On Learning Bounded-Width Branching Programs [*]

Funda Ergün **S Ravi Kumar** **Ronitt Rubinfeld**

Department of Computer Science
Cornell University
Ithaca, NY 14853.

Abstract

In this paper, we study PAC-learning algorithms for specialized classes of deterministic finite automata (DFA). In particular, we study branching programs, and we investigate the influence of the *width* of the branching program on the difficulty of the learning problem. We first present a distribution-free algorithm for learning width-2 branching programs. We also give an algorithm for the proper learning of width-2 branching programs under uniform distribution on labeled samples. We then show that the existence of an efficient algorithm for learning width-3 branching programs would imply the existence of an efficient algorithm for learning DNF, which is not known to be the case. Finally, we show that the existence of an algorithm for learning width-3 branching programs would also yield an algorithm for learning a very restricted version of parity with noise.

1 Introduction

The problem of learning deterministic finite state automata (DFA) has been well studied in recent years. In general, it is hard to learn the class of DFA in the PAC-learning model ([14], [13]). However, there are PAC-learning algorithms for specialized classes of DFA. The techniques used to design them have been adapted for use in algorithms for several applications including text correction, DNA sequencing, part-of-speech tagging, and handwriting recognition [20], [24], and [21].

In this paper, we focus on learning algorithms for a subclass of DFA referred to as bounded-width branching programs. We use the following definition of bounded-width branching programs which is similar to that given in [6], and is a subclass of the more traditional notion of width-w branching programs defined in [7].

The class of *width-w branching programs* (w-BPs) that accept strings $x_1 \ldots x_l$ of a fixed length l are defined as a rectangular w by l array of nodes, where each node in the i-th column is assigned two outgoing edges, one to be followed if the variable x_i has value 1 and the other if has value 0. The edges must terminate at a node in the next column to the right of its source.[1] The *size* of the branching program is the total number of nodes. Note that as stated, this model represents only *read-once* branching programs. However, since we are interested in distribution-free learning, a standard prediction-preserving reduction (which repeats the input several times) can be used to show that learning read-once branching programs is as hard as the general problem of learning branching programs. The problems of learning polynomial size automata and learning polynomial size BPs are reducible to each other by prediction-preserving reductions.

The languages accepted by 5-BPs have been shown to contain all of *NC*^{1} [6]. Thus, by the results of [13], it is *NP*-hard to learn the class of 5-BPs. On the other hand, in this paper we give an algorithm to learn the class of 2-BPs in the PAC-setting. We then prove that learning 3-BPs is as hard as learning DNF. The complexity of learning the latter is not known. We then relate the problem of learning 3-BPs to a class of automata that we refer to as k-mistake parity automata. This is a restricted class of automata which compute parity functions, but are incorrect on a large fraction of the inputs. The inputs on which the automata are incorrect are determined by parity functions on a prefix of the input.

2 Related Work

The problem of learning finite automata has been studied extensively. In the case of learning from examples over which the learner has no control, it has been shown that the problem

[*]This work was supported by ONR Young Investigator Award N00014-93-1-0590. The third author was also supported by the United States - Israel Binational Science Foundation Grant 92-00226.

[1]This is more restrictive than the definition in [6], where the i-th column depends on an arbitrary x_j and more than one column may depend on any particular x_j.

of finding the smallest automaton consistent with a given set of samples, and even approximating the number of states in the automaton by a polynomial, is *NP*-hard ([9], [2], [15]). Even if the condition on the representation of the hypothesis is relaxed, the problem does not become easier: In [13], prediction-preserving reductions of [14] are used to show that (under cryptographic assumptions), predicting the class by any reasonable representation using random examples is hard. However, in [8], algorithms are given for efficient learning of *typical DFA* (automata for which the underlying graph is chosen adversarially but the accept/reject labels at each state are chosen randomly) from random examples, even when there is no means of resetting the machine. In [23], the problem of learning automata with a very small number of states (where the alphabet size is not constant) is investigated. It is shown that learning k-BPs is equivalent to learning k-state automata (over a polynomial size alphabet).

In the stronger model of learning finite automata with membership queries, the task seems to be less difficult. In [4], an algorithm is given which learns DFA, given access to a teacher that answers questions. This algorithm assumes that the automaton is reset between queries. In [17] and [18], this assumption is discarded and the algorithms presented learn automata from input/output behavior, in the absence of a means of resetting the machine to a start state.

The exact complexity of learning DNF (without queries) is not known. However, under uniform distribution on labeled samples, DNF are efficiently learnable with queries. In [11], it is shown that the class of k-term DNF is not properly learnable unless *NP* = *RP* ([16], [11]).

3 Definitions

Let $\mathcal{W}_k$ denote the concept class of width-k branching programs. Let l denote the length of the branching program. Let $\vec{x} = x_1, x_2, \ldots, x_l$ be the input. Unless and otherwise stated, all inputs are assumed to be over the binary alphabet $\{0, 1\}$. The suffix $x_k, x_{k+1}, \ldots, x_l$ is denoted $\vec{x}_k$.

In any width-2 branching program illustrated, let the top (resp. bottom) be the accepting (resp. rejecting) track. We can characterize all transitions of any $M \in \mathcal{W}_2$. M has l stages. Each stage can be identified as being one of the following types:

- Stage k is called (k, b)-*non-merging* if the transitions on symbol b for that stage compute a linear function, *i.e.,* they go to different states. We refer to the Figure 1(a) as a *switch* transition and the Figure 1(b) as a *pass* transition.
- Stage k is called (k, b)-*merging* if the transitions on symbol b for that stage compute a nonlinear function, *i.e.,* they go to the same state (Figure 1(c), (d)).

3.1 Linear Automata

We say that $M \in \mathcal{W}_2$ is k-*linear* if for all $l \geq j \geq k$ and $b \in \{0, 1\}$, the stages of M are (j, b)-non-merging. An

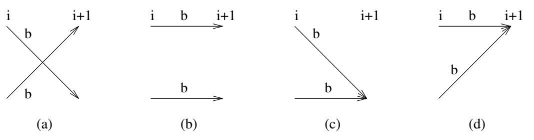

Figure 1: Various Transitions in a Width-2 Automaton

interesting special case of width-2 automata is the class of *1-linear automata* denoted $\mathcal{L}$. These are automata in which all transitions are non-merging. We drop the *1-* prefix when it is obvious from the context.

This special class of automata computes a linear function of its inputs, *i.e.,* functions of the form $f(\vec{x}) = (\sum_{0 \leq i \leq l} a_i x_i + c_i) \bmod 2 = (c + \sum_{0 \leq i \leq l} a_i x_i) \bmod 2$ where $c, c_1, \ldots, c_l, a_1, \ldots, a_l \in \{0, 1\}$. Here, a_i is 1 (resp. 0) if the i-th stage is a switch (resp. pass) transition. The additive constant c_i captures switch transitions on a 0. Thus, $c_i = 1$ when a transition changes tracks on a 0, and $c_i = 0$ otherwise (under our assumption that the top track is the accepting track).

Under this definition, an automaton computing a *parity* function is a special case of a 1-*linear* automaton. This class is denoted by $\mathcal{P}$. In this case, $c_i = 0$ for $1 \leq i \leq l$. In other words, there are no switch transitions on a 0.

Figure 2 is an example of a linear automaton computing $x_0 + x_3 + 1 + x_5 + (x_6 + 1) \bmod 2$. We use the convention throughout this paper that unlabeled arcs implicitly carry both 0 and 1 labels.

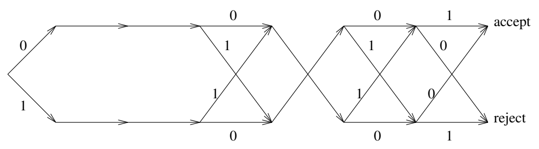

Figure 2: An Example of a Linear Automaton

4 Width-2 Automata

In this section, we present algorithms for learning width-2 automata. The first algorithm does not produce a width-2 automaton as output, but is distribution-free. The second algorithm outputs a width-2 automaton, but is guaranteed to work only under uniform distribution on labeled samples.

4.1 The Distribution-Free Algorithm

Using the characterization of automata in $\mathcal{W}_2$ given in the previous section, we present an algorithm to learn $\mathcal{W}_2$. Let M^* denote the target automaton. We use $M^*(\vec{x})$ to denote the function computed by M^*, *i.e.,* $M^*(\vec{x}) = 1$ if M^* accepts on input $\vec{x}$ and $M^*(\vec{x}) = 0$ otherwise.

Linear functions can be learned by solving systems of equations [10]. It is also easy to construct a linear automaton from a linear function. Let the algorithm for learning linear automata be *linear-explain* (i, S), where S is a labeled set of examples. This algorithm considers the set of examples

$S_i = \{\langle \vec{x}_i, M^*(\vec{x})\rangle | \langle \vec{x}, M^*(\vec{x})\rangle \in S\}$ and returns a linear automaton A that is consistent with S_i or returns *ERROR* if there is no linear automaton consistent with S_i.

Our algorithm **Learn_width-2** takes as input a set D of labeled examples generated by the target automaton and returns an automaton M that is consistent with D (*i.e.,* all $\langle \vec{x}, M^*(\vec{x})\rangle \in D$). We refer to this as "$M$ explains $\vec{x}$ " for each $\langle \vec{x}, M^*(\vec{x})\rangle \in D$. The algorithm is shown in Figure 3. It consists of two phases – the first phase that builds several automata pieces M_k^b each explaining a disjoint (but totally exhaustive) subset of D and the second phase that combines the M_k^bs to obtain M.

Figure 4(a) shows a width-2 automaton (the target automaton), and Figure 4(b) shows the automaton (with 0 error) that is learned by our algorithm with high probability if (a sufficiently large) D consists of $\langle \vec{x}, M^*(\vec{x})\rangle$ pairs such that $\vec{x}$'s are chosen uniformly from the set of strings of length l.

The suffixes of M_k^bs may not be identical to one another, due to the possible existence of multiple automata consistent with the same data set.

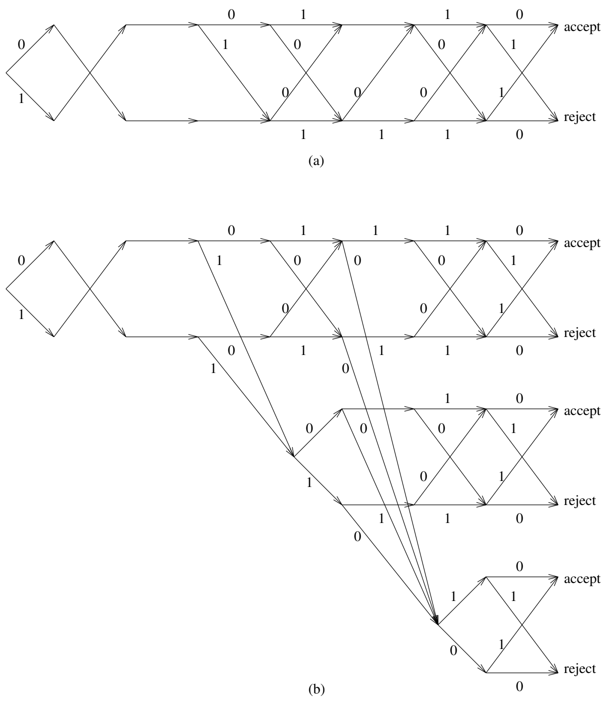

Figure 4: An Example of Learning Width-2 Automata

4.2 Correctness of the Algorithm

The algorithm tries to identify the merge stages in M^* using D. The basic idea of the first phase is: Suppose we isolate those $\vec{x}$ that are affected by a single merge stage (*i.e.,* those $\vec{x}$ with $x_k = b$ if it is a (k, b)-merging stage). We show that this isolation can be done for the last merge stage (Claim 1). Then, we find an automaton M_k^b that explains $\vec{x}_k$, and drop them from D_{k+1} to obtain D_k (pretending that the merge stage did not exist). If this does not affect the solution for the yet unexplained samples in D_k, we can proceed by finding such M_k^bs and getting rid of merge stages until there are none left, at which stage, D_k (if non-empty) is consistent with a linear automaton, which can easily be found. The correctness of this is proved in Claim 2.

We now concentrate on the construction of M_k^b. Observe that when $\vec{x}$ with $x_k = b$ reaches a (k, b)-merging stage, information about the previous stages is "forgotten". Suppose it is the last merge stage, then we can use *linear-explain* to construct an M_k^b that explains the inputs in $S = \{\langle \vec{x}, M^*(\vec{x})\rangle | x_k = b\}$, such that M_k^b depends only on the suffix $\vec{x}_{k+1}$ of the inputs. The following claim states this:

Claim 1 *If for some k, $1 \le k \le l$, (1) stage k is a (k, b)-merging stage, (2) there are no $(j, 0)$ or $(j, 1)$-merging stages for $j > k$ and if (3) $S = \{\langle \vec{x}, M^*(\vec{x})\rangle \in D_{k+1} | x_k = b\}$, then* linear-explain (k, S) *returns an automaton that is consistent with $\vec{x}_{k+1}$ when $\langle \vec{x}, M^*(\vec{x})\rangle \in S$.*

Proof: All the strings in S start from the same state after stage $k+1$. Hence, this state can be treated as a starting state for a smaller linear automaton (returned by *linear-explain*) that is consistent with $\vec{x}_{k+1}$ when $\langle \vec{x}, M^*(\vec{x})\rangle \in S$. □

If k is a (k, b)-linear stage, ideally *linear-explain* should not be able to construct M_k^bs (*i.e.,* returns *ERROR*). However, if D_{k+1} is not "rich" enough, then there might be a linear automaton that is consistent with S. In this case *linear-explain* will misinterpret k to be a (k, b)-merging stage, and will return an M_k^b. Since this interpretation is still consistent with the samples, it does not lead to any future inconsistencies in learning M^*.

If $k(\neq 1)$ is both a $(k, 0)$-merging and $(k, 1)$-merging stage, then M_k^0, M_k^1 are returned by two calls to *linear-explain*, and they are joined together to form M_1^0. In this case, the first phase terminates as $D_k = \emptyset$. On the other hand, if $k = 1$ there exists a linear automaton that explains D_1, in which case *linear-explain* can be used to learn it.

Now, we have to worry about gluing the M_k^b's appropriately to obtain M, which is done in the second phase of the algorithm. Let l_k^b denote the number of stages in M_k^b. The linking of M_k^bs is performed in such a manner that stage $l_k^b - k$ of M_k becomes part of stage $l - k$ of M. From our construction of M in the algorithm, we see that b-transitions of $M_j^b, j < k$ in stage $k - 1$ are redirected to to the start state of M_k^b. This guarantees that M has the following property: if a suffix $\vec{x}_k$ of input $\vec{x}$ was explained by M_k^b during the first phase of the algorithm, and if M is run on $\vec{x}$, the start state of M_k^b will be reached on the first letter of $\vec{x}_k$.

Recall that an r-linear automaton is one whose stages $r \ldots l$ are all non-merging. The following claim justifies our algorithm.

Claim 2 *At any stage k in the first phase of* **Learn_width-2** (D), *so long as $D \neq \emptyset$, there exists a k-linear automaton M' consistent with D_k. M' is essentially the same as M^*: for each (j, b)-merging stage in M^* where $j > k$, the corresponding stage in M' is (j, b)-non-merging, with the b-transitions arbitrarily chosen to be passes (i.e., they stay on the same track) or switches (they change tracks), and all*

```
**Learn_width-2**($D$)
    $D_{l+1} = D$
    **for** $k = l$ **downto** 1 **do**
        **for** $b = 0, 1$ **do**
            $S = \{\langle \vec{x}, M^*(\vec{x})\rangle \in D_{k+1} | x_k = b\}$
            **if** *linear-explain* $(k, S)$ returns an automaton $M_k^b$ **then**
                $D_k = D_{k+1} \backslash S$
                **if** $k > 1$ and $D_k = \emptyset$ **then**
                    create $M_1^0$ as follows:
                        create stage $q_{k-1}$ with $c$-transition to $M_k^c$ for $c = 0, 1$
                        add "don't-care" single-node stages $q_0, \ldots, q_{k-2}$,
                            each going to the next on both 0 and 1
                    **exit loop**
            **else**
                $D_k = D_{k+1}$
    **for** $k = 1$ **to** $l$ **do**
        **for** $b = 0, 1$ **do**
            **if** $\exists$ an automaton $M_k^b$ **then**
                redirect the $b$-transitions at stage $k - 1$
                    of all $M_j^b, j < k$ to the starting state of $M_k^b$
```

Figure 3: Algorithm for Learning Width-2 Automata

other transitions are the same as in M^*.

Proof: The proof is by reverse induction on k. For $k = l$, the claim is trivially true with $M' = M^*$. Assume that the claim was true at stage $k + 1$, with $(k + 1)$-linear automaton M''. Consider stage k in M^*. We have the following three cases:

- k is both a $(k, 0)$ and a $(k, 1)$-non-merging stage: If **Learn_width-2** recognizes that k is a linear stage, then $D_k = D_{k+1}$. If **Learn_width-2** recognizes stage k as a merging stage, $D_k \subset D_{k+1}$. So, $M' = M''$ will satisfy the induction hypothesis.
- k is both a $(k, 0)$ and a $(k, 1)$-merging stage: D_k becomes $\emptyset$, and the claim is trivially true.
- k is a (k, c)-merging and $(k, 1 - c)$-non-merging stage for $c \in \{0, 1\}$: Without loss of generality, let $c = 0$. Then by Claim 1, *linear-explain* will build an M_k^0 to explain all inputs $\langle \vec{x}, M^*(\vec{x})\rangle \in D_{k+1}$ with $x_k = 0$, and it will delete those inputs from D_{k+1} to obtain D_k. Construct a k-linear M' from M'' as follows: redirect one of the two merging 0-transitions at stage k of M'' to the other track in stage $k+1$ so that the two 0-transitions now go to different states. This clearly turns stage k into a linear stage, hence making M' k-linear. Note that the only difference between the behaviors of M'' and M' is on those inputs $\vec{x}$ with $x_k = 0$. However, all such inputs have been deleted from D_{k+1}. Therefore M' explains D_k, making the claim true for stage k.

□

We have thus shown that the M_k^bs we constructed explain the whole of D and that our way of connecting them to get M preserves this property. M has width $\leq 2l$. From these claims we have:

Lemma 3 *Algorithm* **Learn_width-2** *produces a hypothesis in $\mathcal{W}_{2t}, t \leq l$ that is consistent with D.*

Theorem 4 *Algorithm* **Learn_width-2** *learns width-2 automata on any distribution.*

Proof: Let δ, ϵ be the usual error parameters. Using the above Lemma and Occam's razor, since $|\mathcal{W}_{2l}| = O(2^{8l})$, after finding an $h \in \mathcal{W}_{2t}, t \leq l$ consistent with $O\left(\frac{\log 1/\delta}{\epsilon} + \frac{\log |\mathcal{W}_{2l}|}{\epsilon}\right)$ samples, we have a learning algorithm for $\mathcal{W}_2$. □

4.3 Larger Alphabet Size

Let $\mathcal{W}_k^m$ denote the concept class of width-k automata over an m-symbol alphabet. We can easily extend our algorithm to learn $\mathcal{W}_2^m$. Let the alphabet symbols be $\{0, \ldots, m - 1\}$. At stage k of the extended algorithm, we will obtain m automata pieces $M_k^0, \ldots, M_k^{m-1}$. After gluing these pieces together to obtain M, we see that $M \in \mathcal{W}_{2(m-1)l}^m$. However, $|\mathcal{W}_k^m| = O(k^{kml})$ and by using Theorem 4 our learning algorithm can be seen to be polynomial in l and m.

Thus, our results can be restated as follows: 2-state automata are efficiently learnable, learning 3-state automata is as hard as learning DNF (which we will show Section 5), and learning 5-state automata is *NP*-hard.

4.4 Proper Learning Under Uniform Distribution

In the previous section, the automaton output was $M \notin \mathcal{W}_2$. In this section, we show that proper learning of $\mathcal{W}_2$ is possible *i.e.,* we will be able to obtain a width-2 automaton. However, we need to assume uniform distribution of labeled samples.

4.4.1 The Proper Learning Algorithm

We first run **Learn_width-2** on the labeled samples D. Let $M \in \mathcal{W}_{2t}, t \leq l$ be the automaton produced by **Learn_width-2**(D). We can write the labeled samples $D = \{\langle \vec{x}, M^*(\vec{x})\rangle\}$ as $[SL]$ where S is the list of example strings and L is the vector of labels. We can identify pairs of accepting/rejecting tracks as *track-pairs*. Thus, M has $t(\leq l)$ track-pairs. By our algorithm, if M has a branch on an alphabet character (into another track-pair labeled k) at stage k, then stage k of M^* is $(k, 0)$-merging or $(k, 1)$-merging. For the rest of this section, we get rid of our earlier assumption that the top track is an accepting track.

For simplicity, we require a normal form representation for 2-track automata.

Claim 5 *Every $M \in \mathcal{W}_2$ can be expressed in a normal form where all the (k, b)-merging stages are of the form $(1 - b)$-pass.*

Proof: If a (k, b)-merging stage has a $(1-b)$-switch, we can just flip the tracks following stage k to make it $(1 - b)$-pass. □

As we saw before, we can represent linear stages as a sum of parity stages and a constant. For a (k, b)-non-merging stage (for $b = 0$ and $b = 1$) let a_k denote the linear function computed at this stage. Under normal form assumptions, we have $a_k = 0$ for a (k, b)-merging stage (by Claim 5). We can partition D into disjoint subsets corresponding to (suffixes of) those inputs that reach the k-th track-pair. Let $D_k = \{\langle \vec{x}_k, M^*(\vec{x})\rangle : \langle \vec{x}, M^*(\vec{x})\rangle \in D$ and $\vec{x}$ reaches k-th track in $M\} = [S_k L_k]$. Since M is known, these sets can easily be computed from D. Note that each D_k defines a system of equations that determine the $a_{k'}$s for the non-merging stages (in M), for $k' > k$. The motivation behind considering these sets is that if these sets are ensured to be big enough, then the non-merging stages of M are uniquely determined and isomorphic to those stages in M^*. We can then hope to collapse the t track-pairs of M into one, with the branches in M appropriately translated to merge stages in M'. Therefore, we look for conditions that would guarantee a unique solution to the system of equations defined by the D_k's. The exact conditions are presented in the next section.

Ideally, we would like to get an $M' \in \mathcal{W}_2$ that agrees with the entire sample set D. However, we relax this requirement slightly and obtain an $M' \in \mathcal{W}_2$ that agrees with M (and hence with D) on most of the input strings (*i.e.,* some $S' \subseteq S$). In particular, S' will contain (with high probability) those strings that reach the track-pairs in M that are reached by "lot" of other strings. A proper choice of parameters will ensure that the Occam sense of learning is still preserved. The number of samples S required to satisfy this is discussed in the next section.

Let ϵ, δ be the usual learning parameters. The algorithm is described below (Figure 5).

If the S_k's are sufficiently large, we will show that *linear-explain* will return a unique solution. In other words, $a_k, \ldots,$ $a_{k'}$ are uniquely determined with high probability at each step

```
Proper_Learn_width-2($M, S$)
    use $M$ to obtain $S_k,\ 1 \leq k \leq l$
    $k = l, k' = l$
    while $k \geq 1$ do
        if $|S_k| \geq \frac{l(l+2)}{\delta}$ then
            learn $a_{k+1}, \ldots, a_{k'}, c_k$ using
                linear-explain$(k + 1, S_k)$
            $a_k = 0$
            label arcs in linear stages $k + 1, \ldots, k'$
                of $M'$ using $a_{k+1}, \ldots, a_{k'}$
            label arcs in merge stage $k$
                of $M'$ using $M$ and $c_k$
            $k' = k - 1$
        else
            $k = k - 1$
```

Figure 5: Proper Learning Algorithm for Width-2 Automata

of the algorithm. Now, we have to argue that the $a_k, \ldots, a_{k'}$ are 'consistent' with all other $S_j,\ j < k$. If this is so, then after determining the appropriate arc labels (using c_k) in the merge stage a_k, we can proceed to work with the next S_k.

To establish the consistency of the computed linear stages with the input, we can view an automaton as a function of the remainder of the inputs. Define $f_k^0(\vec{x}_k)$ (resp. $f_k^1(\vec{x}_k)$) to be the function computed at stage k in M^* when starting on the top (resp. bottom) track. Call that function the solution to a set of $\vec{x}_k$'s. It is easy to see that if M is k'-linear, then, for all $k \geq k'$, $f_k^0(\vec{x}_k) = f_k^1(\vec{x}_k) + 1$.

Claim 6 *Either f_{k+1}^0 or f_{k+1}^1 is a solution to all $S_j,\ j \geq k$.*

Proof: All examples in $S_j,\ j \geq k$ either go to the top track or the bottom track at stage $k + 1$. Also, by definition, these do not pass through any merge stages after stage $k + 1$. So, one and only one of f_{k+1}^0 and f_{k+1}^1 agrees with all S_j's. □

By Claim 5, M^* is in normal form and hence the (k, b)-merging stage was a $(k, 1 - b)$-pass and so we set a_k to 0 and go to S_{k-1}. Thus, a_k is a merge stage in M' and the arc labels are set appropriately depending on the c_k obtained and the corresponding branch at stage k in M.

4.4.2 Required Sample Size

In this section, we derive the required sample size $|S|$. We have to address two issues here: S should be big enough so that *linear-explain* will return a unique solution (with high probability), and S should also be big enough so that the number of samples dropped (*i.e.,* $|S - S'|$) is rendered insignificant. The following claim states that a uniformly chosen random collection of 0-1 vectors is highly likely to have full rank.

Claim 7 *If vectors are chosen uniformly at random from $\{0, 1\}^l$ until they (viewed as a matrix) have full rank, then the expected number of vectors to be chosen is $l + 2$.*

Proof: Given that the set of vectors already picked has rank $i-1$, define a random variable X_i to be the expected number of additional vectors to be picked such that the total set of vectors picked has rank i. Then, the expected number of vectors to be picked so that we get a set of vectors with rank l is just $E[X_1+\cdots+X_l] = E[X_1]+\cdots+E[X_l]$, using the linearity of expectation. After picking a non-zero x_1 unconditionally (thus, $X_1 = \frac{2^l}{2^l-1}$), the expected number of trials to pick the second linearly independent vector is $\frac{2^l}{2^l-2} = X_2$. In a similar manner, the expected number of trials to pick the i-th independent vector (given a set of vectors of rank $i-1$) is $\frac{2^l}{2^l-2^{i-1}}$. Thus, $E[X_1+\cdots+X_n] = \sum_{i=0}^{l-1}\frac{2^l}{2^l-2^i} = l+\sum_{i=1}^{l}\frac{1}{2^i-1} \leq l+2$. □

Claim 8 *Given $\delta > 0$, if $|S_k| \geq \frac{l(l+2)}{\delta}$, then the probability that for all k, the system of equations defined by $S_k\vec{a}_k = L_k$ does not have a unique solution, is $\leq \delta$.*

Proof: Using Markov's inequality and Claim 7, the probability that for a given k the matrix S_k does not have full rank $\leq \frac{\delta}{l}$. All these bad probabilities sum to $\leq \delta$. □

From S, the algorithm picked those S_k's such that $|S_k| \geq \frac{l(l+2)}{\delta}$. Let S' be the set of samples thus chosen. Now our task is to prove that Occam learning is still valid even if some small fraction of labeled samples are not learnt. First, we make sure that we don't discard too many samples.

Claim 9 *Given $\frac{\epsilon}{2}$, if $|S| \geq \frac{2l^2(l+2)}{\epsilon\delta}$ then $|S'| \geq (1-\frac{\epsilon}{2})|S|$.*

Proof: We dropped those S_k's with $|S_k| < \frac{l(l+2)}{\delta}$. The total number of samples thus dropped $\leq \frac{l^2(l+2)}{\delta}$ which we require to be $\leq \frac{\epsilon}{2}|S|$. □

Claim 10 *Given $\epsilon > 0, 2\delta > 0$, if $|S| > \frac{2}{\epsilon^2}(l\ln 16 - \ln\delta)$ and if M' disagrees with $|S|$ on $\leq \frac{\epsilon}{2}$ of S, then the probability that it is an ϵ-bad hypothesis with respect to M^* is $\leq \delta$.*

Proof: We use similar ideas from [5], [19]. Let $M'' \in \mathcal{W}_2$ with l-stages be an ϵ-bad hypothesis with respect to M^*. Then, given a random set of samples S, the expected number of samples on which M'' disagrees with M^* is $\geq \epsilon|S|$. Using Chernoff bounds, the probability that M'' disagrees with M^* on no more than $\frac{\epsilon}{2}|S|$ of the samples is $\leq e^{-2(\frac{\epsilon}{2})^2|S|}$. Easily $|\{M \in \mathcal{W}_2 : M \text{ has } l \text{ stages }\}| \leq 2^{4l}$. Hence, the probability that we find such an M'' that disagrees with M^* on no more than $\frac{\epsilon}{2}$ of the samples is $\leq 2^{4l}e^{-\frac{1}{2}\epsilon^2|S|}$ which we want to be $< \delta$. From this, we get $|S| > \frac{2}{\epsilon^2}(l\ln 16 - \ln\delta)$. □

Finally, the following theorem follows from Claims 9 and 10 and gives the size of labeled samples required for the given parameters.

Theorem 11 *Given $\epsilon > 0, \delta > 0$, if $|S| = \max(\frac{2l^2(l+2)}{\epsilon\delta}, \frac{2}{\epsilon^2}(l\ln 16 - \ln\delta))$, then the probability that we get an ϵ-bad hypothesis with respect to M^* is $\leq \delta$.*

In our construction, note that the only place where we required uniform distribution of input samples was in Claim 8.

5 Width-3 Automata

In this section, we show that learning width-3 automata is as hard as learning DNF. We also relate the learnability of width-3 automata to a special class of width-4 automata constructed out of parity functions.

5.1 Relation to DNF

In this section, we exhibit a reduction which shows that learning width-3 automata is at least as hard as learning DNF. Our reduction is similar to the reduction in [14]. Our original reduction showed that learning $\mathcal{W}_3$ is hard as learning decision trees. Rob Schapire ([22]) has pointed out that a similar reduction can be used to relate $\mathcal{W}_3$ and DNF. We present the latter result which is stronger since learning decision trees is known to be as hard as learning DNF. The exact complexity of learning DNF is not known.

Let $\mathcal{F}$ denote the concept class of DNF. Recall that a DNF consists of a disjunction of k clauses, each of which is a conjunction of literals. We can construct a width-3 automaton M_F corresponding to a k-term DNF F as follows: Let M_F have a devoted track called A (signifying acceptance). Intuitively, A is "joined" whenever a clause is satisfied.

We build a width-2 automaton M_c for each clause c. M_c accepts if the current input satisfies c in F. Then, by taking the "or" of the k clauses, we can construct an automaton that checks if any of the k clauses is satisfied. More formally, let the input be $\vec{x}$. For each clause $c = x_{i_1} \wedge \ldots \wedge x_{i_j}$ in F, we can construct a width-2 automaton M_c such that $(x_{i_1} \wedge \ldots \wedge x_{i_j}) = 1 \Leftrightarrow M_c(\vec{x})$ accepts. M_c consists of l sections, one for each input variable in $\vec{x}$. If the variable x_i appears (resp. complemented) in c, then if we are on the upper track we stay on the upper track (resp. make a transition to the lower track) on 1 (resp. 0) and go to the lower track (resp. stay on the upper track) on 0 (resp. 1). If we are on the lower track, we stay on the lower track on both 0 and 1. If the variable x_i does not appear at all in c, we remain on the track that we are currently following. For instance, for the clause $c = x_a \wedge x_b \wedge x_c$ with $1 \leq a < b < c \leq l$, the corresponding M_c is shown in Figure 6.

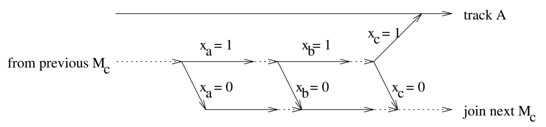

Figure 6: Converting a Clause to a Width-2 Automaton

For any two clauses c_1, c_2, the corresponding automata M_{c_1}, M_{c_2} can be juxtaposed by "joining" the accept (resp. reject) track of M_{c_1} to A (resp. M_{c_2}). Finally, M_F is constructed by juxtaposing the M_c's for each clause c in F. Clearly, M_F is a width-3 automaton corresponding to F. If $\vec{x}$ is an input to F, then the corresponding input to M_F is $\vec{x}' = \overbrace{\vec{x}\cdots\vec{x}}^{\text{k times}}$.

The above construction shows that for every $F \in \mathcal{F}$ there exists $M_F \in \mathcal{W}_3$ such that $\vec{x} \in F \Leftrightarrow \vec{x}' \in M_F$. The instance transformation $\vec{x} \mapsto \vec{x}'$ is clearly polynomial (squares the input length) and the size of the image concept in $\mathcal{W}_3$ is linear in the size of the concept in $\mathcal{F}$. Hence, our reduction is compliant with the notion of reduction as defined in [14].

5.2 An Application

Consider the class of probabilistic finite automata (PFA), which are automata in which each edge is labeled with a probability and an alphabet character. A walk on the automaton follows edges leaving the current state, chosen according to the probability labels, and outputs the alphabet character labeling that edge.

The problem of learning PFA is hard ([1], [12]). In fact, even learning width-2 PFA is known to be hard ([12]), based on the hardness of parity with noise which is the following problem: Let f be the parity function computed by a parity automaton, define a parameter $0 \leq \eta < 1/2$ called the *noise rate*. The oracle, when asked for a labeled example, randomly picks an input x according to its distribution, flips a coin whose probability of heads is $1-\eta$ and whose probability of tails is η, if the outcome is heads returns $\langle x, f(x) \rangle$, and if the outcome is tails returns the incorrectly labeled example $\langle x, 1 - f(x) \rangle$.

We do not know if there exists a class of deterministic automata that is hard to learn on the uniform distribution. However, in our search for a class of DFA that is hard to learn on the uniform distribution, we study classes of automata that attempt to deterministically simulate the parity with noise function. One class of deterministic automata that is related to the parity with noise problem is the following class $\mathcal{W}_4'$ of width-4 automata, which can be viewed as a width-2 automata with a "fork" in the middle. Intuitively, this fork models a "mistake".

Consider the case of an automaton M with l stages, such that initially it is a width-2 automaton and then a fork at stage k splits the automaton into two separate width-2 automata (Figure 7(a)). f, g, h are functions computed by various width-2 pieces of M, such that one of f or g is a parity function. Let the concept class $\mathcal{W}_4'$ consist of functions $\{\langle f, g, h \rangle | f, g, h \in \mathcal{W}_2, f \in \mathcal{P} \vee g \in \mathcal{P}\}$. We show that we can use an algorithm that learns $\mathcal{W}_3$ to learn $\mathcal{W}_4'$.

Without loss of generality, $g \in \mathcal{P}$. M' is shown in Figure 7(b). Unlabeled dashed arcs can carry a label of 0, 1, 0/1, or none (in which case they do not exist) depending on the first (or last) stage of g and h. If $\langle \vec{x}, b \rangle$ is a labeled example for $\mathcal{W}_4'$, then $\langle \vec{x}' = \vec{x}\vec{x}_k\vec{x}_k, b \rangle$ is a labeled example for $\mathcal{W}_3$.

The crux of the idea is: since $g \in \mathcal{P}$, $g(\vec{x}_k) \circ g(\vec{x}_k) = 0$. If an input $\vec{x}$ leads to computing $f \circ g$ in M (by choosing the upper branch), $\vec{x}'$ leads to computing $f \circ g$ in M' as well. If $\vec{x}$ leads to computing $f \circ h$ in M (by choosing the lower branch), $\vec{x'}$ leads to computing $f \circ h \circ g \circ g = f \circ h$ in M'. Therefore, M' accepts $\vec{x'} \Leftrightarrow M$ accepts $\vec{x}$. The instance transformation is linear (doubles the original input) and the size of the image concept in $\mathcal{W}_3$ is linear in the size of the concept in $\mathcal{W}_4'$. Note that even if k is not known, we can try to learn for each $k = 1, \ldots, l$.

As a consequence of mangling of inputs, the input distribution is not preserved in this reduction. Our construction is "tight" in the sense that the parity restraint cannot be relaxed further. Informally, if g were not a parity function (*i.e.,* it were to have a merging state), then there exist inputs $0\vec{x}_{k+1}$ and $1\vec{x}_{k+1}$ such that we would end up in (say) the accepting track. We cannot proceed as in our construction since the track in which we would end up after h is "forgotten" by g for any input with this particular adversarial suffix.

5.3 k-Mistake Parity

Consider the subclass $\mathcal{W}_4''$ of $\mathcal{W}_4'$ (defined in the previous section), where $f \in \mathcal{P}$. Then $\mathcal{W}_4''$ captures a parity error model, where the inputs on which the parity is computed erroneously are determined by parity functions on a prefix of the input, as described below. Let M be a parity automaton. Consider the situation where at stage i on input 0 (or 1) M errs by computing a different function h from that stage on. Clearly the subclass $\mathcal{W}_4''$ models this class. We have thus shown a reduction from this restricted error model to $\mathcal{W}_3$.

Our reduction can be a applied to a generalization of the above class, where M can err up to k times. We will call this class the *k-mistake parity* class. In this case, assume that M errs at stages $s_1, s_2, \ldots s_k$ by branching off to k functions $h_1, \ldots, h_k$.

Consider the automaton shown in Figure 8 (dotted lines indicate repetition of stages). Here, $f_i, g_i \in \mathcal{P}, i \leq k$. Without loss of generality, arcs leaving g_i, h_i can be assumed to be identically labeled (if not, the edges in the last stage of h_i can always be flipped). If $s_1, \ldots, s_k$ are known, then by applying our reduction in the previous section to each of the stages, we can see that learning the k-mistake parity class reduces to learning $\mathcal{W}_3$. Otherwise, if $s_1, \ldots, s_k$ are unknown, and k is a constant, then the learning algorithm can guess $s_1, \ldots, s_k$, and apply the learning algorithm for each guess.

5.4 Open Questions

We note some of the interesting issues that are not yet resolved. The issue of the complexity of learning width-3 automata remains open. Is it provably harder than learning DNF? Furthermore, what is the complexity of learning width-4 automata? Can width-2 automata be properly learned under arbitrary distributions?

Acknowledgments

We wish to thank Rob Schapire for his useful comments and suggestions, in particular, for pointing out the relationship between learning width-k automata and learning automata with very small number of states.

References

[1] N. Abe and M. Warmuth. On the Computational Complexity of Approximating Distributions by Probabilistic Automata. *Machine Learning*, 9:205-260, 1992.

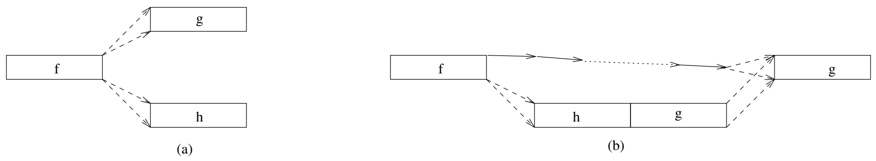

Figure 7: The Reduction

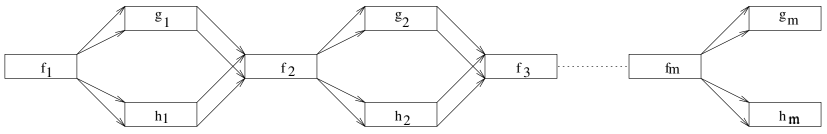

Figure 8: k-Mistake Parity

[2] D. Angluin. On the Complexity of Minimum Inference of Regular Sets. *Information and Control*, 39:337-350, 1978.

[3] D. Angluin. A Note on the Number of Queries Needed to Identify Regular Languages. *Information and Control*, 51:76-87, 1981.

[4] D. Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75:87-106, 1987.

[5] D. Angluin and P. Laird. Learning from Noisy Examples. *Machine Learning*, 2:343-370, 1988.

[6] D. Barrington. Bounded-Width Polynomial-Size Branching Programs Recognize Exactly Those Languages in NC^1. *JCSS*, 38:150-164. 1989.

[7] A. Borodin, D. Dolev, F. Fich, and W. Paul. Bounds for Width Two Branching Programs. *SIAM J. Computing*. 15:549-560. 1986.

[8] Y. Freund, M. Kearns, D. Ron, R. Rubinfeld, R. Schapire, and L. Sellie. Efficient Learning of Typical Finite Automata from Random Walks. In *Proc. 25th STOC*, pages 315-324. ACM, 1993.

[9] E. Gold. Complexity of Automaton Identification from Given Data. *Information and Control*, 37:302-320, 1978.

[10] D. Helmbold, R. Sloan, and M. K. Warmuth. Learning Integer Lattices. *SIAM J. Computing*, 21:240-266, 1992.

[11] M. Kearns, M. Li, L. Pitt, and L. Valiant. On the Learnability of Boolean Formulae. In *Proc. 19th STOC*, pages 285-295. ACM, 1987.

[12] M. Kearns, Y. Mansour, D. Ron, R. Rubinfeld, R. Schapire, and L. Sellie. Efficient Learning of Typical Finite Automata from Random Walks. In *Proc. 26th STOC*, pages 273-282. ACM, 1994.

[13] M. Kearns and L. Valiant. Cryptographic Limitations on Learning Boolean Formulae and Finite Automata. In *Proc. 21st STOC*, pages 433-444. ACM. 1989.

[14] L. Pitt and M. Warmuth. Prediction-preserving Reducibility. *JCSS*, 41:430-467, 1990.

[15] L. Pitt and M. Warmuth. The Minimum Consistent DFA Problem Cannot be Approximated Within Any Polynomial. *J. ACM*, 40:95-142, 1993.

[16] L. Pitt and L. Valiant. Computational Limitations on Learning from Examples. *J. ACM*, 35:965-984, 1988.

[17] R. Rivest and S. Schapire. Diversity-based Inference of Finite Automata. In *Proc. 28th FOCS*, pages 78-87. IEEE, 1987.

[18] R. Rivest and S. Schapire. Inference of Finite Automata Using Homing Sequences. In *Proc. 21st STOC*, pages 411-420. ACM, 1989.

[19] D. Ron and R. Rubinfeld. Learning Fallible Finite State Automata. In *Proc. 6th COLT*, pages 218-227. ACM, 1993.

[20] D. Ron, Y. Singer, and N. Tishby. The Power of Amnesia. In *Advances in Neural Information Processing Systems*, Volume 6, Morgan Kauffman, 1993.

[21] D. Ron, Y. Singer, and N. Tishby. On the Learnability and Usage of Acyclic Probabilistic Finite Automata. Manuscript, January 1995.

[22] R. Schapire. Personal Communication. 1995.

[23] R. Schapire and M. Warmuth. Personal Communication. 1990.

[24] H. Schutze and Y. Singer. Part-of-Speech Tagging Using a Variable Memory Markov Model, In *Proc. 32nd ACL*, 1994.