemma: The proof is by induction on d. We will let $\Sigma = \{0, 1, \star\}$. (The symbol "$\star$" will be used as a sort of a wildcard in our construction, and should not be confused with the regular expression operator.) For each d we will produce an instance mapping f^d and a concept mapping g^d that preserve membership, and also satisfy the other conditions of the theorem. We will focus on the bounds on level width (of $d+2$) and string length (of $(n+2)2^d$), since the bound on instance size is implied by these bounds.

Base case. Let $\eta = a_1 \ldots a_n$ be an assignment to $x_1, \ldots, x_n$. (*I.e.*, $a_i = 1$ if x_i is true and $a_i = 0$ if x_i is false.) Let $\overline{a_i}$ denote the negation of a_i. (*I.e.*, $\overline{0} = 1$ and $\overline{1} = 0$.) Recalling that the instance mapping must map an assignment η to a DFA, and define $f^0(\eta)$ to be an automaton with the following structure.

1. There are $2n+1$ states: the start state q^T_0, n states $q^T_1, \ldots, q^T_n$, and n states $q^F_1, \ldots, q^F_n$. The only accepting state is q^T_n.
2. For $i : 1 \leq i \leq n$, there is an arc labeled $\overline{a_i}$ from q^T_{i-1} to q^F_i, there are two arcs labeled a_i and $\star$ from q^T_{i-1} to q^T_i, and if $i > 1$, then there are three arcs labeled a_i, $\overline{a_i}$, and $\star$ from q^F_{i-1} to q^F_i.

See Figure 1 for examples of this construction.

New, we will define the corresponding concept mapping g^0. Recalling that g^0 must map a monomial to a string, let $b \in \mathcal{B}^0_{n,*}$ be a monomial over $x_1, \ldots, x_n$ and define σ_i as follows:

$$\sigma_i \equiv \begin{cases} 1 & \text{if } x_i \in b \\ 0 & \text{if } \overline{x_i} \in b \\ \star & \text{else} \end{cases}$$

We define $g^0(b)$ to be the string $\sigma_1 \ldots \sigma_n$. For example, when $n = 4$ then $g^0(x_2\overline{x_4}) = \star 1 \star 0$

Let us consider now the size and level width bounds. Clearly, the level width of $f^0(\eta)$ is exactly $2 = 0 + 2$, and also $g^0(b)$ is of size $n < (n+2)2^0$. It is also obvious that f^0 is computable in polynomial time, and that there are two states of maximal depth, exactly one of which is accepting, and no other accepting states in the automaton.

It remains to be shown that membership is preserved. For a string s, let $M(s)$ denote the state reached by M after reading in the string s, and let $M^0_\eta = f^0(\eta)$. Consider the monomial b as a subset of the literals $\{x_1, \overline{x_1}, \ldots, x_n, \overline{x_n}\}$, and let $b|_j$ denote $b \cap \{x_1, \overline{x_1}, \ldots, x_j, \overline{x_j}\}$. By induction one can show that for all j, $M^0_\eta(\sigma_1 \ldots \sigma_j)$ must be either q^T_j or q^F_j, and that $M^0_\eta(\sigma_1 \ldots \sigma_j) = q^T_j$ iff η satisfies $b|_j$. Thus the lemma holds for the base case of $d = 0$.

[2] More precisely, the prediction problem is intractable if one or more of the following are intractable: solving the quadratic residue problem, inverting the RSA encryption function, or factoring Blum integers [Kearns and Valiant, 1989].

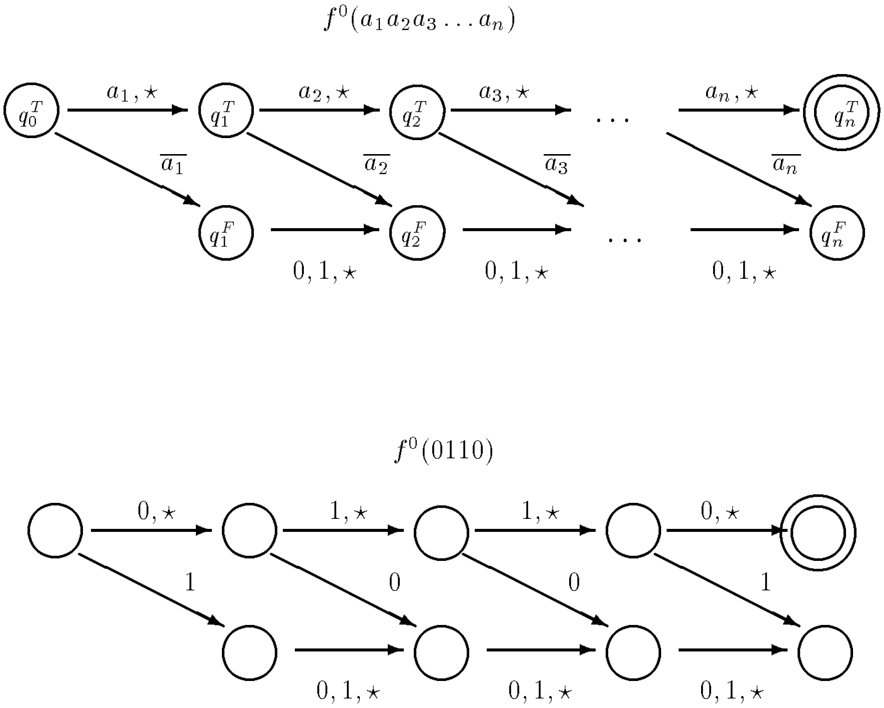

Figure 1: The general construction used for $f^0(\eta)$, and a specific example.

The inductive step. Let us consider first the case in which d is odd—and hence for a formula $b \in \mathcal{B}^d_{n,*}$, $b = b_1 \vee b_2$. By induction there exists an instance mapping f^{d-1} and a concept mapping g^{d-1} that satisfy the conditions of the lemma; also b_1 and b_2 are both in $\mathcal{B}^{d-1}_{n,*}$.

We now define f^d and g^d as follows. For an assignment η, $f^d(\eta)$ is an automaton with the following structure.

1. Let m be the depth of $f^{d-1}(\eta)$. We will use q_0^T to denote the start state of $f^{d-1}(\eta)$, q_m^T to denote the maximal-depth accepting state, and q_m^F to denote the maximal-depth rejecting state. (Note that by induction, q_m^T and q_m^F are unique.) The states of the automaton include each state q in $f^{d-1}(\eta)$; a copy $\hat{q}$ of each state q of $f^{d-1}(\eta)$; m states $r_1, \ldots, r_m$; and two additional states r^T and r^F. The start state is q_0^T. The sole accepting state is r^T.
2. If there is an arc labeled a in $f^{d-1}(\eta)$ between states q_i and q_j, then there is an arc labeled a in $f^d(\eta)$ between states q_i and q_j, and also between states $\hat{q}_i$ and $\hat{q}_j$.
3. For $i : 1 \leq i < m$ there are arcs labeled 0, 1 and $\star$ from state r_i to state r_{i+1}.
4. There are arcs labeled 0, 1 and $\star$ between all of following pairs of states: q_m^T and r_1, q_m^F and $\hat{q}_0^T$, r_m and r^T, $\hat{q}_m^T$ and r^T, and $\hat{q}_m^F$ and r^F.

When d is even, then conditions 1 through 3 are the same, but condition 4 is amended to require arcs labeled 0, 1 and $\star$ between these pairs of states: q_m^T and $\hat{q}_0^T$, q_m^F and r_1, r_m and r^F, $\hat{q}_m^T$ and r^T, and $\hat{q}_m^F$ and r^F.

See Figure 2 for examples of the construction. Clearly this DFA can be constructed in polynomial time (assuming that the size bounds of the theorem hold.)

Finally, let $\alpha = g^{d-1}(b_1)$ and $\beta = g^{d-1}(b_2)$. We define $g^d(b_1 \vee b_2)$ (for odd d) or $g^d(b_1 \wedge b_2)$ (for even d) to be the string $\alpha\star\beta\star$. For example, when $n = 4$, then the following are examples of the mapping g^d. (The underlining is for clarity, and shows the recursive structure of the strings.)

$$
\begin{aligned}
&g^0(x_2\overline{x_4}) = \star 1 \star 0 \star \\
&g^1(x_2\overline{x_4} \vee \overline{x_2}x_4) = \underline{\star 1 \star 0} \star \underline{\star 0 \star 1 \star} \\
&g^1(x_3\overline{x_4} \vee \overline{x_3}x_4) = \underline{\star\star 10} \star \underline{\star\star 01 \star} \\
&g^2((x_2\overline{x_4} \vee \overline{x_2}x_4) \wedge (x_3\overline{x_4} \vee \overline{x_3}x_4)) = \\
&\qquad \underline{\underline{\underline{\star 1 \star 0} \star \underline{\star 0 \star 1 \star}}} \star \underline{\underline{\underline{\star\star 10} \star \underline{\star\star 01 \star}}} \star
\end{aligned}
$$

Let us now consider the size and level width bounds. Inspection of the construction shows that the level width of $f^d(\eta)$ is bounded by one plus the level width of $f^{d-1}(\eta)$. By induction, this can be bounded by $((d-1)+2)+1 = d+2$. For the size bound, let $\mathrm{LEN}(d)$ denote the length of $g^d(b)$ for $b \in \mathcal{B}^d_{n,*}$. We claim that for all d

$$
\mathrm{LEN}(d) \equiv n2^d + \sum_{i=1}^{d} 2^i \;<\; n2^d + 2^{d+1} \;=\; (n+2)2^d
$$

The base case is immediate, as $\mathrm{LEN}(0) = n$. The inductive case can be easily verified by substition, using the fact that $\mathrm{LEN}(d) = 2\mathrm{LEN}(d-1) + 2$.

Finally let us consider if concept membership is preserved by these mapping. We wish to show that for any assignment η and any $b \in \mathcal{B}^d_{n,*}$, η satisfies b iff $f^d(\eta)$

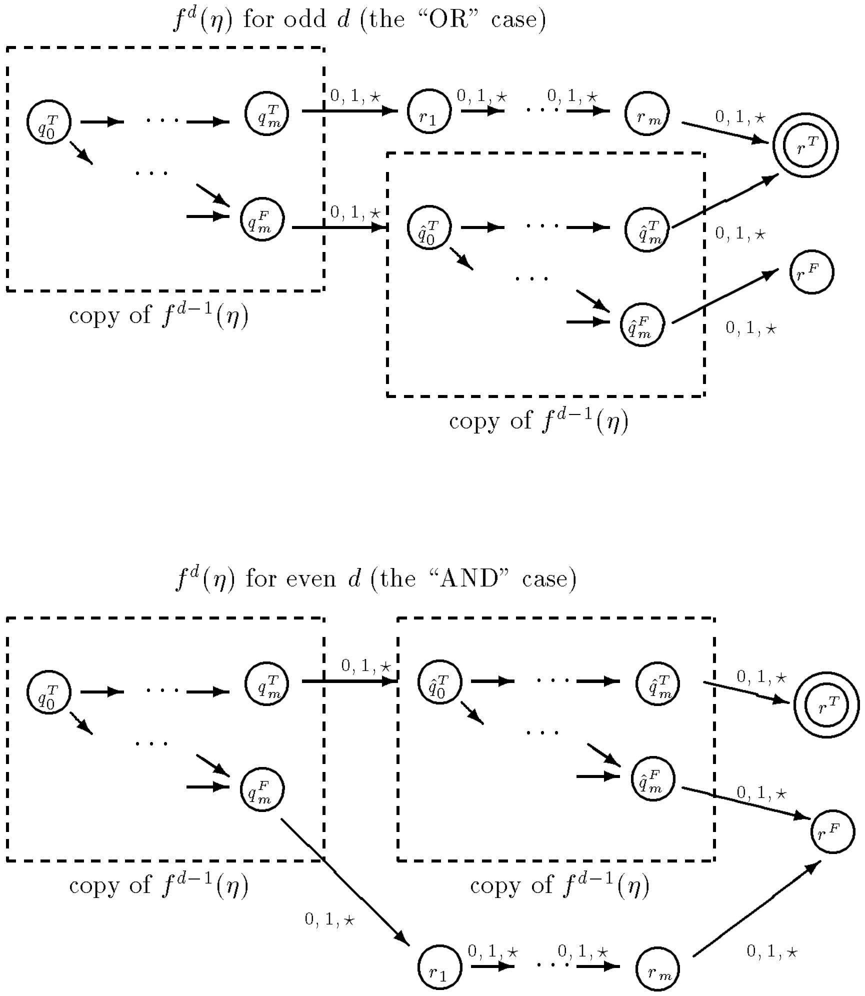

Figure 2: The constructions used for $f^d(\eta)$ for $d > 0$

accepts $g^d(b)$. Assume that this is true for $d-1$, let M_η^d be the machine $f^d(\eta)$, and let $M(s)$ denote the state that the DFA M is in after reading in the string s. We can now argue as follows.

Case 1. Assume d is odd. Then if η satisfies $b = b_1 \vee b_2$, either η satisfies b_1 or b_2 or both. For the following, please refer to Figure 2. If η satisfies b_1 then by the inductive hypothesis $M_\eta^{d-1}(g^{d-1}(b_1))$ is the accepting state q_m^T; referring to Figure 2, clearly $M_\eta^d(s_d) = r^T$, and hence $g^d(b)$ is accepted by M_η^d. If η satisfies b_2 but not b_1 then $M_\eta^{d-1}(g^{d-1}(b_1))$ is the rejecting state q_m^F, but $M_\eta^{d-1}(g^{d-1}(b_2))$ is the accepting state q_m^T. Again, $M_\eta^d(g^d(b)) = r^T$, following the path $q_0^T, \ldots, q_m^F, \hat{q}_0^T, \ldots, \hat{q}_0^T, r^T$.

Conversely, suppose η does not satisfy $b = b_1 \vee b_2$. Then η satisfies neither b_1 nor b_2. By induction M_η^{d-1} rejects both $g^{d-1}(b_1)$ and $g^{d-1}(b_2)$, and hence the automaton M_η^d will, on reading the string $g^d(b)$, visit the states $q_0^T, \ldots, q_m^F, \hat{q}_0^T, \ldots, \hat{q}_m^F$, and finally r^F, rejecting the string.

Case 2. Assume d is even. The argument is analogous to Case 1. If η satisfies $b = b_1 \wedge b_2$, then M_η^{d-1} accepts $g^{d-1}(b_1)$ and $g^{d-1}(b_2)$. Hence M_η^d will, on reading the string $g^d(b)$, visit the states $q_0^T, \ldots, q_m^T$, $\hat{q}_0^T, \ldots, \hat{q}_m^T$, and finally r^T, accepting the string. If η does not satisfy b, then then M_η^{d-1} rejects either $g^{d-1}(b_1)$ or $g^{d-1}(b_2)$. In either case M_η^d finally reach the state r^F, rejecting the string $g^d(b)$.

This completes the proof of the lemma. ■

Lemma 2 gives a polynomial reduction from $\mathcal{B}_{n,*}^{\log n}$ to the dual DFA problem. To complete the proof of Theorem 1, it is only necessary to show (a) that $\mathcal{B}_{n,*}^{\log n}$ is cryptographically hard, and that (b) the requirement that $|\Sigma| \geq 3$ can be weakened to $|\Sigma| \geq 2$. Point (b) is fairly obvious. Point (a) also follows easily from known results on circuit complexity [Boppana and Sipser, 1990]; after performing the necessary conversions, a circuit of

```
(AND woman
     (ALL daughter
          (AND theoryPhD
               unemployed))
     (ALL son
          (AND married
               (ALL spouse
                    doctor))))
```

"The set of women whose daughters are all unemployed theory PhDs, and all of whose sons are married to doctors."

```
(AND mortgageApplication
     (SAME-AS (guarantor)
              (applicant spouse
                         mother)))
```

"The set of mortgage applications that are guaranteed by the applicant's mother-in-law."

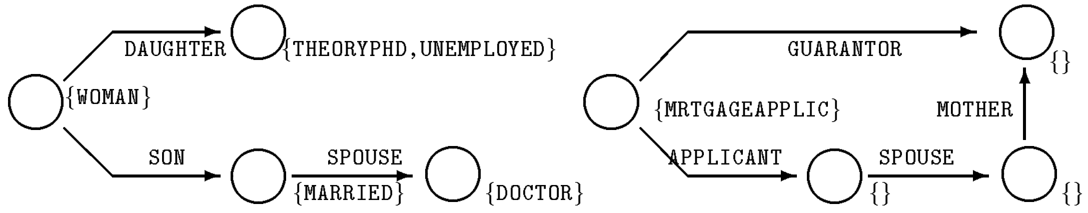

Figure 3: Example CoreClassic concepts, and equivalent concept graphs

size s and depth d over n variables can be converted to a formula in $\mathcal{B}^{2\log n}_{n,sn}$ of size at most sn.

This completes the proof of Theorem 1.

4 The dual DFA result and first-order learnability

The dual DFA problem may seem at first to be an interesting but artificial problem. However, our analysis of the dual DFA problem was motivated by its relationship to certain problems that arise when analyzing the learnability of certain first-order languages, such as logic programs and description logics. In this section we will discuss these relationships.

Description logics are a family of representation languages that have been heavily investigated by the knowledge representation community. (For surveys of this work, see [Borgida, 1992; MacGregor, 1991; Woods and Schmolze, 1992].) The learnability of description logics has also been analyzed [Cohen and Hirsh, 1994a; Cohen and Hirsh, 1994b; Pitt and Frazier, 1994]. In this section we will consider the pac-predictability of CoreClassic, the simple description logic analyzed by Cohen and Hirsh [1994a].

CoreClassic is a description logic containing primitive concepts, roles, attributes, and the constructors AND, ALL, and SAME-AS (which denotes attribute-chain equality). Space limitations do not allow a complete description of this language; however some examples of CoreClassic descriptions are shown in Figure 3, and for readers unfamiliar with description logics, Appendix A gives (briefly) the semantics for the language.

Cohen and Hirsh [1994a] and Pitt and Frazier [1994] have investigated the learnability of CoreClassic and similar description logics in a setting in which examples are concepts, marked as positive if they are *subsumed* by the target concept. (Roughly speaking, concept C_1 subsumes concept C_2 if C_1 is more general than C_2. See Appendix A for more discussion.) Cohen and Hirsh showed that CoreClassic is not pac-learnable in this model. Pitt and Frazier showed that CoreClassic is not exactly learnable from membership queries alone, but is exactly learnable from membership and equivalence queries. However, the question of the pac-predictability of CoreClassic in this model was left open.[3]

The following result is a corollary of Theorem 1.

Theorem 3 *Under the cryptographic assumptions of Theorem 1, CoreClassic is not polynomially predictable (in the model in which examples are concepts labeled by their subsumption relationship with the target concept.)*

Proof (sketch): An important tool in the analysis of CoreClassic is the notion of a *concept graph*. A concept graph is a directed rooted graph in which the arcs are labeled by roles and attributes and the nodes are labeled by primitives. SAME-AS restrictions are represented as multiple paths to a node. Figure 3 also contains two examples of concept graphs. Notice that, at least superficially, concept graphs resemble DFAs: they are both rooted directed graphs with labeled nodes and

[3] An earlier "proof" that CoreClassic was hard to pac-predict [Cohen and Hirsh, 1992] turned out to be erroneous [Cohen and Hirsh, 1995].

edges. A concept graph is said to be *well-formed* if certain other conditions are met, including a restriction that states that for every node of the graph and every possible arc label a, there will be at most one outgoing edge labeled a.

Again, a detailed description of the semantics of CoreClassic concept graphs is beyond the scope of this paper. For our purposes, the following two facts are critical.

- For every DFA M over the alphabet Σ there is a well-formed concept graph G over the primitive alphabet $\{acc\}$, role alphabet $\emptyset$ and attribute alphabet Σ such that $\|G\|$ is polynomial in $\|M\|$ and $a_1 \ldots a_n \in L(M)$ if and only if (ALL a_1 (ALL a_2 …(ALL a_n acc) …)) subsumes G. Furthermore, G can be constructed from M in polynomial time [Cohen and Hirsh, 1994a, Proposition 1].
- The language of CoreClassic and concept graphs are equivalent up to polynomial factors: that is, for every CoreClassic concept D, there is a semantically equivalent well-formed concept graph of size polynomial in $\|D\|$ that can be constructed in polynomial time, and for every well-formed concept graph G, there is a semantically equivalent CoreClassic description of size polynomial in $\|G\|$ that can be constructed in polynomial time [Cohen and Hirsh, 1994a, Theorem 1].

From this it is clear that prediction-preserving reducibilities exist from S^{DFA} to CoreClassic concept graphs to CoreClassic. ■

Another well-studied problem is the learnability of logic programs [Page and Frisch, 1992; Džeroski *et al.*, 1992; Cohen, 1993; Kietz, 1993; Cohen and Jr., 1995]. A special case that has received much attention is the learnability of determinate non-recursive function-free bounded-arity one-clause programs. We will denote this language below a-$\mathcal{D}et\mathcal{LP}$, where a is the bound on arity. Again, we will assume that the reader is familiar with this representation; however Appendix B contains a brief overview of the necessary background on logic programs.

In previous work, Džeroski, Muggleton and Russell [1992] showed that for any constant a, constant depth a-$\mathcal{D}et\mathcal{LP}$ programs are pac-learnable. Later, Kietz [1993] showed that 2-$\mathcal{D}et\mathcal{LP}$ programs of arbitrary depth are not pac-learnable, and Cohen [1993] showed that 3-$\mathcal{D}et\mathcal{LP}$ log-depth programs are not predictable. To date, however, the predictability of arbitrary-depth 2-$\mathcal{D}et\mathcal{LP}$ programs has remained an open question.

Kietz considered a learning model similar to the one considered by Cohen and Hirsh: the target concept is a 2-$\mathcal{D}et\mathcal{LP}$ non-recursive function-free clause, and the examples are non-recursive ground clauses which are labeled as positive iff they are *θ-subsumed* (see the Appendix for a definition) by the target clause. We have the following result for this model.

Corollary 4 *Under the cryptographic assumptions of Theorem 1, 2-$\mathcal{D}et\mathcal{LP}$ non-recursive function-free clauses are not polynomially predictable in the model of Kietz.*

Proof (sketch): Kietz showed that there are functions f_K and g_G such that the following hold: f_K maps a DFA M to a ground clause; g_K maps a string s to a function-free clause; and $s \in L(M)$ iff $g_K(s)$ θ-subsumes $f_K(M)$ [Kietz, 1993, Lemma 13]. The mappings g_K and f_K are similar to the construction used for CoreClassic.

Kietz used this construction to reduce the DFA intersection problem to a consistency problem, thus demonstrating that 2-$\mathcal{D}et\mathcal{LP}$ non-recursive function-free clauses are not pac-learnable. However, the construction also constitutes a prediction-preserving reduction from the dual DFA problem to the language of 2-$\mathcal{D}et\mathcal{LP}$ non-recursive function-free clauses. ■

5 Automatic programming

Another well-studied problem in artificial intelligence is automatic programming from examples (*e.g.*, [Summers, 1977; Biermann, 1978].) Here the goal is to learn a program, typically in a functional language such as LISP, from examples of the form (x, y) where x is an input to the target program and y is the associated output.

Below we will formalize a simple version of this problem. We will consider learning a program that is the composition of n functions from a designated set. This relatively "easy" case is the functional equivalent of n lines of "straight-line" code—code that includes no loops or branches. As an additional restriction, we will require that the output of the target function be binary, by requiring the target function to be of the form $\lambda(x).p(o_n(o_{n-1}(\cdots o_1(x)\cdots)))$ where the range of $p(x)$ is $\{0,1\}$. (The other functions o_1,…,o_n will map strings to strings.)

In previous work, many variants of the automatic programming problem have also been considered, such as learning programs from traces instead of input-output pairs. We will argue below that some of these variations can be adequately modeled by augmenting the standard learning model with additional oracles.

First, however, we will define more precisely the problem of learning straightline code, as introduced informally above.

Define an *operator set* $\langle O, P\rangle$ to be a two sets of functions $O = \{o_1, \ldots, o_{k_O}\}$ and $P = \{p_1, \ldots, p_{k_P}\}$ such that each o_i is a function from $\{0,1\}^n$ to $\{0,1\}^n$ and each p_i is a function from $\{0,1\}^n$ to $\{0,1\}$. The number n is the *domain width* of the operator set, and $k_O + k_P$ is its *cardinality*. We will use $\mathcal{OP}_{n_I,n_O}$ to denote the set of all operator sets of domain width at most n_I and cardinality at most n_O. For any operator set $\langle O, P\rangle$, we define $\mathcal{AP}[\langle O, P\rangle]$ be the set of all concepts with characteristic functions of the form $o_{i_1} \circ \ldots \circ o_{i_n} \circ p_k$ where each o_{i_j} is in O and p_k is in P. We will use $\mathcal{AP}[\mathcal{OP}_{n_I,n_O}]$ to denote the set $\{\mathcal{AP}[\langle O, P\rangle] : \langle O, P\rangle \in \mathcal{OP}_{n_I,n_O}\}$. The size of a concept $C = o_{i_1} \circ \ldots \circ o_{i_n} \circ p_k$ in $\mathcal{AP}[\langle O, P\rangle]$ is defined to be n.

To motivate the slightly awkward notation introduced above, notice that for some operator sets $\langle O, P\rangle$, the class $\mathcal{AP}[\langle O, P\rangle]$ is easily learnable—for example, if every o_i is the identity function then the problem is trivial. However, one would really like to ask questions about the existence of "operator-independent" learning algorithms—algorithms that work for any given sets O and P. With the definitions given above, we can formalize this question succinctly as follows:

> Is there an algorithm PacPredict that polynomially predicts any concept class $\mathcal{AP}[\langle O, P\rangle] \in \mathcal{AP}[\mathcal{OP}_{n_I,n_O}]$, given access to oracles for O and P ?

If the answer to this question is affirmative then we will say that $\mathcal{AP}[\mathcal{OP}_{n_I,n_O}]$ is *uniformly (polynomially) predictable*. *Uniform pac-learnability* is defined analogously. Thus we have reduced the question of operator-independent learnability to a set of (closely related) questions, posed in the usual pac-learning models.

Our first result pertains to automatic programming from input-output pairs. The result shows that even with a small fixed set of operators, and even in the case of straightline code, this problem can be hard. This result is a little surprising, given that most of the research issues arising in automatic programming seem to relate to problems such as inducing loops and conditional statements, not to difficulties associated with learning straightline code.

Theorem 5 *For $n_O \geq 4$, $\mathcal{AP}[\mathcal{OP}_{n_I,n_O}]$ is not uniformly predictable, under the cryptographical assumptions of Theorem 1.*

Proof: We will construct a particular operator set $\langle O, P\rangle$ that is hard to predict.

Theorem 1 gives a prediction-preserving reduction from circuits of depth $\log n$ to

$$\mathcal{S}^{\mathrm{DFA}}_{p_I(n),p_T(n)}$$

via two functions $g^{\log n}$ and $f^{\log n}$, where $p_I(*)$ and $p_T(*)$ are polynomials. Let $n_I = n + \log p_I(n)$, and choose some systematic way of numbering the states in a DFA such that the start state has number 0 (*e.g.*, breadth first). In this way, if η is an assignment to n variables, n_I bits can be used to encode a state in the DFA $f^{\log p_I(n)}(\eta)$: the first n bits encode the assignment η, and the remaining $logp_I(n)$ bits encode the number of the state. We will write the encoding of the i-th state in $f^{\log n}(\eta)$ as $c(\eta)c(i)$ where $c(\eta)$ is the encoding of η and $c(i)$ is the encoding of i.

Now let $O = \{o_0, o_1, o_\star\}$ and $P = \{p_{acc}\}$, where the functions p_{acc}, o_0, o_1, and $o_\star$ are defined as follows. For $a \in \{0, 1, \star\}$, define $o_a(c(\eta)c(i)) \equiv c(\eta)c(j)$ where j is the index of the state in $f^{\log n}(\eta)$ reached by traversing the edge labeled a from state i. Also define $p_{acc}(c(\eta)c(i))$ to be 1 if the i-th state of $f^{\log n}(\eta)$ is accepting and 0 if the state i-th state of $f^{\log n}(\eta)$ is rejecting.

Note that this encoding satisfies the following condition: the string $a_1 \ldots a_n$ is accepted by $f(\eta)$ iff

$$(o_{a_1} \circ \ldots \circ o_{a_n} \circ p_{acc})(c(\eta)c(0)) = 1$$

In other words, these operations allow one to emulate the automaton $f(\eta)$ using a composition of the operators. The theorem now follows from Theorem 1. ∎

Another variant of automatic programming problem is the problem of learning programs from traces. Here the assumption is that the learner can observe (perhaps only partially) the actions taken by a working copy of the target program. Programming from traces is closely related to the currently active research area of *programming by demonstration* [Cypher, 1993]. Here the operations used in the target program are (typically) operations on a graphical user interface, and the "teacher" from whom traces are obtained is the end user.

In a formal learning model it is natural to model these traces with an oracle. However, in a programming by demonstration context, what a "trace" is depends on the system being used.

One reasonable assumption is that one can observe the intermediate values derived by the teacher but not the operators. This might be the case if the operators appearing in the learned program are *abstractions* of the actions actually taken by the user. For example, consider the problem of learning a sequence of commands that converts an entry in a mailing list from one format to another. The user's actions might be primitive editor commands, such as *forward-one-character*, *back-one-word*, *insert-newline*, and so on. However, the mailing list conversion problem might be more appropriately modeled with abstract operators such as *forward-to-the-beginning-of-sentence*, *back-to-beginning-of-postal-code*, and so on. Designing such abstract operator sets and learning "programs" containing such abstract operators is an important emphasis of at least some programming by demonstration research efforts.

In this scenario, even after observing use of primitive operator, one cannot always determine what "abstract operator" was applied. (In learning an editor macro, for instance, one might be able to observe that a user moved the cursor from point p_1 to point p_2 in a document, but it might not be obvious whether this action corresponds to the operator *forward-to-end-of-line* or to the operator *forward-to-end-of-postal-code*.) However, one can observe the *effects* of the abstract operator. One can model this situation with an oracle TRACE(x) which, when called with the input x while learning the target concept $o_{i_1} \circ \ldots \circ o_{i_n} \circ p_k$ returns the sequence of strings

$\langle s_1, \ldots, s_n \rangle$ where $s_j = (o_{i_1} \circ \ldots \circ o_{i_j})(x)$. In other words, s_j is the j-th intermediate value constructed by the target function.

Let us now consider the problem of learning from examples and the TRACE oracle. While this problem is certainly more complex than recording a macro, it is easy to show that this restricted information is enough to allow efficient learnability in the pac sense.

Theorem 6 *$\mathcal{AP}[\mathcal{OP}_{n_I,n_O}]$ is uniformly pac-learnable from examples and the TRACE oracle. Furthermore, the prediction algorithm runs in time polynomial in n_O.*[4]

Proof: Omitted from this abstract. ■

In some circumstances, it may not be appropriate to assume that one has access to the intermediate values computed by the target function. A slightly weaker assumption is to assume that some but not all aspects of the intermediate states are observable. These situations can be modeled as follows. If $s \in \{0,1\}^n$, then let $\mathrm{HIDE}_k(s)$ denote the set of all strings $s' \in \{0,1,\star\}^n$ that can be derived by changing up to k elements of s from "0" or "1" to "$\star$". For example,

$$\mathrm{HIDE}_1(1101) = \{\star 101, 1\star 01, 11\star 1, 110\star, 1101\}$$

We now define the oracle $\mathrm{PTRACE}_k(x)$ to return some sequence $\langle h_1, \ldots, h_n \rangle$ such that for all $j \leq 1$, $h_j \in \mathrm{HIDE}_k((o_{i_1} \circ \ldots \circ o_{i_j})(x))$. In other words, the oracle is allowed to edit the correct intermediate values by changing up to k bits of each value to the symbol $\star$. Learning from a PTRACE oracle and examples is one model of learning from partially observable intermediate values.

Notice that while we are assuming that the elements in the PTRACE result from "hiding" at most k bits of the intermediate values, we impose no constraints on which k bits are "hidden". In effect our model assumes that these bits are hidden by an adversary. However, in the constructions below the adversary follows a very simple strategy—the hidden bits are always the lowest order bits of a string.

Recall that in the model of learning from input-output pairs, the trace is completely hidden; thus Theorem 5 shows that if all bits in a trace are hidden, then learning can be hard. (*I.e.*, for $n_O \geq 4$, uniformly predicting $\mathcal{AP}[\mathcal{OP}_{n_I,n_O}]$ from examples and PTRACE_{n_I} is difficult.) This result can be strengthened as follows.

Theorem 7 *For $n_O \geq 4$ and $k \geq \log(\log n_I)$, $\mathcal{AP}[\mathcal{OP}_{n_I,n_O}]$ is not uniformly predictable from examples and the $PTRACE_k$ oracle, under the cryptographic assumptions of Theorem 1.*

[4]Notice that if PacPredict is a uniform prediction algorithm, then PacPredict must run in time polynomial in n_I, since n_I is the size of the examples used by PacPredict. However, PacPredict need *not* run in time polynomial in the number of operators n_O.

Proof: The argument follows the argument of Theorem 5, in which $\log n$-depth circuits are reduced to a learning problem in $\mathcal{AP}[\mathcal{OP}_{n_I,n_O}]$. However, for this theorem we will encode a state q in $f^{\log n}(\eta)$ in three sections: first, an encoding of η, which requires $O(n)$ bits; second, an encoding of the depth d of the state, which requires $O(\log n)$ bits; and finally, an encoding of where q lies in the d-th level of $f^{\log n}(\eta)$, which requires $O(\log \log n)$ bits.

Theorem 5 shows how examples are converted, and also how the oracles for F and G can be simulated. To prove the theorem it is only necessary to show how the PTRACE oracle can be emulated.

To emulate the oracle for PTRACE(x) on $x = f^{\log n}(\eta)$, simply construct $f^{\log n}(\eta)$ and choose a random path to an accept state (if x is positive) or a reject state (if x is negative), giving a sequence of intermediate values $s_1, \ldots, s_n$. Of course, these need not agree with the actual values $s_1^*, \ldots, s_n^*$ taken by the automata in accepting the target string; however, they will agree in all bits *except* those $O(\log \log n)$ bits needed to encode the position within a level. To construct a legal output for PTRACE, therefore, it is only necessary to "hide" these bits by replacing them with $\star$. Thus if exactly these bits are "hidden" the resulting sequence $h_1, \ldots, h_n$ is a legal output for PTRACE. ■

This theorem shows that allowing an adversary to hide a small portion of each intermediate value in a trace makes learning difficult. Carrying this line of investigation a little further, one can show that hiding even a *constant* number of bits in each element of the trace can make learning as hard as learning boolean functions in disjunctive normal form (DNF), an open problem in computational learning theory.

Theorem 8 *For $n_O \geq 4$ and $k \geq 2$, $\mathcal{AP}[\mathcal{OP}_{n_I,n_O}]$ is not uniformly predictable from examples and the $PTRACE_k$ oracle unless DNF is polynomially predictable.*

Proof: In the argument above, the key insight is that the dual DFA problem for leveled width w automata can be reduced to the problem of automatic programming from traces, where the number of bits hidden by PTRACE is logarithmic in w, the maximal automata width. In other words, predicting $\mathcal{S}_{n_I,*}^{\mathrm{LDFA}(w)}$ can be reduced to a prediction problem $\mathcal{AP}[\langle O, P \rangle] \in \mathcal{AP}[\mathcal{OP}_{poly(n_I),4}]$, where the learning is done from examples and a $\mathrm{PTRACE}_{\log w}$ oracle. Thus to prove the theorem it is only necessary to show a prediction-preserving reducibility from DNF over n variables to $\mathcal{S}_{poly(n),*}^{\mathrm{LDFA}(4)}$, the dual DFA problem for leveled automata with level width 4. We will sketch this reduction below.

Let $T_1, \ldots, T_s$ be monomials, and let $\phi = \vee_{i=1}^s T_i$ be an polynomial-sized DNF formula of size n_T over n variables. Assume without loss of generality that the number of terms s is equal to n. (By padding we can make these quantities equal). We define the concept mapping g_{DNF} to be

$$g_{\mathrm{DNF}}(T_1 \vee T_2 \vee \ldots T_n) = g^0(T_1) \star g^0(T_2) \star \ldots \star g^0(T_n)$$

where $g^0 : \mathcal{B}^d_{n,*} \rightarrow \{0,1,\star\}^n$ is the concept mapping used in Theorem 1. For example, if $n = 4$,

$$g_{\mathrm{DNF}}(x_2\overline{x_3} \vee \overline{x_2}x_3) = \underline{\star 10 \star} \star \underline{\star 01 \star}$$

For an assignment η, we define the instance mapping $f_{\mathrm{DNF}}(\eta)$ to consist of n copies of $f^0(\eta)$, together with a linear sequence of "success" states. These copies will be connected so that if the automata reaches the accept state of any copy of $f^0(\eta)$, it will always jump to a "success" state; however if the automata reaches the reject state of a copy of $f^0(\eta)$, it will always procede to the start state of the next copy. The automata accepts if it eventually reaches the "success" sequence. Such an automaton is as shown in Figure 4. In the interests of brevity, no more formal description will be given. Clearly the size of this construction is polynomial, it can be computed in polynomial time, and the level width of the DFA is bounded by 3. The arguments for the correctness of the mapping parallel the arguments used in Theorem 1. ■

To summarize the results of this section, we have investigated an extension of the problem of recording a macro, in which the goal is to learn a linear sequence of operators from a known set. We showed that this learning problem is easy if the operators are observable, but hard if the operators are hidden. Somewhat surprisingly, the problem is hard even if the operators are only partially hidden—hiding even two bits from each intermediate value makes learning as hard as learning DNF, and hiding $O(\log \log n)$ bits makes learning cryptographically hard.

6 Conclusions

In this paper we analyzed a dual version of the problem of learning DFAs, in which examples are DFAs, concepts are strings, and a string s denotes the set of DFAs that accept s. We showed that this "dual DFA problem" is as hard as learning log-depth boolean circuits, even if example DFAs are restricted to be over a alphabet of only two letters and also acyclic, leveled, and of logarithmic level width.

This result solves two open probems in the learnability of first-order representations. First, under cryptographic assumptions, the description logic CLASSIC is not learnable in the model proposed by Cohen and Hirsh [1994a]. Second, under cryptographic assumptions, "determinate" function-free Prolog clauses are not polynomially predictable in the model proposed by Kietz [1993].

The dual DFA result also has implications for the problem of learning straightline programs (without branches or loops) from input-output pairs or traces—a trace being a sequence revealing the intermediate values required to evaluate the target function on an example x. We motivate a particular formalization of this problem and show that learning from input-output pairs is cryptographically hard, but that learning from traces is tractable. As an intermediate between these two models, we then propose a model of learning from a *partial trace*. In particular we consider learning from examples and the oracle PTRACE_k, which can be thought of as returning a complete trace that has been edited by an adversary who can "hide" at most k bits of every intermediate value. We show that learning from partial traces is cryptographically hard if even only $O(\log \log n)$ bits of each value are hidden. Furthermore, learning from partial traces is as hard as learning DNF if even two bits of each value are hidden by an adversary. These results may have implications for the research area of "programming by demonstration" [Cypher, 1993].

In all of these results an adversarial choice of example distributions was assumed; this means in effect that an adversary determines both the architecture and labeling of every example DFA. We leave as open questions the complexity of the dual DFA learning problem in less adversarial settings, such as the distributions of "typical" DFAs considered by Freund *et. al* [1993].

Acknowledgements

The author would like to thank many colleagues but particularly Haym Hirsh, Lenny Pitt, and Rob Schapire, for numerous helpful discussions; Haym Hirsh, for proofreading a draft of the paper; and the members of the COLT program committee, for several helpful technical suggestions.

References

(Angluin, 1987) Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Control*, 75:87–106, 1987.

(Biermann, 1978) A. Biermann. The inference of regular lisp programs from examples. *IEEE Transactions on Systems, Man and Cybernetics*, 8(8), 1978.

(Boppana and Sipser, 1990) R. Boppana and M. Sipser. The complexity of finite functions. In *Handbook of Theoretical Computer Science*, pages 758–804. Elsevier, 1990.

(Borgida, 1992) Alex Borgida. Description logics are not just for the flightless-birds: a new look at the utility and foundations of description logics. Technical Report DCS-TR-295, Rutgers University Department of Computer Science, 1992.

(Cohen and Hirsh, 1992) William W. Cohen and Haym Hirsh. Learnability of description logics. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, Pittsburgh, Pennsylvania, 1992. ACM Press.

(Cohen and Hirsh, 1994a) William W. Cohen and Haym Hirsh. The learnability of description logics with equality constraints. *Machine Learning*, 17(2/3), 1994.

(Cohen and Hirsh, 1994b) William W. Cohen and Haym Hirsh. Learning the CLASSIC description logic: Theoretical and experimental results. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourth International Conference (KR94)*. Morgan Kaufmann, 1994.

(Cohen and Hirsh, 1995) William W. Cohen and Haym Hirsh. Corrigendum for "learnability of description logics". In *Proceedings of the Eighth Annual Workshop on Computational Learning Theory*, Santa Cruz, California, 1995. ACM Press.

(Cohen and Jr., 1995) William W. Cohen and C. David Page Jr. Polynomial learnability and inductive logic programming: Methods and results. *New Generation Computing*, 13(3), 1995.

(Cohen, 1993) William W. Cohen. Cryptographic limitations on learning one-clause logic programs. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, Washington, D.C., 1993.

(Cypher, 1993) Allen Cypher, editor. *Watch what I do: Programming by demonstration*. The MIT Press, Cambridge, Massachusetts, 1993.

(Džeroski *et al.*, 1992) Sašo Džeroski, Stephen Muggleton, and Stuart Russell. Pac-learnability of determinate logic programs. In *Proceedings of the 1992 Workshop on Computational Learning Theory*, Pittsburgh, Pennsylvania, 1992.

(Freund *et al.*, 1993) Y. Freund, M. Kearns, D. Ron, R. Rubinfeld, R. Schapire, and L. Sellie. Efficient learning of typical finite automata from random walks. In *Proceedings of the 25th ACM Symposium on the Theory of Computing*, pages 315–324. ACM Press, 1993.

(Kearns and Valiant, 1989) Micheal Kearns and Les Valiant. Cryptographic limitations on learning Boolean formulae and finite automata. In *21th Annual Symposium on the Theory of Computing*. ACM Press, 1989.

(Kietz, 1993) Jörg-Uwe Kietz. Some computational lower bounds for the computational complexity of inductive logic programming. In *Proceedings of the 1993 European Conference on Machine Learning*, Vienna, Austria, 1993.

(Lloyd, 1987) J. W. Lloyd. *Foundations of Logic Programming: Second Edition*. Springer-Verlag, 1987.

(MacGregor, 1991) R. M. MacGregor. The evolving technology of classification-based knowledge representation systems. In John Sowa, editor, *Principles of semantic networks: explorations in the representation of knowledge*. Morgan Kaufmann, 1991.

(Muggleton and Feng, 1992) Stephen Muggleton and Cao Feng. Efficient induction of logic programs. In *Inductive Logic Programming*. Academic Press, 1992.

(Page and Frisch, 1992) C. D. Page and A. M. Frisch. Generalization and learnability: A study of constrained atoms. In *Inductive Logic Programming*. Academic Press, 1992.

(Pitt and Frazier, 1994) L. Pitt and M. Frazier. Classic learning. In *Proceedings of the Seventh Annual ACM Conference on Computational Learning Theory*, New Brunswick, NJ, 1994. ACM Press.

(Pitt and Warmuth, 1990) Leonard Pitt and Manfred Warmuth. Prediction-preserving reducibility. *Journal of Computer and System Sciences*, 41:430–467, 1990.

(Summers, 1977) Phillip D. Summers. A methodology for LISP program construction from examples. *Journal of the Association for Computing Machinery*, 24(1):161–175, January 1977.

(Valiant, 1984) L. G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11), November 1984.

(Woods and Schmolze, 1992) W. A. Woods and J. G. Schmolze. The KL-ONE family. *Computers And Mathematics With Applications*, 23(2-5), March 1992.

A Semantics of CoreClassic

Concepts in CoreClassic describe subsets of a domain I of "individuals". Concepts are built from a alphabet of *primitive class symbols* p, each of which corresponds to a subset of I; *role symbols* r, each of which corresponds to a subset of $I \times I$; *attribute symbols* a, each of which corresponds to a function from I to I; and the operators AND, ALL, and SAME-AS.

For a primitive p, let $ext(p)$ denote the subset of I that corresponds to p; for a role r, let $r(x, y)$ be the corresponding binary predicate; and for an attribute a, let $a(x)$ be the corresponding function. A CoreClassic concept is defined inductively as follows.

- If p is a primitive class symbol then p is a concept denoting $ext(p)$
- If r is a role or attribute and C is a concept, then (ALL r C) is a concept, denoting the set of all $x \in I$ such that $\forall y \in I,\ r(x,y) \Rightarrow y \in ext(C)$, where $ext(C)$ is the set denoted by the concept C.
- If $a_1, \ldots, a_k, b_1, \ldots, b_l$ are attribute symbols, then (SAME-AS $(a_1 \ldots a_k)$ $(b_1 \ldots b_l)$) is a concept denoting the set of $x \in I$ such that $a_k(\cdots a_2(a_1(x)) \cdots) = b_l(\cdots b_2(b_1(x)) \cdots)$.
- If $C_1, \ldots, C_n$ are concepts then (AND $C_1 \ldots C_n$) is a concept denoting $\bigcap_{i=1}^{n} ext(C_i)$.

An important relationship in description logics is *subsumption*. Concept C_1 subsumes C_2 if $ext(C_1) \supseteq ext(C_2)$ regardless of the extensions of the primitive concepts, roles and attributes used in C_1 and C_2. For example the concept C_1 = (AND politician lawyer) would not subsume the concept C_2 = (AND congressman lawyer (SAME-AS mistress aide)) even if it happened to be the case $ext(C_1) \supseteq ext(C_2)$. (This would be the case, for instance, if $ext(politician) \supset ext(congressman)$.) However , $C_1' =$ (AND congressman lawyer) would subsume C_2, since every element of C_1' is necessarily a member of C_2.

B Semantics of logic programs

In the interests of simplicity, the definitions below only coincide with the usual ones for the case of non-recursive function-free single-clause Prolog programs. For a more complete description of logic programming see one of the standard texts (*e.g.*, [Lloyd, 1987]).

Logic programs are written over an alphabet of *constant symbols*, *predicate symbols*, and *variables*. A *function-free literal* is written $p(X_1, \ldots, X_k)$ where p is a predicate symbol and $X_1, \ldots, X_k$ are variables. A *fact* is written $p(t_1, \ldots, t_k)$ where p is a predicate symbol and $t_1, \ldots, t_k$ are constant symbols. The number of arguments k to a literal (or fact) is called its *arity*.

A *ground clause* is written

$$a\text{:-}b_1, \ldots, b_n$$

where a and the b_i's are all facts. A *function-free clause* is written

$$C\text{:-}D_1, \ldots, D_n$$

where C and the D_i's are all function-free literals. The fact (or literal) to the left of the ":-" symbol is the *head* of the clause and the facts (literals) to the right of the ":-" symbol are the *body* of the clause.

A *substitution* is a partial function mapping variables to constant symbols or variables. If θ is a substitution and A is a literal, we will use $A\theta$ to denote the result of replacing each variable X in A with the constant symbol to which X is mapped by θ.

The function-free clause $C\text{:-}D_1, \ldots, D_n$ is said to *θ-subsume* the ground clause $a\text{:-}b_1, \ldots, b_m$ if there is some substitution θ such that $C\theta = a$ and $\forall i : 1 \leq i \leq n$, $D_i\theta \in \{b_1, \ldots, b_m\}$.

The following restrictions are modified from Muggleton and Feng [1992]. If $C\text{:-}D_1 \wedge \ldots \wedge D_n$ is an (ordered) definite clause, then the *input variables* of the literal D_i are those variables appearing in D_i that also appear in the clause $C\text{:-}D_1 \wedge \ldots \wedge D_{i-1}$; all other variables appearing in B_i are called *output variables*. A literal D_i is *determinate* (with respect to a ground clause $a\text{:-}b_1, \ldots, b_m$) if for every possible substitution σ such that $C\text{:-}D_1 \wedge \ldots \wedge D_{i-1}$ θ-subsumes $a\text{:-}b_1 \wedge \ldots \wedge b_m$, there is at most one substitution θ so that $D_i\sigma\theta \in \{b_1, \ldots, b_m\}$. Less formally, a literal is determinate if its output variables have only one possible binding—that is, if the predicate associated with the literal denotes a function, rather than an arbitrary relation.

A function-free clause $C\text{:-}D_1 \wedge \ldots \wedge D_n$ is *determinate* with respect to a ground clause $a\text{:-}b_1, \ldots, b_m$ if every literal D_i in the body of the clause is determinate with respect to the ground clause. If $\mathcal{D}$ is a distribution over ground clauses, a function-free clause is *determinate* (with respect to $\mathcal{D}$) if it is determinate with respect to every ground clause with non-zero weight under $\mathcal{D}$.

Finally, define the *depth* of a variable appearing in a function-free clause $C\text{:-}D_1 \wedge \ldots \wedge D_n$ as follows. Variables appearing in the head of a clause have depth zero. Otherwise, let D_i be the first literal containing the variable X, and let d be the maximal depth of the input variables of D_i; then the depth of X is $d+1$. The depth of a clause is the maximal depth of any variable in the clause.

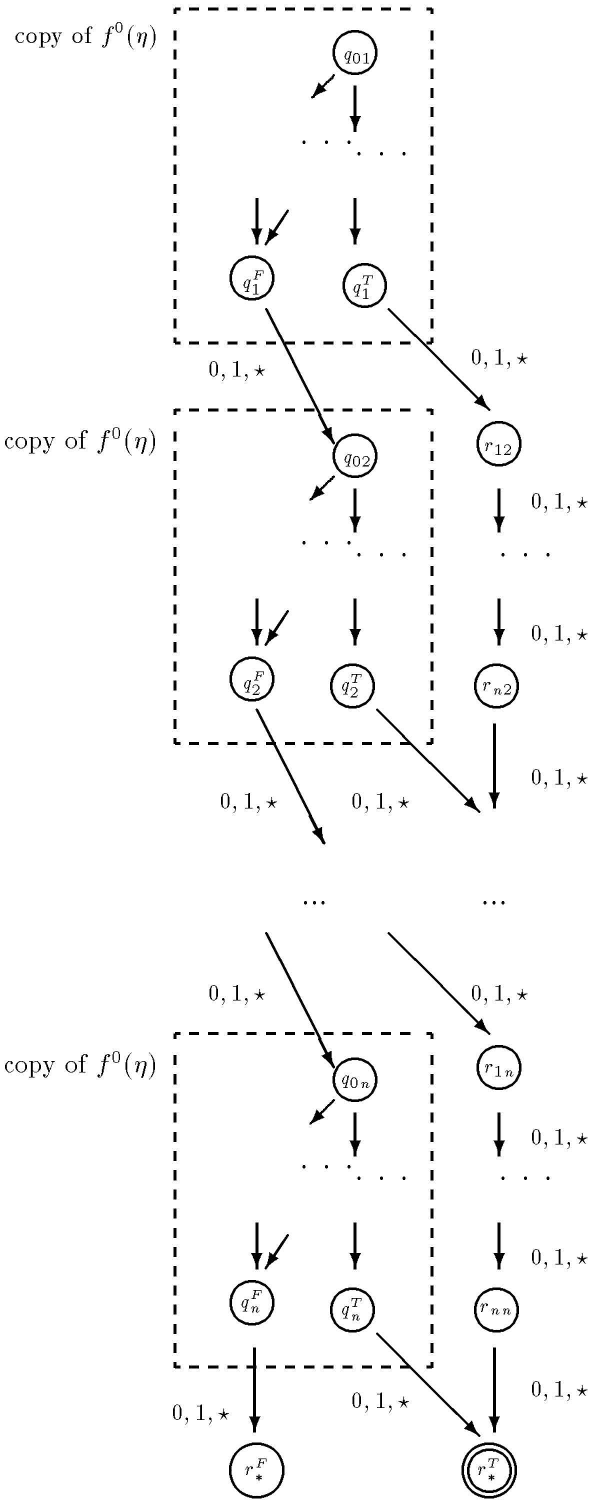

Figure 4: The instance mapping for the reduction from DNF to $\mathcal{S}^{\mathrm{LDFA}(3)}$