
On Learning Width Two Branching Programs

–Extended Abstract–

Nader H. Bshouty Christino Tamon David K. Wilson

Department of Computer Science
The University of Calgary
2500 University Drive NW
Calgary, Alberta, Canada T2N 1N4

e-mail:{bshouty, tamon, wilsond}@cpsc.ucalgary.ca

Abstract

We prove that strict width two branching programs or \mathcal{SW}_2 (which are width two branching programs with exactly two sinks, as defined in [BDFP86]) are properly PAC learnable under any distribution. We also observe that PAC learning monotone width two branching programs (which are width two branching programs with exactly one rejecting sink) is as hard as learning DNF formulae.

This work refines both the positive and negative results in [ERR95] and answers one of the open questions in that paper.

1 Introduction

Many interesting results have been found due to the study of branching programs most notably by Barrington [B89] who demonstrated that a very restricted form (width five) can accept all languages contained in NC^1 .

A branching program is an acyclic digraph where each node is labelled by a Boolean variable from $X = \{x_1, \dots, x_n\}$ or the values 0 or 1. There is one designated starting node and an arbitrary number of sinks labelled with either 0 or 1. Boolean functions are computed on a particular assignment to the variables in X by returning the label of the sink that is found when these assignment values are used to trace a route through the graph. One form of branching program, mentioned above, are those that are restricted to having a maximum number of nodes on any given level of the digraph (bounded width). Intuitively, the level of a node is its distance from the starting node. These ideas will be made precise later.

Learning unrestricted branching programs in polynomial time appears to be extremely difficult. Angluin and Kharitonov [AK91] have shown that, under certain cryptographic assumptions, general Boolean formulae are not learnable using random examples even when membership queries are avail-

able. Since any Boolean formula can be transformed into a branching program of at most the same size, the same hardness result applies to branching programs. Even learning width five branching programs appears to be difficult. Combining the results of [B89] and [KV89] implies that learning width five branching programs is hard, again under cryptographic assumptions.

Obviously, given the above, any study of the learnability of branching programs should focus on restricted versions. One type of restriction, that we do not address, imposes a bound on the number of times a variable may be used to label a node in the program. Read once branching programs allow each variable to occur only once and these types of programs were shown to be learnable in Angluin's exact model [A88] by Raghavan and Wilkins [RW93]. Our focus falls to those branching programs having a bounded width. The following will show that it is necessary to continue the restriction down to a width of two.

Borodin et al. [BDFP86] call width two branching programs *monotone* if they contain only one sink that is labelled 0. Clearly, a polynomial-sized DNF formula can be computed by a monotone width two branching program with a polynomial number of nodes. Learning these would imply the learnability of DNF formulae which is a long standing open question of the field. A *strict* width two branching program contains exactly one sink labelled 0 and exactly one sink labelled 1 and without loss of generality these two nodes are located in the last level of the program.

Recently, Ergün et al. [ERR95] have shown how to learn strict width two branching programs where each level i contains nodes labelled only by variable x_i . We call these branching programs *ordered* strict width two. Notice this condition further restricts the branching program by limiting the number of times a variable may appear as well as where it may occur. Their algorithm is distribution free but outputs a hypothesis branching program whose width may be larger than two. For the specific case of a uniform distribution on the example space they give an algorithm that is proper.

We show that strict width two branching programs are properly PAC learnable under any distribution. This result is based on a new characterization of strict width two branching programs as a specific type of decision lists. We then employ an algorithm similar to that of Rivest [R87] to gain our result.

These techniques can also be used to properly PAC learn ordered strict width two branching programs which solves an open question put forth in [ERR95].

We summarize our results in relation to the results of [ERR95] in Figure 1 which depicts a hierarchy of width two branching programs. Our diagram includes one class that we have not defined yet. These are *levelled* strict width two branching programs which require nodes on the same level to be labelled by the same variable. We show how to learn these types of branching programs and then provide a transformation that implies the learnability of strict width two branching programs.

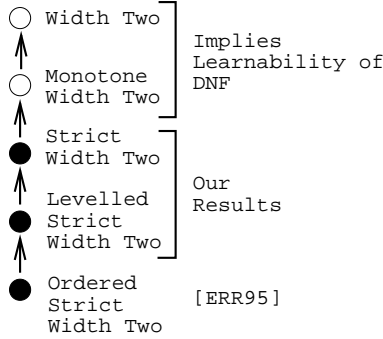


Figure 1: A Hierarchy of Width Two Branching Programs

The remainder of the paper is organized as follows. Section 2 contains preliminary definitions including a more detailed definition of branching programs and a review of the PAC learning model. Section 3 presents a new characterization of strict width two branching programs in terms of decision lists and, finally, section 4 shows exactly how our learning algorithm works.

2 Preliminaries

As mentioned above we are interested in Boolean functions with a domain $X = \{x_1, \dots, x_n\}$. Possible assignments to these variables come from the set of length n vectors over $\{0, 1\}$. Each variable is associated with two *literals*; one being the variable itself, x_i , and the other, its negation, \bar{x}_i . A *concept* c is a subset of $\{0, 1\}^n$ and is typically thought of as a function $c : \{0, 1\}^n \rightarrow \{0, 1\}$ where c outputs 1 if the input belongs to the concept and outputs 0 otherwise. A *labelled example* of c is an ordered pair consisting of a vector from $a \in \{0, 1\}^n$ accompanied with $c(a)$.

A branching program is defined as an acyclic digraph possessing the following properties. One node is designated the start node or source. Each node of non-zero out-degree is labelled with a Boolean variable from $X = \{x_1, \dots, x_n\}$ and all sinks are labelled with either 0 or 1. Each node has out-degree at most two. Those nodes with out-degree two have one edge labelled 0 and the other labelled 1. If only one edge leaves a node it is not labelled.

A branching program P computes a function, given an assignment $a = (a_1, \dots, a_n) \in \{0, 1\}^n$ for the Boolean vari-

ables in X , as follows. The computation begins from the source node. If at any point the computation has reached a node labelled x_i then the computation continues as follows. If only one edge leaves node x_i then the computation follows it. If two edges leave node x_i then the computation will follow the edge labelled 1 if $a_i = 1$ and the edge labelled 0 if $a_i = 0$. The computation proceeds until a sink is reached with the value of the function equalling its label. (Note that this applies to the deterministic case which is all we are concerned with here.)

If the nodes of a digraph can be partitioned into levels L_0, L_1, \dots such that for all i , any edge leaving a node in level L_i will enter some node in level L_{i+1} , then the digraph is called *levelled*. The notion of *width* is associated with a levelled digraph and is the maximum number of nodes contained in any level. For any integer $k > 0$ a width k branching program is a branching program with a width of at most k . The definition of strict width two branching programs over X have been defined previously and are denoted by \mathcal{SW}_2 .

We briefly review the definition of the PAC learning model that was introduced by Valiant [V84]. The learning algorithm is given a set of labelled examples (the samples) of a target function f drawn according to an unknown but fixed probability distribution D . On completion the learning algorithm returns a hypothesis h for f from a concept class H . The error of h is the probability that $h(a) \neq f(a)$ for any $a \in \{0, 1\}^n$ drawn according to D .

Letting $|f|$ be the size of the smallest standard encoding for the target $f \in C$ we say that algorithm A PAC learns C by H if for all targets $f \in C$, distributions D , and error parameters ϵ and δ , the following holds. There exist polynomials s and t in parameters $n, \frac{1}{\epsilon}, \frac{1}{\delta}$ and f such that given a set of labelled examples of size s , A runs in at most time t and outputs a hypothesis $h \in H$ which with probability $1 - \delta$ has error at most ϵ . An algorithm *properly* PAC learns when $H = C$.

The symbol $+$ represents the exclusive-or operation (XOR). The symbol \oplus_k represents the class of formulas over X containing at most k literals using the basis $\{+\}$. The class \oplus puts no restriction on the number of literals allowed in the formula.

A *Decision list* is a list L of pairs $(f_1, v_1), \dots, (f_r, v_r)$ where each f_i is a Boolean function over X , each v_i is a value in $\{0, 1\}$ and the last function f_r is the constant 1 function. A decision list L defines a Boolean function as follows. For an assignment $a \in \{0, 1\}^n$, $L(a)$ is equal to v_i where i is the least index such that $f_i(a) = 1$. A decision list can easily be illustrated as a degenerate tree whose internal nodes contain the functions f_i and whose leaves contain the values of v_i . The class of decision lists (C_1, C_2) - \mathcal{DL} are decision lists whose internal nodes contain functions from class C_1 and whose leaves contain functions from class C_2 . Both C_1 and C_2 are over the same variable set.

3 A Characterization of \mathcal{SW}_2

In this section we present a new characterization of \mathcal{SW}_2 that will help us prove that \mathcal{SW}_2 is properly PAC learnable in the

distribution-free model. We start by stating a lemma due to Borodin et al. [BDFP86].

Lemma 1 [BDFP86] *Strict width two branching programs are the smallest class of Boolean functions containing the constant functions 0 and 1 which has the following closure properties: if a and b are literals or constants and $f \in \mathcal{SW}_2$ then*

1. $f \wedge (a + b) \in \mathcal{SW}_2$, and
2. $f + a \in \mathcal{SW}_2$.

Using the above we show that the class \mathcal{SW}_2 is equivalent to the class $(\oplus_2, \oplus)\text{-}\mathcal{DL}$ which are decision lists whose nodes are labelled with a parity of at most two literals and whose leaves are labelled with unrestricted parities. This is proven in the following lemma.

Lemma 2 *Let $X = \{0, 1\}^n$. The class of strict width two branching programs over X is equivalent to the class of $(\oplus_2, \oplus)\text{-}\mathcal{DL}$ over X , i.e.,*

$$\mathcal{SW}_2 = (\oplus_2, \oplus)\text{-}\mathcal{DL}.$$

Proof First we show that $\mathcal{SW}_2 \subseteq (\oplus_2, \oplus)\text{-}\mathcal{DL}$. Assume that $f \in \mathcal{SW}_2$. The proof proceeds by structural induction on f using the cases in Lemma 1.

1. If f is a constant function, it has an obvious representation from $(\oplus_2, \oplus)\text{-}\mathcal{DL}$, i.e., it is either $(1, 1)$ or $(1, 0)$.
2. Case $f = g \wedge (a + b)$, where $g \in \mathcal{SW}_2$:
By induction $g \in (\oplus_2, \oplus)\text{-}\mathcal{DL}$, so by appending a node labelled with $a + b$ to the front of the decision list for g we get a decision list for f , (see Figure 2).

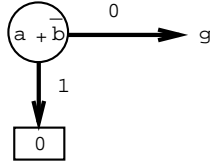


Figure 2: Transformation Construction.

3. Case $f = g + a$, where $g \in \mathcal{SW}_2$:
By induction $g \in (\oplus_2, \oplus)\text{-}\mathcal{DL}$, so the decision list for f can be obtained from the decision list of g by XORing each leaf with a .

Now we will show that $(\oplus_2, \oplus)\text{-}\mathcal{DL} \subseteq \mathcal{SW}_2$ using double induction on the number of nodes in the list, l_0 , and the number of literals appearing in the first leaf, l_1 . We will write $l_0(f)$, $l_1(f)$ to denote the number of nodes and the number of literals in the first leaf in the decision list for f , respectively.

First we prove the case when $l_0 = 1$ and l_1 is arbitrary. This is simply a parity of literals which is clearly in \mathcal{SW}_2 (by repeated application of rule (2) in Lemma 1). Now assume that the claim is true for all functions in $(\oplus_2, \oplus)\text{-}\mathcal{DL}$ with $l_0 < k$. Let f be a function in $(\oplus_2, \oplus)\text{-}\mathcal{DL}$ with $l_0 = k$. To

prove that $f \in \mathcal{SW}_2$, we use induction on l_1 . For $l_1 = 0$, this is the case when the first leaf of f is labelled by a constant $c \in \{0, 1\}$. Let y be the label of the first node in the decision list for f . Let g be the function computed by the decision list of f excluding the first node and the first leaf. Note that $g \in \mathcal{SW}_2$ since $l_0(g) < k$. If $c = 0$ then $f = g \wedge \bar{y} \in \mathcal{SW}_2$ by Lemma 1 because $g \in \mathcal{SW}_2$ and y is in \oplus_2 . If $c = 1$ then

$$f = g \vee y = \overline{(g \wedge \bar{y})} = (\bar{g} \wedge \bar{y}) + 1.$$

But then, again by Lemma 1, we have

$$\bar{g} = g + 1 \in \mathcal{SW}_2 \Rightarrow \bar{g} \wedge \bar{y} \in \mathcal{SW}_2 \Rightarrow (\bar{g} \wedge \bar{y}) + 1 \in \mathcal{SW}_2.$$

For $l_1 > 0$, assume that any function in $(\oplus_2, \oplus)\text{-}\mathcal{DL}$ with $l_0 = k$ and $l_1 < k'$ is also in \mathcal{SW}_2 . Now let $f \in (\oplus_2, \oplus)\text{-}\mathcal{DL}$ satisfy $l_0 = k$ and $l_1 = k'$. Since $l_1 > 0$ there must be a literal a that appears in the first leaf of f . Consider the function $f' = f + a$. By induction $f' \in \mathcal{SW}_2$ since $l_1(f') < k'$ and $l_0(f') = k$. But then, by Lemma 1, $f = f' + a \in \mathcal{SW}_2$. ■

We now state a corollary to the above lemma that we will need in our proper PAC learning algorithm for \mathcal{SW}_2 (since we will learn any $f \in \mathcal{SW}_2$ as a decision list in $(\oplus_2, \oplus)\text{-}\mathcal{DL}$).

Corollary 3 *There is a polynomial time algorithm that can convert any function $f \in (\oplus_2, \oplus)\text{-}\mathcal{DL}$ into a function $g \in \mathcal{SW}_2$ so that $f \equiv g$, and vice versa.*

Proof (Sketch) Follows from the above lemma. ■

4 Properly PAC Learning \mathcal{SW}_2

This section presents the details of the proper PAC learning algorithm for \mathcal{SW}_2 . We first note that the class of linear (parity) concepts is properly PAC learnable under any distribution, (see [HSW92]).

Fact 4.1 *The concept class $\oplus = \{a^T x + b \mid a \in \{0, 1\}^n, b \in \{0, 1\}\}$ is properly PAC learnable.*

Proof (Sketch) Suppose that the target concept is $f(x) = a^T x + b$. We can change each example y to $(y, 1)$ and learn a vector c of length $n + 1$ that satisfies $c^T(x, 1) = a^T x + b$. Hence, from now on we will always assume that $b = 0$. The standard algorithm for learning \oplus is to collect enough examples that are linearly independent over $GF(2)^n$. If the number of linearly independent instances are strictly less than n then we can always complete them into a basis for $GF(2)^n$. The classification of the target on these additional instances can be chosen arbitrarily. Thus we may obtain a unique solution a to the full-rank linear system of $n \times n$ equations. This unique solution defines a parity equation $a^T x$ that consistently classifies the sample set according to the target concept. ■

To prove that $(\oplus_2, \oplus)\text{-}\mathcal{DL}$ is properly PAC learnable it will suffice to prove that $(\oplus_1, \oplus)\text{-}\mathcal{DL}$ is properly PAC learnable by a standard learning reduction. For completeness we prove this reduction in the following lemma.

Lemma 4 *If $(\oplus_1, \oplus)\text{-}\mathcal{DL}$ is properly PAC learnable then so is $(\oplus_2, \oplus)\text{-}\mathcal{DL}$.*

Proof (Sketch) The idea is to substitute for each pair of variables $x_i \oplus x_j$, a new variable $x_{i,j} = x_i \oplus x_j$, and to run the PAC learning algorithm for (\oplus_1, \oplus) - \mathcal{DL} on filtered examples. More formally, map each labelled example (x_1, \dots, x_n, b) to $(x_1, \dots, x_n, x_{1,1}, \dots, x_{n,n}, b)$. The learning algorithm for (\oplus_1, \oplus) - \mathcal{DL} will output a hypothesis decision list h over $x_1, \dots, x_n, x_{1,1}, \dots, x_{n,n}$ and then the algorithm for (\oplus_2, \oplus) - \mathcal{DL} will simply replace any occurrences of $x_{i,j}$ with $x_i \oplus x_j$ to get a hypothesis decision list over x_1, \dots, x_n . ■

Now we are ready to complete the picture by proving that (\oplus_1, \oplus) - \mathcal{DL} is properly PAC learnable.

Lemma 5 *The class (\oplus_1, \oplus) - \mathcal{DL} is properly PAC learnable.*

Proof (Sketch) We follow Rivest’s technique for learning decision lists in the distribution-free PAC model [R87]. We call a pair (x_i, ξ) of variable x_i and bit $\xi \in \{0, 1\}$ “good” if the examples of S that satisfy $x_i = \xi$ can be explained by some parity equation $a^T x + b$. Note that a good pair always exists since the first node in the target concept induces a good pair. Hence the algorithm will proceed by checking all possible pairs (x_i, ξ) to find a good one. Once a good pair (x_i, ξ) is found, we can augment the decision list with a node x_i and a leaf labelled with the parity equation $a^T x + b$ that explains the instances consistent with $x_i = \xi$. At this point, we need only to explain the instances from the sample that satisfy $x_i = 1 - \xi$. We can argue in the same manner as in [R87] that this process can be continued.

Also note that the size of the class (\oplus_1, \oplus) - \mathcal{DL} is at most $n!(2^{n+1})^{n+1}$. By Occam’s razor [BEHW87] it is sufficient to collect the following number of sample points:

$$m = \frac{1}{\epsilon} \left(\log |(\oplus_1, \oplus)\text{-}\mathcal{DL}| + \log \frac{1}{\delta} \right) = O \left(\frac{1}{\epsilon} (n^2 + \log \frac{1}{\delta}) \right). \blacksquare$$

The following theorem, which is our main result, is implied by the preceding discussion of this section and Corollary 3.

Theorem 6 *The concept class \mathcal{SW}_2 of strict width two branching programs is properly PAC learnable under any distribution with sample complexity*

$$O \left(\frac{1}{\epsilon} (n^2 + \log \frac{1}{\delta}) \right).$$

Proof Note that by using the straightforward reduction of Lemma 4 we get a sample complexity of $O \left(\frac{1}{\epsilon} (n^4 + \log \frac{1}{\delta}) \right)$. To get the claimed sample complexity, we note that the length of any decision list in (\oplus_2, \oplus) - \mathcal{DL} is at most n . This is because if the list is of the form

$$((f_1, b_1), \dots, (f_m, b_m))$$

where $m > n$, then there is an f_r that is “affinely” dependent on f_i , $i \in \{1, 2, \dots, r-1\}$, i.e., $f_r = \sum_{j < r} \lambda_j f_j + \xi$, where $\lambda_1, \dots, \lambda_{r-1}, \xi \in \{0, 1\}$. Since all assignments that reach f_r satisfy $f_1 = \dots = f_{r-1} = 0$, the value of the

test f_r is determined, so it can be removed. Using this, the size of the class (\oplus_2, \oplus) - \mathcal{DL} can be bounded from above by $\binom{n^2}{n} n! (2^{n+1})^{n+1}$. Finally, note that the logarithm of this size is still $O(n^2)$. ■

Dedication

We would like to dedicate this paper to the memory of Roman Smolensky who passed away last year.

References

- [A88] Dana Angluin. Queries and Concept Learning. *Machine Learning*, 2:319–342, 1988.
- [AK91] Dana Angluin and Michael Kharitonov. When Won’t Membership Queries Help? In *Proceedings of the ACM Symposium on Theory of Computing*, pages 444–454, 1991.
- [B89] David Barrington. Bounded-Width Polynomial-Size Branching Programs Recognize Exactly Those Languages in NC^1 . *JCSS*, 38:150–164, 1989.
- [BEHW87] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K. Warmuth. Occam’s Razor. In *Information Processing Letters*, 24:377–380, 1987.
- [BDFP86] Allan Borodin, Danny Dolev, Faith Fich and Wolfgang Paul. Bounds For Width Two Branching Programs. In *SIAM Journal on Computing*, 15(2):549–560, 1986.
- [ERR95] Funda Ergün, S. Ravi Kumar and Ronitt Rubinfeld. On Learning Bounded-Width Branching Programs. In *Proceedings of the 8th Annual ACM Conference on Computational Learning Theory*, pages 361–368, 1995.
- [HSW92] David Helmbold, Robert Sloan, and Manfred K. Warmuth. Learning Integer Lattices. In *SIAM Journal on Computing*, 21(2):240–266, 1992.
- [K90] Michael Kearns. *The Computational Complexity of Machine Learning*. MIT Press, 1990.
- [KV89] Michael Kearns and Leslie Valiant. Cryptographic Limitations on Learning Boolean Formulae and Finite Automata. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 433–444, 1989.
- [L88] Nick Littlestone. Learning Quickly When Irrelevant Attributes Abound: A New Linear-Threshold Algorithm. *Machine Learning*, 2:285–318, 1988.
- [RW93] Vijay Raghavan and Dawn Wilkins. Learning μ -Branching Programs with Queries. In *Proceedings of the 6th Annual Workshop on Computational Learning Theory*, pages 27–36, 1993.
- [R87] Ronald L. Rivest. Learning Decision Lists. *Machine Learning*, 2:229–246, 1987.
- [V84] Leslie G. Valiant. A Theory of the Learnable. *Communications of the ACM*, 27(11):1134–1142, November 1984.