What circuit classes can be learned with non-trivial savings?

Rocco A. Servedio[*]
Columbia University
rocco@cs.columbia.edu

Li-Yang Tan[†]
Toyota Technological Institute
liyang@cs.columbia.edu

March 27, 2017

Abstract

Despite decades of intensive research, efficient — or even sub-exponential time — distribution-free PAC learning algorithms are not known for many important Boolean function classes. In this work we suggest a new perspective for algorithmic research on these learning problems, inspired by a surge of recent research in complexity theory, in which the goal is to determine whether and how much of a savings over a naive 2^n runtime can be achieved.

We establish a range of exploratory results towards this end. In more detail,

1. We first observe that a simple approach building on known uniform-distribution learning results gives non-trivial distribution-free learning algorithms for several well-studied classes including AC^0, arbitrary functions of a few linear threshold functions (LTFs), and AC^0 augmented with mod_p gates.
2. Next we present an approach, based on the method of random restrictions from circuit complexity, which can be used to obtain several distribution-free learning algorithms that do not appear to be achievable by approach (1) above. The results achieved in this way include learning algorithms with non-trivial savings for LTF-of-AC^0 circuits and improved savings for learning parity-of-AC^0 circuits.
3. Finally, our third contribution is a generic technique for converting lower bounds proved using Nečiporuk's method to learning algorithms with non-trivial savings. This technique, which is the most involved of our three approaches, yields distribution-free learning algorithms for a range of classes where previously even non-trivial uniform-distribution learning algorithms were not known; these classes include full-basis formulas, branching programs, span programs, etc. up to some fixed polynomial size.

[*] Supported by NSF grants CCF-1420349 and CCF-1563155.

[†] This research was done while visiting Columbia University. Supported by NSF grant CCF-1563122.

1 Introduction

Simple concepts are easy to learn, while complicated ones are harder to learn. Some of the earliest and most fundamental work in computational learning theory aims at elucidating this truism from a theoretical computer science perspective: can we understand how the algorithmic complexity of learning Boolean functions (i.e. the running time required by learning algorithms) scales with the computational complexity of the functions being learned? Achieving such an understanding was first articulated as an explicit goal in (indeed, arguably *the* explicit goal of) Valiant's landmark paper "A theory of the learnable" [Val84]:

> "The results of learnability theory would then indicate the maximum granularity of the single concepts that can be acquired without programming. In summary, this paper attempts to explore the limits of what is learnable as allowed by algorithmic complexity. The identification of these limits is a major goal of the line of work proposed in this paper."

So, more than thirty years later, how has learning theory fared in achieving these goals? Perhaps disappointingly, the roster of concept classes (classes of Boolean functions over $\{0,1\}^n$) for which efficient learning algorithms have been developed in Valiant's original distribution-independent PAC learning model, or in other similarly general learning frameworks, is quite short. Classes that are known to be learnable in polynomial time include linear threshold functions (LTFs) and degree-k polynomial threshold functions for $k = O(1)$ [BEHW89] (subsuming the classes of k-CNF, k-DNF [Val84] and k-decision lists [Riv87]); parity functions and $\mathbb{F}_2$ polynomials of constant degree [HSW92, FS92], and not much more. (If membership queries are allowed, then a few other classes are known to be distribution-independent PAC learnable in polynomial time, such as decision trees of polynomial size [Bsh95, BBB$^+$96] and regular languages computed by polynomial-size DFAs [Ang87].) In fact, only a relatively small number of natural Boolean functions classes are additionally known to be learnable even if we only require sub-exponential time for the learning. DNF formulas of $\mathrm{poly}(n)$ size can be learned in $2^{\tilde{O}(n^{1/3})}$ time [KS04], $\mathrm{poly}(n)$-sparse $\mathbb{F}_2$ polynomials can be learned in $2^{\tilde{O}(n^{1/2})}$ time [HS07], and de Morgan formulas of size s can be learned in time $n^{O(s^{1/2})}$ [Rei11].

Even simple generalizations of the above-mentioned subexponential-time-learnable function classes have remained frustratingly out of reach for the distribution-independent PAC model. Prominent examples here include the class of $\mathrm{poly}(n)$-size depth-3 AC^0 circuits, and intersections of even just two LTFs over $\{0,1\}^n$: despite extensive research effort, no positive algorithmic results are known for these classes (hence, needless to say, for their generalizations as well). This is quite a disappointment, given the rich variety of natural Boolean function classes that have been intensively studied in concrete complexity over the past several decades: well-known examples include AC^0 (augmented in various ways with more exotic gates such as mod_p gates, majority gates, threshold gates, and the like), SYM^+ circuits, various classes of branching programs, functions of a few LTFs, and the like. These functions play a starring role in concrete complexity theory, but learning theorists cannot even score an autograph.[1]

[1] We note that in the (significantly) easier *uniform-distribution* learning model, in which the learner need only succeed w.r.t. the uniform distribution over $\{0,1\}^n$, positive learning results are known for functions of a few LTFs [KOS04, GKM12] and AC^0 [LMN93] as well as some generalizations of these classes [GS10, CIKK16]; we will have occasion to revisit these results later.

This work: A change of perspective. In this paper we propose a new point of view for algorithmic research on the challenging learning problems discussed above. Since the quest for polynomial or even sub-exponential time distribution-independent learning algorithms has been unsuccessful, we suggest that a more fruitful perspective may be to *study the question of whether, and how much of, a savings over a naive 2^n running time can be achieved for these learning problems.* We thus are interested in obtaining learning algorithms that run in time $2^{n-s(n)}$ for some *savings* function $s(n)$ which is as large as possible. (To use a hackneyed metaphor, instead of expecting a brim-full glass, we are now hoping for a mouthful of water at the bottom...)

While this appears to be a new perspective for algorithmic research on challenging Boolean function learning problems, we stress that the point of view which we advocate here has been a mainstay for other (non-learning) problems for a long time. Well-known results give non-trivial upper bounds (mildly better than 2^n) on the running time of satisfiability or (exact) counting algorithms for k-CNFs, general CNFs, and a host of NP-hard or $\#\mathsf{P}$-hard problems (see e.g. [PPZ99, Sch99, PPSZ05, Sch05, FK10, IMP12, BIS12, Bjö12, IW12]). Williams's breakthrough connection [Wil10, Wil11] linking non-trivial savings of satisfiability algorithms to circuit lower bounds has intensified the interest in results of this sort for richer circuit classes, and even more recently there has been a surge of interest in questions of a similar flavor because of the connections that have been established between hypotheses like SETH and prominent questions in algorithm design (see e.g. the survey of [Wil15]).

We additionally note that several recent works [KKO13, OS16] have shown that non-trivial learning algorithms for Boolean function classes in various contexts imply circuit lower bounds. The current paper differs from these complexity-theoretic works by virtue of its explicitly algorithmic flavor; to the best of our knowledge, ours is the first work that aims at developing (and develops) $2^{n-o(n)}$-time algorithms for specific challenging Boolean function classes.

In this paper we give some first algorithmic results in the study of what circuit classes can be learned with non-trivial savings; happily, it turns out that this new perspective yields a rich bounty of positive results. As our main contribution, we present three techniques and show how each technique yields new learning algorithms of the sort we are interested in. Cumulatively, our results achieve the first non-trivial savings for many well-studied circuit classes; however, several natural questions about learning with non-trivial savings are left open by our work. We hope (and expect) that further results extending our knowledge of "non-trivially learnable" function classes will follow.

A quick and dirty proof of concept. Before describing our main results, for the sake of intuition we sketch a simple argument showing that AC^0 indeed admits a non-trivial distribution-free learning algorithm (more precisely, one whose running time is $2^{n-n^{\Omega(1/d)}}$ for poly(n)-size depth-d AC^0). The argument is based on Håstad's switching lemma [Hås86] which, roughly speaking, states that any depth-d, poly(n)-size AC^0 circuit F collapses to a shallow decision tree with very high probability under a random restriction. This can be shown to imply that if $\{0,1\}^n$ is partitioned into translations of a random subcube (corresponding to all possible settings of the live variables of a random restriction), then with very high probability almost every such subcube has the property that if F is restricted to the subcube, then F collapses to a shallow decision tree. Since it is possible to learn such a shallow decision tree relatively efficiently (in time much less than the number of points in its domain, i.e. in the subcube), this means that by learning F separately on each subcube it is possible to achieve a significant savings over brute-force search on every "good" subcube, i.e. on almost every subcube. Trading off the fraction of bad subcubes (which corresponds to the failure probability of the switching lemma, and decreases with the dimension of the subcubes) against the number of subcubes (which provides a lower bound on the running time of this learning approach, and which increases as the dimension of the subcubes decreases) and working out the parameters, the running time of this simple-minded approach comes out to be $2^{n-n^{\Omega(1/d)}}$.

Two comments: First, we note that we will improve significantly on this running time in Section 3, using a more sophisticated instantiation of this idea, and will achieve this improved running time even for various augmentations of AC^0 circuits. Second, it may not be completely clear how to run a separate copy of a distribution-free learning algorithm on each subcube in the above sketch. This will become clear in Section 2 when we describe the formal model (based on online learning, or equivalently the model of exact learning from equivalence queries) that we will use for all of our positive results (and which is well-known to imply distribution-free PAC learnability).

Relation to previous work: compression of Boolean functions. We have already explained how our goal of achieving non-trivial savings for learning is directly inspired by work aiming towards this goal for the algorithmic problems of satisfiability and counting. Another line of research which is more closely related to our study of non-trivial learning is the recent work on "compression" of Boolean functions that was initiated by [CKK$^+$15]. A compression algorithm for a class $\mathcal{C}$ (such as the class of AC^0 circuits) is a deterministic algorithm which is given as input the 2^n-bit truth table of a function in $\mathcal{C}$, must run in time polynomial in its input length (i.e. in $2^{O(n)}$ time), and must output any Boolean circuit C, computing f, such that the size of C is less than the trivial $2^n/n$ bound.

Deterministic learning is easily seen to be at least as hard as compression; we discuss the exact relation between the two tasks in more detail in Section 2, after we have given a precise definition of our learning model. Our learning algorithms, which are randomized, imply randomized variants of almost all of the compression results in [CKK$^+$15], in several cases with new and simpler proofs. We also establish non-trivial learning results (and hence randomized compression results) for many classes for which compression results were not previously known. These classes include LTF-of-AC^0, arbitrary functions of $o(n/\log n)$ LTFs, $n^{1.99}$-size switching networks, $n^{1.49}$-size switching-and-rectifier networks, $n^{1.49}$-size non-deterministic branching programs, and $n^{1.49}$-size span programs; in fact, for the last four of these classes we obtain deterministic compression algorithms.

1.1 Our techniques and results

To begin, in Section 2.1 we make the simple observation that uniform-distribution PAC learning algorithms can be converted to exact learning algorithms with membership queries simply by "patching up" the $\varepsilon \cdot 2^n$ points where an ε-accurate hypothesis is in error. (This observation was already employed by [CIKK16] in the context of compression.) Using known uniform-distribution learning results, this straightforward approach gives non-trivial distribution-free learning algorithms for several well-studied classes including AC^0, arbitrary functions of a few LTFs, and AC^0 augmented with mod_p gates.

However, as we explain in Section 3, there are uniform-distribution learning algorithms (such as the algorithms of [GS10, Kan13] for LTF-of-AC^0 circuits) which for technical reasons do not yield exact learning algorithms with non-trivial savings. To address this, in Section 3 we show how the method of random restrictions from circuit complexity can be employed to obtain non-trivial learning algorithms in settings where the approach of Section 2.1 does not apply. Recall that, roughly speaking, the "method of random restrictions" refers to a body of results showing that certain types of Boolean functions "collapse" to simpler functions with high probability when they are hit with a random restriction fixing a random subset of input variables to randomly chosen constant values. Our approach is based on learning the simpler functions that result from random restriction and thereby obtaining an overall savings in learning the original unrestricted function. This is similar to the "quick and dirty" proof of concept sketched earlier, but by adapting a recent powerful "multi-switching" lemma of Håstad [Hås14] to our learning context, we are able to achieve a significantly better savings than the "quick and dirty" argument which uses only the original [Hås86] switching lemma. Via this approach we obtain exact learning algorithms for LTF-of-AC^0 and parity-of-AC^0 that match the savings of our learning algorithm for AC^0 from Section 2.1. As indicated above the uniform-distribution approach of Section 2.1 does not give a result for LTF-of-AC^0, while for parity-of-AC^0 circuits our random restriction approach yields significantly improved savings over the results achieved for this class in Section 2.1. Furthermore, for both these classes our learning algorithms based on random restrictions do not require membership queries (in contrast to the uniform-distribution based approach, which does require membership queries).

Our third and most involved technique for non-trivial learning is based on Nečiporuk's celebrated lower bound method: in Section 4 we give a generic translation of lower bounds proved using Nečiporuk's method to non-trivial exact learning algorithms. Roughly speaking, Nečiporuk's lower bounds are established by showing that low complexity functions have few subfunctions (and exhibiting explicit functions that have many subfunctions, and hence must have high complexity). We give an exact learning algorithm that achieves non-trivial savings for classes of functions that have few subfunctions. A key technical component of our learning algorithm is a pre-processing-based technique for executing many copies of the classical halving algorithm in a highly efficient amortized manner. While simple, this technique appears to be new and may be of use elsewhere. We thus obtain a single unified learning algorithm that achieves non-trivial savings for a broad range of function classes, including full-basis binary formulas of size $n^{1.99}$, branching programs of size $n^{1.99}$, switching networks of size $n^{1.99}$, switching-and-rectifier network of size $n^{1.49}$, non-deterministic branching programs of size $n^{1.49}$, and span programs of size $n^{1.49}$. Our learning results recapture the [CKK+15] compression results for $n^{1.99}$-size formulas and branching programs with a new and simpler proof, and give the first compression results for the other classes of switching networks, switching-and-rectifier networks, non-deterministic branching programs, and span programs listed above.

2 Preliminaries

The learning model we consider. The distribution-independent PAC model has several parameters (a confidence parameter which is usually denoted δ, and an accuracy parameter usually denoted ε) which make precise statements of running times somewhat unwieldy. In this paper we will work in a elegant model of online mistake-bound learning [Lit88] which is well known to be equivalent to the model of exact learning from equivalence queries only [Ang88] and to be even more demanding than the distribution-independent PAC learning model [Ang88, Blu94]. A brief description of this model is as follows: Let $\mathcal{C}$ be a class of functions from $\{0,1\}^n$ to $\{0,1\}$ that is to be learned and let $f \in \mathcal{C}$ be an unknown target function. The learning algorithm always maintains a hypothesis function $h : \{0,1\}^n \to \{0,1\}$ (more precisely, a representation of h in the form of a Boolean circuit computing h). The learning process unfolds in a sequence of *trials*: at the start of a given trial,

- If $h(x) = f(x)$ for all $x \in \{0,1\}^n$ then the learning algorithm has succeeded and the process stops.

- Otherwise a *counterexample* — an arbitrary x such that $h(x) \neq f(x)$ — is presented to the learning algorithm, and the learning algorithm may update its hypothesis h before the start of the next trial.

The running time of a learning algorithm in this framework is simply the worst-case running time until the algorithm succeeds (taken over all $f \in \mathcal{C}$ and all possible sequences of counterexamples). We will also sometimes have occasion to consider an extension of this model in which at each trial the learning algorithm may, instead of receiving a counterexample, at its discretion choose instead to make a *membership query* (i.e. to submit a string $x \in \{0,1\}^n$ of its choosing to the oracle, and receive the value $f(x)$ in response). This is the well-studied framework of "exact learning from membership and equivalence queries" [Ang88].

We will also have occasion to consider randomized exact learning algorithms. We say that a randomized algorithm learns class $\mathcal{C}$ in time $T(n)$ if for any target function $f \in \mathcal{C}$, the algorithm succeeds with probability at least $1-\delta$ (over its internal coin tosses) after at most $T(n) \cdot \log(1/\delta)$ time steps. While many of the learning results we present will be for randomized exact learning algorithms, in the rest of this section for simplicity we confine our discussion to deterministic learning algorithms.

Besides being a clean and attractive learning model, learnability in the exact learning model (optionally augmented with membership queries) is well known to imply learnability in the distribution-independent PAC model (correspondingly augmented with membership queries). More precisely, if a class $\mathcal{C}$ is learnable in time $T(n)$ using $Q(n)$ queries in the exact model, then by a standard argument[2] $\mathcal{C}$ is learnable to confidence $1-\delta$ and accuracy $1-\varepsilon$ in the PAC model in time $T_{\mathrm{PAC}} = O(\frac{T(n)}{\varepsilon}\ln(\frac{T(n)}{\delta}))$ using $O(\frac{Q(n)}{\varepsilon}\ln(\frac{Q(n)}{\delta}))$ queries.

Non-trivial savings. It is easy to see that any class $\mathcal{C}$ can be learned in time $\mathrm{poly}(n) \cdot 2^n$ in our model via a simple memorization-based approach; our goal in this work will be to come up with algorithms whose running time is $2^{n-s(n)}$ where the *savings* $s(n)$ is as large as possible. We say that any savings function $s(n) = \omega(\log n)$ is *non-trivial.* We observe that the conversion from exact learning to distribution-independent PAC learning described above preserves learnability with non-trivial savings: learnability with non-trivial savings in our exact model implies learnability to any $1/\mathrm{poly}(n)$ accuracy and confidence in the PAC model with non-trivial savings (since if $T(n) = 2^{n-\omega(\log n)}$ and $\varepsilon, \delta = 1/\mathrm{poly}(n)$ then $T_{\mathrm{PAC}} = 2^{n-\omega(\log n)}$).

Deterministic learning implies compression. As mentioned in the introduction, now that we have a precise definition of our learning model it is easy to verify that any class of functions that admits a non-trivial deterministic learning algorithm admits a compression algorithm. To see this, observe that our learning algorithms (i) are not given the full truth table of f as input, and (ii) must run in time strictly less than 2^n (as opposed to $2^{O(n)}$ for compression), while (iii) a learning algorithm in our framework must (like a compression algorithm) ultimately construct a circuit computing f that has size less than $2^n/n$. We may summarize this discussion in the following observation:

[2] See e.g. Section 2.4 of [Ang88], replacing the expression "$i \ln 2$" by "$\ln(2i^2)$".

Observation 1. *Let $\mathcal{C}$ be a class of n-variable Boolean functions that has a deterministic exact learning algorithm using membership and equivalence queries with savings $s(n) = \omega(\log n)$ (i.e. in time $2^{n-s(n)}$). Then there is a deterministic algorithm that compresses $\mathcal{C}$ to circuits of size $2^{n-s(n)}$.*

As an application of this observation, consider the class of size-S read-once branching programs (ROBPs) over $x_1, \ldots, x_n$. Since every such size-S ROBP is a deterministic finite automaton with S nodes over the binary alphabet $\{0, 1\}$ accepting only n-bit strings, Angluin's deterministic exact learning algorithm [Ang87] (which uses membership and equivalence queries) can learn any such ROBP in time $O(S^2)$. By Observation 1, this implies an algorithm that compresses $2^{n/2-s(n)/2}$-size ROBPs to circuits of size $O(2^{n-s(n)})$. This recovers a compression result for this class that was previously obtained by Chen et al. in the paper [CKK$^+$15] that initiated the study of compression algorithms (see their Theorem 3.8).

2.1 A first simple approach based on uniform-distribution learning

In contrast with the state of affairs for distribution-independent PAC learning, a more significant body of results is known for *uniform-distribution* PAC learning (as we will see later in this section). In this section we describe a simple approach by which some uniform-distribution PAC learning algorithms — roughly speaking, those which have a good dependence on the accuracy parameter ε — can easily be translated into non-trivial exact learning algorithms.

The simple approach, which was already suggested in [CIKK16] in the context of compression, is as follows. Using membership queries, we may simulate uniform random examples $(\boldsymbol{x}, f(\boldsymbol{x}))$ and run the uniform-distribution learning algorithm to obtain an ε-accurate hypothesis h in time $T(n, 1/\varepsilon, \log(1/\delta))$. Then we use at most $\varepsilon \cdot 2^n$ equivalence queries to identify and correct all of the (at most $\varepsilon \cdot 2^n$) many points on which h is incorrect. Since updating the hypothesis after each equivalence query can clearly be done in time $\mathrm{poly}(n)$ we thus obtain the following:

Claim 2.1. *Let $\mathcal{C}$ be a class of n-variable Boolean circuits such that there is a uniform-distribution PAC learning algorithm (which may possibly use membership queries) running in time $T(n, 1/\varepsilon, \log(1/\delta))$, which with probability $1-\delta$ outputs an ε-accurate hypothesis. Then there is a randomized exact learning algorithm for $\mathcal{C}$ which uses membership and equivalence queries and runs in time*

$$\mathrm{poly}(n) \cdot \min_{\varepsilon > 0} \left\{ T(n, 1/\varepsilon, \log(1/\delta)) + \varepsilon \cdot 2^n \right\}. \tag{1}$$

First application of Claim 2.1: Learning AC^0 circuits. The seminal work of Linial, Mansour, and Nisan [LMN93] established Fourier concentration bounds for size-M depth-d circuits, and showed how these bounds straightforwardly yield uniform-distribution learning algorithms. An essentially optimal strengthening of the Fourier concentration bound of [LMN93] was recently obtained by Tal [Tal15], who showed that there exists a universal constant $c > 0$ such that every size-M depth-d circuit C satisfies $\sum_{|S| \geq c \log^{d-1}(M) \log(1/\varepsilon)} \widehat{C}(S)^2 \leq \varepsilon$. Via the connection between Fourier concentration and uniform-distribution learning established by [LMN93], this implies that the class of size-M depth-d circuits can be learned to accuracy ε by a randomized algorithm in time $\mathrm{poly}\left(\binom{n}{\leq c \log^{d-1}(M) \log(1/\varepsilon)}\right) \cdot \log(1/\delta)$. Consequently, by Claim 2.1, taking $\varepsilon = 2^{-\Theta(n/(\log M)^{d-1})}$ we get a randomized exact learning algorithm which uses membership and equivalence queries which runs in time $\mathrm{poly}(n) \cdot 2^{n-\Omega(n/(\log M)^{d-1})}$. We note that this matches the circuit size given by the compression theorem of [CKK$^+$15] for such circuits.

Learning functions of k LTFs. Gopalan et al. [GKM12] have given a randomized uniform-distribution membership-query algorithm that learns any function of k LTFs over $\{0,1\}^n$ in time $O((nk/\varepsilon)^{k+1})$. Choosing $\varepsilon = 2^{-\frac{n}{k+2}}$ in Claim 2.1, we get a randomized exact learning algorithm which uses membership and equivalence queries and runs in time $\mathrm{poly}(n) \cdot 2^{\frac{k+1}{k+2}n} = \mathrm{poly}(n) \cdot 2^{n-\frac{n}{k+2}}$, thus achieving a non-trivial savings for any $k = o(n/\log n)$, and a linear savings for any constant k.

Learning $\mathsf{AC}^0[p]$ circuits. A recent exciting result of [CIKK16] gives a randomized uniform-distribution membership-query algorithm for learning the class of n-variable size-M depth-d $\mathsf{AC}^0[p]$ circuits to accuracy ε in time $2^{(\log(Mn/\varepsilon))^{O(d)}}$. By Claim 2.1, taking $\varepsilon = M \cdot 2^{-n^{\Theta(1/d)}}$, we get a randomized exact learning algorithm which uses membership and equivalence queries and runs in time $2^{n-n^{\Omega(1/d)}}$ for all circuits of size $M \leq 2^{n^{c/d}}$ for some absolute constant $c > 0$.

3 Beyond uniform-distribution learnability: Learning via random restrictions

As noted briefly in Section 2.1, in order for Claim 2.1 to give a non-trivial savings for exact learning the running time $T(n, 1/\varepsilon, \log(1/\delta))$ of the uniform-distribution learning algorithm must not depend too badly on $1/\varepsilon$. This requirement limits the applicability of Claim 2.1; to see a concrete example of this, let us consider the class of all $\mathrm{poly}(n)$-size, depth $d = O(1)$ LTF-of-AC^0 circuits (so such a circuit has an arbitrary linear threshold function as the output gate with $\mathrm{poly}(n)$ many $\mathrm{poly}(n)$-size depth-$(d-1)$ AC^0 circuits feeding into the threshold gate). Uniform-distribution learning results [GS10, Kan13] are known for this class, based on Fourier concentration which is established via known upper bounds on the average sensitivity of low-degree polynomial threshold functions. As discussed in [GS10], the best running time that can be achieved for learning via this approach is $n^{(\log n)^{O(d)}/\varepsilon^2}$, which would follow from a conjecture of Gotsman and Linial, known to be best possible, upper bounding the average sensitivity of low-degree polynomial threshold functions. (The current state of the art learning algorithms, based on Kane's upper bound [Kan13] on the average sensitivity of low-degree polynomial threshold functions which nearly matches the Gotsman-Linial conjecture, have a slightly worse running time.) As a result of this poor dependence on ε, the value of (1) is $\Omega(2^n/\sqrt{n})$, so no non-trivial savings is achieved. We note that even for the $d = 1$ case of a single linear threshold gate as the function to be learned, the best possible running time of a learning algorithm based on Fourier concentration is $n^{\Omega(1/\varepsilon^2)}$ (see Theorem 23 of [KOS04]).

An approach based on random restrictions. In this section we show that by taking a more direct approach than Claim 2.1, it is possible to achieve a non-trivial savings for LTF-of-AC^0 circuits, and to improve on the results achievable via Claim 2.1 for the class of Parity-of-AC^0 circuits, which is covered by the final learning result in Section 2.1. An additional advantage of this random restriction based approach is that (unlike the uniform distribution approach based on Claim 2.1) the resulting exact learning algorithms do not require membership queries, only equivalence queries.

This approach is based on the method of random restrictions; it is reminiscent of the simple "proof of concept" from the Introduction (though we will ultimately instantiate it with a more sophisticated switching lemma than Håstad's original switching lemma [Hås86]). Roughly speaking the approach works as follows: Let $\mathcal{R}_p$ denote the distribution over n-variable random restrictions (i.e. over $\{0,1,*\}^n$) that independently sets each coordinate to 0, 1, or $*$ with probabilities $\frac{1-p}{2}$, $\frac{1-p}{2}$ and p respectively. Let $\mathcal{C}$ be the class of functions that we would like to learn, and let $\mathcal{C}'$ be some other class of functions (which should be thought of as "simpler" than the functions in $\mathcal{C}$). If we have

(i) a switching lemma type statement establishing that for $\boldsymbol{\rho} \leftarrow \mathcal{R}$, any $f \in \mathcal{C}$ with high probability collapses under $\boldsymbol{\rho}$ to a function in $\mathcal{C}'$, and (ii) an exact algorithm A that can learn functions in $\mathcal{C}'$ in time significantly faster than brute force, then we can achieve nontrivial savings by (a) drawing a random restriction $\boldsymbol{\rho} \leftarrow \mathcal{R}$, (b) partitioning $\{0,1\}^n$ into translates of the $|\boldsymbol{\rho}^{-1}(*)|$-dimensional subcube corresponding to the unfixed variables of $\boldsymbol{\rho}$, and (c) running the algorithm A on each of the $2^{n-|\boldsymbol{\rho}^{-1}(*)|}$ many such subcubes. By (i), for most subcubes we will achieve a significant savings over a brute-force $2^{|\boldsymbol{\rho}^{-1}(*)|}$ running time for that subcube; even "paying full fare" for the (few) remaining bad subcubes, this results in an overall algorithm with non-trivial savings.

We make this discussion formal in the following lemma:

Lemma 3.1. *Let $\mathcal{C}$ and $\mathcal{C}'$ be two classes of Boolean functions, where $\mathcal{C} = \bigcup_{n \geq 1} \mathcal{C}_n$ and functions in $\mathcal{C}_n$ are n-variable Boolean functions and likewise for $\mathcal{C}'$. Suppose that $\mathcal{C}$ and $\mathcal{C}'$ are such that for some value $\frac{8}{n} \leq p < 1$, we have*

1. *(switching lemma from $\mathcal{C}$ to $\mathcal{C}'$) For every function $f \in \mathcal{C}_n$,*

$$\Pr_{\boldsymbol{\rho} \leftarrow \mathcal{R}_p}\left[f \upharpoonright \boldsymbol{\rho} \text{ does not belong to } \mathcal{C}'_{|\boldsymbol{\rho}^{-1}(*)|}\right] \leq \alpha(n); \tag{2}$$

2. *(efficient learnability of $\mathcal{C}'$) There is an exact learning algorithm A for $\mathcal{C}'$ that uses equivalence queries only and runs in time $T(\ell) = 2^{o(\ell)}$ when it is run on a function in $\mathcal{C}'_\ell$.*

Then there is a randomized exact learning algorithm for $\mathcal{C}_n$ which uses equivalence queries only, outputs a correct hypothesis with probability $1-\delta$, and runs in time

$$\text{poly}(n) \cdot \left(2^{n-pn/2} \cdot T(pn/2) + \alpha(n) \cdot 2^n\right) \cdot \log(1/\delta). \tag{3}$$

Proof. The randomized exact learning algorithm executes a sequence of at most $O(\log(1/\delta))$ independent stages, halting the first time a stage succeeds. We will show below that each stage succeeds in producing an exactly correct hypothesis with probability at least 0.35, and runs in time $\text{poly}(n) \cdot \left(2^{n-pn/2} \cdot T(pn/2) + \alpha(n) \cdot 2^n\right)$; the lemma follows easily from this.

Each stage consists of two substages and is structured as follows. In the first substage, the exact learning algorithm draws a random restriction $\boldsymbol{\rho} \leftarrow \mathcal{R}_p$. By a standard multiplicative Chernoff bound (using $p \geq \frac{8}{n}$) we have that $|\boldsymbol{\rho}^{-1}(*)| < pn/2$ with probability at most $\exp(-pn/8) < e^{-1}$; if $|\boldsymbol{\rho}^{-1}(*)| < pn/2$ then this stage ends in failure, otherwise the algorithm continues to the second substage (described in the next paragraph). Let $C_{\boldsymbol{\rho}}$ be the subcube of $\{0,1\}^n$ (of dimension $|\boldsymbol{\rho}^{-1}(*)|$ and containing $2^{|\boldsymbol{\rho}^{-1}(*)|}$ many points) corresponding to the live variables of $\boldsymbol{\rho}$, and let $C_{\boldsymbol{\rho},\text{translates}}$ be the set of all $2^{|n-\boldsymbol{\rho}^{-1}(*)|}$ many disjoint translates of $C_{\boldsymbol{\rho}}$ which together cover $\{0,1\}^n$. We say that a translate $C_{\boldsymbol{\rho}} + z \in C_{\boldsymbol{\rho},\text{translates}}$ (viewing addition as being over $\mathbb{F}_2$) of $C_{\boldsymbol{\rho}}$ is *bad* if the translated restriction $\boldsymbol{\rho}+z$ (whose $*$'s are in the exact same locations as those of $\boldsymbol{\rho}$) corresponding to $C_{\boldsymbol{\rho}}+z$ is such that $f \upharpoonright (\boldsymbol{\rho}+z)$ does not belong to $\mathcal{C}'$, and we say that $\boldsymbol{\rho}$ is bad if more than a $4\alpha(n)$ fraction of the $2^{|n-\boldsymbol{\rho}^{-1}(*)|}$ translates of $C_{\boldsymbol{\rho}}$ are bad. By Markov's inequality applied to (2), we have that $\boldsymbol{\rho} \leftarrow \mathcal{R}_p$ is bad with probability at most $1/4$. We thus have that with overall probability at least $1 - 1/4 - e^{-1} > 0.35$ over the draw of $\boldsymbol{\rho} \leftarrow \mathcal{R}_p$, the stage proceeds to the second substage with a restriction $\boldsymbol{\rho}$ that is not bad (and that satisfies $|\boldsymbol{\rho}^{-1}(*)| \geq pn/2$).

In the second substage, the exact learning algorithm then runs $2^{|n-\boldsymbol{\rho}^{-1}(*)|}$ copies of algorithm A in parallel, each one to learn the ($\ell = |\boldsymbol{\rho}^{-1}(*)|$)-variable function which is $f \upharpoonright (C_{\boldsymbol{\rho}} + z)$ for one of the translates of $C_{\boldsymbol{\rho}}$. This can be done using equivalence queries only: the overall hypothesis at each time step is obtained from the $2^{|n-\boldsymbol{\rho}^{-1}(*)|}$ many hypotheses (one for each subcube) in the obvious way. Each counterexample received allows one of the $2^{|n-\boldsymbol{\rho}^{-1}(*)|}$ copies of algorithm A (the one running over the subcube that received the counterexample) to update its hypothesis. Let $M(\ell) \leq T(\ell)$ be the maximum number of counterexamples that A can ever receive when it is run on a function in $\mathcal{C}'_\ell$. Within each subcube, if the copy of A running in that subcube receives more than $M(\ell)$ counterexamples, then since that subcube must be bad, the overall exact learning algorithm switches from running A on that subcube to running a naive equivalence-query learning algorithm that simply builds a truth table (and takes time at most $\mathrm{poly}(n) \cdot 2^{|\boldsymbol{\rho}^{-1}(*)|}$, the number of points in the subcube).

The second substage carries out this process until either

(i) no counterexample is provided (meaning that all $2^{|n-\boldsymbol{\rho}^{-1}(*)|}$ copies of the algorithm have obtained an exactly correct hypothesis, and thus the overall combined hypothesis is exactly correct and the stage succeeds), or

(ii) more than $4\alpha(n)2^{n-\ell}$ copies of the algorithm have each received more than $M(\ell)$ counterexamples; since this can only happen if $\boldsymbol{\rho}$ is bad, in this case the stage halts and ends in failure.

We observe that case (i) must occur if $\boldsymbol{\rho}$ is not bad, and hence case (i) occurs and the stage succeeds with overall probability at least 0.35. In either case the running time for the stage is at most

$$\begin{aligned}\mathrm{poly}(n) \cdot \left(2^{|n-\ell|} \cdot T(\ell) + 4\alpha(n)2^{n-\ell} \cdot (T(\ell) + 2^\ell)\right) &< \mathrm{poly}(n) \cdot \left(2^{|n-\ell|} \cdot T(\ell) + 4\alpha(n)2^{n-\ell} \cdot (2 \cdot 2^\ell)\right) \\ &< \mathrm{poly}(n) \cdot \left(2^{n-pn/2} \cdot T(pn/2) + \alpha(n) \cdot 2^n\right)\end{aligned}$$

time steps, where the first summand on the LHS upper bounds the total running time of all the learning algorithms that are running over non-bad subcubes, and the second summand bounds the total running time of all the learning algorithms that are running over the (at most $4\alpha(n)2^{n-\ell}$) many bad subcubes. As discussed at the beginning of the proof, this establishes the lemma. □

3.1 An application of Lemma 3.1: learning LTF-of-AC^0 and Parity-of-AC^0

In this subsection we use Lemma 3.1 to obtain non-trivial exact learning algorithms for LTF-of-AC^0 and Parity-of-AC^0 circuits. As discussed at the start of Section 3, it does not seem possible to obtain a non-trivial exact learning algorithm for LTF-of-AC^0 using known uniform-distribution learning results. The learning algorithm for Parity-of-AC^0 that we give in this subsection achieves significantly better savings than the algorithm from Section 2.1, and moreover does not require membership queries.

In order to apply Lemma 3.1 we need a suitable switching lemma from $\mathcal{C}$ to $\mathcal{C}'$ and a learning algorithm for $\mathcal{C}'$. Looking ahead, for LTF-of-AC^0 the class $\mathcal{C}'$ will be the class of low-degree polynomial threshold functions, and for Parity-of-AC^0 it will be the class of low-degree $\mathbb{F}_2$ polynomials. We can use the same switching lemma for both results; to describe the switching lemma we need, we recall some terminology from [Hås14]. Let $\mathcal{G}$ be a family of Boolean functions. A decision tree T is said to be a *common ℓ-partial decision tree for $\mathcal{G}$* if every $g \in \mathcal{G}$ can be expressed as T with depth-ℓ decision trees hanging off its leaves. (Equivalently, for every $g \in \mathscr{G}$ and root-to-leaf path π in T, we have that $g \upharpoonright \pi$ is computed by a depth-ℓ decision tree.)

If g is a Boolean function and $\mathcal{C}$ is a class of circuits, we say that g is *computed by a* $(t, \mathcal{C})$-*decision tree* if g is computed by a decision tree of depth t (with single Boolean variables x_i at internal nodes as usual) in which each leaf is labeled by a function from $\mathcal{C}$. We write DT_k to denote the class of depth-k decision trees.

We use a recent powerful switching lemma for multiple DNFs due to Håstad [Hås14] (a similar switching lemma was independently obtained by [IMP12]):[3]

Theorem 2 ([Hås14] multi-switching lemma)**.** *Let $\mathscr{F} = \{F_1, \ldots, F_S\}$ be a collection of depth-2 circuits with bottom fan-in w. Then for any $t \geq 1$,*

$$\Pr_{\boldsymbol{\rho} \leftarrow \mathcal{R}_p} \left[\mathscr{F} \upharpoonright \boldsymbol{\rho} \textit{ does not have a common } (\log S)\textit{-partial DT of depth} \leq t \right] \leq S(24pw)^t.$$

We will use the following simple corollary for AC^0 circuits augmented with some gate G on top as our "switching lemma from $\mathcal{C}$ to $\mathcal{C}'$" in Lemma 3.1 (see Appendix A for the proof):

Corollary 3.2. *Let G be any Boolean function, and let F be a size-S depth-$(d+1)$ $G \circ \mathsf{AC}^0$ circuit (where we view G as a single gate at the output of the circuit). Then for $p = \frac{1}{48}(48 \log S)^{-(d-1)}$ and any $t \geq 1$,*

$$\Pr_{\boldsymbol{\rho} \leftarrow \mathcal{R}_p} \left[F \upharpoonright \boldsymbol{\rho} \textit{ is not computed by a } (2^d t, G \circ \mathsf{DT}_{\log S})\textit{-decision tree} \right] \leq d \cdot S \cdot 2^{-t}.$$

For the exact learning results we need, we recall the following well-known facts (the first follows easily from [MT94], see e.g. [HS07], and the second follows easily from Gaussian elimination):

Fact 3.3.

1. *There is an exact learning algorithm (using equivalence queries only) that learns degree-d polynomial threshold functions (PTF) over ℓ Boolean variables in time* $\mathrm{poly}(\binom{\ell}{\leq d})$.
2. *The same running time holds for exact learning degree-d $\mathbb{F}_2$ polynomials (again using equivalence queries only).*

All the pieces are now place for our exact learning algorithms for LTF-of-AC^0 and parity-of-AC^0:

Theorem 3.

1. *There is an exact learning algorithm (using equivalence queries only) that learns the class of size-S depth-$(d+1)$ LTF-of-AC^0 circuits over $\{0,1\}^n$ in time $S \cdot 2^{n - n/O(\log S)^{d-1}}$.*
2. *The same running time holds for exact learning size-S depth-$(d+1)$ Parity-of-AC^0 (again using equivalence queries only).*

Proof. We prove part (1) first (part (2) is almost identical). Let $\mathcal{C}'$ be the class of all PTFs of degree $2^d t + \log S$ (where t will be chosen later). We observe that any $\mathsf{LTF} \circ \mathsf{DT}_{\log S}$ circuit computes a PTF of degree $\log S$, and moreover that any $(2^d t, \mathsf{LTF} \circ \mathsf{DT}_{\log S})$-decision tree computes a PTF of degree $2^d t + \log S$. Applying part (1) of Fact 3.3, Corollary 3.2, and Lemma 3.1 with p as in Corollary 3.2 and choosing $t = 0.1pn/(2 \cdot 2^d)$, we get the desired learning algorithm. Part (2) follows similarly but now using the observation that any $(2^d t, \mathsf{PAR} \circ \mathsf{DT}_{\log S})$-decision tree computes an $\mathbb{F}_2$ polynomial of degree $2^d t + \log S$. □

[3] We note that this multi-switching lemma is the key technical ingredient in [Tal15]'s sharpening of the [LMN93] Fourier concentration result which gave our AC^0 learning result in Section 2.1.

4 Learning with non-trivial savings via Nečiporuk's method

In this section we present our third technique for learning with non-trivial savings. This technique is based on Nečiporuk's method, which gives a lower bound on the complexity of a function f (in various computational models such as formula size, branching program size, etc.) in terms of the number of subfunctions of f. In more detail, Nečiporuk's theorem essentially says that if the variables of f can be partitioned into disjoint subsets $S_1, S_2, \dots$ such that the product, across all i, of (the number of distinct subfunctions than can arise when all variables in $[n] \setminus S_i$ are fixed to constants in all $2^{n-|S_i|}$ possible ways) is large, then f must have high complexity. Our technique is based on a contrapositive view: if f is a function of "not too high" complexity, then in any partition of the variables into "large" equal-size subsets, $S_1, S_2, \dots$ there must be some S_i over which f has "not too many" distinct subfunctions — in particular, far fewer than $2^{n-|S_i|}$, the number of distinct subcubes corresponding to the restrictions that fix all variables in $[n] \setminus S_i$. We show that this structure (having "few" subfunctions over a "large" subset of variables) can be exploited to learn f with non-trivial savings.

Warmup: Compression. In Section 4.1 we first develop this idea for the easier problem of compression rather than learning. We obtain a new and simpler algorithm and analysis recovering the deterministic compression results of [CKK+15] for $n^{1.99}$-size full-basis binary formulas and $n^{1.99}$-size branching programs. ([CKK+15] had to develop new high-probability analyses of shrinkage under random restrictions using novel martingale arguments and combine these analyses with a generalization of the greedy set-cover heuristic, whereas we only use the statement of Nečiporuk's theorem in a black-box way together with short and elementary arguments.) Thanks to the generality of Nečiporuk's method, our algorithm and analysis also yields new deterministic compression results for switching networks of size $n^{1.99}$, switching-and-rectifier networks of size $n^{1.49}$, non-deterministic branching programs of size $n^{1.49}$, and span programs of size $n^{1.49}$.

Learning. Progressing from compression to learning, next in Section 4.2 we describe how pre-processing can be used to create a data structure which enables a highly efficient implementation of the classic "halving algorithm" from learning theory. While a naive implementation of the halving algorithm to learn an unknown function from a class of N functions over an M-element domain takes time $O(NM)$, we show that by first carrying out a pre-processing step taking time $M^{O(\log N)}$ it is possible to run the halving algorithm in time only $\mathrm{poly}(\log N, \log M)$, an *exponential* savings. This means that if we need to run the halving algorithm many times, by first running the pre-processing step (which needs to be done only once) we can carry out these many runs of the halving algorithm in a highly efficient *amortized* way. Intuitively, running the halving algorithm many times is precisely what we need to do in our Nečiporuk-based learning approach: if S is the "large" subset of variables such that f has "not too many" subfunctions over S, then we will run the halving algorithm $2^{n-|S|}$ times, once for each possible subcube keeping the variables in S free, to learn the corresponding $2^{n-|S|}$ different restrictions of f.

Finally, in Section 4.3 we describe and analyze our general learning algorithm based on Nečiporuk's method. The algorithm has three stages: in the first stage, membership queries are used to randomly sample subcubes corresponding to S, which are exhaustively queried to learn the subfunctions they contain. In this way the first stage constructs a set A containing all "important" subfunctions (ones that occur in "many" subcubes); crucially, thanks to the Nečiporuk argument, the set is not too large (since there are "few" distinct subfunctions in total, important or otherwise). The second stage performs the above-described pre-processing on the set A of subfunctions, and the third stage runs the halving algorithm over all $2^{n-|S|}$ subcubes corresponding to S in the efficient amortized way described above. This results in a hypothesis which is exactly correct on every subcube containing an "important" subfunction; by definition there are only "few" subcubes that contain non-important subfunctions, and the hypothesis can be patched up on those subcubes at relatively small cost.

4.1 Compression based on having few subfunctions

Given $f : \{0,1\}^n \to \{0,1\}$ and $S \subseteq [n]$, let $\mathcal{R}_S$ denote the set of all $2^{n-|S|}$ restrictions that leave precisely the variables in S free and assign either 0 or 1 to each element of $[n] \setminus S$ in all possible ways. Let $\mathrm{Num}(f, S)$ denote the number of distinct functions from $\{0,1\}^S$ to $\{0,1\}$ that occur in $\{f \upharpoonright \rho\}_{\rho \in \mathcal{R}_S}$ (i.e. that occur as subfunctions of f).

Lemma 4.1 (Compression based on few subfunctions)**.** *Fix any partition $S_1, S_2, \ldots, S_{n^{1-\delta}}$ of $[n]$ into equal-size subsets S_i of size n^δ each, where $\delta > 0$. Let $\mathcal{C}$ be a class of n-variable functions such that for each $f \in \mathcal{C}$ there is a set S_i such that $\mathrm{Num}(f, S_i) \leq 2^{n^\beta}$, where $\beta < 1$. Then there is a compression algorithm for $\mathcal{C}$ running in time $2^{O(n)}$ with savings n^δ (i.e. given as input the truth table of any $f \in \mathcal{C}$, the algorithm outputs a circuit computing f of size $\mathrm{poly}(n) \cdot 2^{n-n^\delta}$).*

Proof. Fix $f \in \mathcal{C}$, and say that any $i \in [n^{1-\delta}]$ for which $\mathrm{Num}(f, S_i) \leq 2^{n^\beta}$ is *good*. The compression algorithm works as follows:

1. For $i = 1, 2, \ldots$ check whether i is good by building a sorted list of all the distinct subfunctions occuring in $\{f \upharpoonright \rho\}_{\rho \in \mathcal{R}_{S_i}}$. This can be done in time $2^{O(n)}$. The hypothesis of the lemma ensures that some i is good; in the following steps for notational simplicity we suppose that $i = 1$ is good. So at this point the algorithm has a sorted list L containing at most 2^{n^β} truth tables (each being an 2^{n^δ}-bit string), and for every $\rho \in \mathcal{R}_{S_1}$ the truth table of $f \upharpoonright \rho$ is in the list.

2. Iterate across all $\rho \in \mathcal{R}_{S_1}$ to construct a function $\Phi : \mathcal{R}_{S_1} \to [2^{n^\beta}]$ such that for each $\rho \in \mathcal{R}_{S_1}$ the value of $\Phi(\rho)$ is the index j such that the truth table of $f \upharpoonright \rho$ is the j-th element of the list L. (Note that the description length of the function Φ is $|\mathcal{R}_{S_1}| \cdot \log(2^{n^\beta}) = \mathrm{poly}(n) \cdot 2^{n-n^\delta}$.) This can be done in time $2^{O(n)}$.

3. Finally, the compression algorithm outputs a circuit which works as follows: given input $x \in \{0,1\}^n$, let ρ_x be the element of $\mathcal{R}_{S_1}$ that is consistent with x (fixing the variables in $[n] \setminus S_1$ according to x). The circuit outputs the appropriate output bit (corresponding to the bit-string x restricted to the coordinates in S_1) from the $\Phi(\rho_x)$-th truth table of L. This circuit computes f and is of size $\mathrm{poly}(n) \cdot (|\mathcal{R}_{S_1}| + 2^{n^\delta} \cdot |L|) = \mathrm{poly}(n) \cdot (2^{n-n^\delta} + 2^{n^\delta} \cdot 2^{n^\beta}) \leq \mathrm{poly}(n) \cdot 2^{n-n^\delta}$, and this step can be done in time $2^{O(n)}$.

□

Given Lemma 4.1, a direct invocation of the lower bounds provided by Nečiporuk's method for various computational models gives the following corollary, providing a wide range of deterministic compression results. We refer the reader to [Juk12] for detailed definitions of all the computational models mentioned in Corollary 4.2.

Corollary 4.2. *Boolean n-variable functions computable by computational model $\mathcal{A}$ of size S are compressible in time $2^{O(n)}$ to circuits of size at most 2^{n-n^ε} for a fixed $\varepsilon > 0$, where*

1. *$\mathcal{A}$ = full-basis binary formulas, $S = n^{1.99}$;*

2. *$\mathcal{A}$ = branching programs, $S = n^{1.99}$;*

3. *$\mathcal{A}$ = switching networks, $S = n^{1.99}$;*

4. *$\mathcal{A}$ = switching-and-rectifier networks, $S = n^{1.49}$;*

5. *$\mathcal{A}$ = non-deterministic branching programs, $S = n^{1.49}$;*

6. *$\mathcal{A}$ = span programs, $S = n^{1.49}$.*

Proof. We first give the argument for (1), full-basis binary formulas and $S = n^{1.99}$. We take $\delta = 0.004$ in Lemma 4.1, so $1 - \delta = 0.996$. Let f be any n-variable function with a full-basis binary formula of size at most $n^{1.99}$. We recall that Nečiporuk's lower bound for full-basis formula size of f (denoted $L(f)$) implies that

$$\frac{1}{4}\sum_{i=1}^{n^{0.996}} \log(\mathrm{Num}(f, S_i)) \le L(f) \le n^{1.99},$$

so there is some $i \in [n^{0.996}]$ such that $\log(\mathrm{Num}(f, S_i)) \le 4n^{0.994} < n^{0.995}$, so we have $\beta = 0.995$ and obtain the claimed compression result from Lemma 4.1.

The arguments for (2) and (3) follow similarly, using

$$\tau \cdot \sum_{i=1}^{n^{0.996}} \frac{\log(\mathrm{Num}(f, S_i))}{\log\log(\mathrm{Num}(f, S_i))} \le S(f) \le BP(f)$$

(see e.g. Theorem 15.1 of [Juk12]) for some absolute constant τ, where $BP(f)$ denotes the branching program size of f and $S(f)$ denotes the switching network size of f.

(4) and (5) also follow similarly, recalling that Nečiporuk's method gives

$$\frac{1}{4} \cdot \sum_{i=1}^{n^{0.996}} \sqrt{\log(\mathrm{Num}(f, S_i))} \le RS(f) \le NBP(f)$$

(see [Pud87]), where $RS(f)$ denotes the rectifier-and-switching network size of f and $NBP(f)$ denotes the non-deterministic branching program size of f. Finally, for (6) we recall that

$$\frac{1}{2} \cdot \sum_{i=1}^{n^{0.996}} \sqrt{\log(\mathrm{Num}(f, S_i))} \le SPAN(f)$$

(see Theorem 1 of [KW93]), where $SPAN(f)$ denotes the span program size of f. □

4.2 More efficient implementation of the halving algorithm via pre-processing

We begin by recalling the halving algorithm [BF72, Ang88, Lit88] and its running time when it is executed over a domain X of M points to learn an unknown function that is promised to belong to a set $\mathcal{C}$ of at most N (known) functions, where each function in $\mathcal{C}$ may be viewed simply as a truth table of length M. (In the context of the previous subsection the domain size M corresponds to

2^{n^δ}, the number of points in each subcube, and N corresponds to 2^{n^β}, the number of subfunctions.) Recall that after a set A of labeled examples has been received, the *version space* of A is the subset of functions in $\mathcal{C}$ that are consistent with A. At each stage in the halving algorithm's execution, its current hypothesis is the majority vote over the version space of the labeled examples received thus far. Before any counterexamples have been received, initially the version space is all of $\mathcal{C}$; thus the first thing that the halving algorithm does is spend NM time to (a) read the entire bit-matrix corresponding to the current version space $\mathcal{C}$ (think of this matrix as having N columns, which are the truth tables of the functions in the class, and M rows corresponding to the points in the domain) and (b) for each row compute and record the majority vote over the elements in this row (which is the initial hypothesis). The halving algorithm gets a counterexample, and then updates its version space; since its hypothesis was the majority vote of all functions in the previous version space, at least half of the columns (the functions that are inconsistent with the counterexample) are erased, and the size of the version space goes down by at least $1/2$. To form the next hypothesis the halving algorithm spends at most $(N/2)M$ time to read the matrix corresponding to the current version space and for each row compute and record the majority vote over the surviving elements in this row. This proceeds for at most $\log N$ steps, after which the version space must be of size one, and this sole surviving function must be the unknown target function. In the i-th stage the time required is $(N/2^i)M$ so the overall runtime is $O(NM)$. (Note that if the halving algorithm were performed separately and independently Z times (corresponding in our setting to the $Z = 2^{n-|S|}$ many distinct subcubes), the overall runtime would be $ZNM > 2^n$, which is too expensive for learning with non-trivial savings.)

The following lemma shows that the halving algorithm can be implemented *exponentially* more efficiently after an initial pre-processing stage. (Crucially, the pre-processing can be done only once even if the halving algorithm will be run many times; this leads to a tremendous amortized savings.) While simple, we are not aware of previous work giving an efficient amortized implementation of the halving algorithm.

Lemma 4.3. *Given a class $\mathcal{C}$ of N functions over an M-point domain X, there is a pre-processing procedure that (i) can be carried out in time $M^{O(\log N)}$ and (ii) creates a data structure* DS *such that given access to* DS*, the halving algorithm can be run to learn an unknown $f \in \mathcal{C}$ in time* $\mathrm{poly}(\log N, \log M)$*. (Consequently, given access to* DS*, the halving algorithm can be run Z times to learn a sequence $f_1, \ldots, f_Z$ of functions from $\mathcal{C}$ in total time $Z \cdot \mathrm{poly}(\log N, \log M)$.)*

Proof. We first describe the data structure DS and then establish (i) by explaining how it can be constructed in $M^{O(\log N)}$ time. We then establish (ii) by showing how DS can be used to run the halving algorithm efficiently.

The data structure DS. We say that a *size-i sample* is a set of precisely i labeled pairs $(x^1, y^1), \ldots, (x^i, y^i)$ where $x^1, \ldots, x^i$ may be any i distinct elements of X and $(y^1, \ldots, y^i)$ may be any bit-string in $\{0,1\}^i$. We write SAMP_i to denote the set of all size-i samples, so $|\mathsf{SAMP}_i| = \binom{M}{i} \cdot 2^i \le (2M)^i$. Observe that some elements of SAMP_i may not be consistent with any function $f \in \mathcal{C}$, while others may be consistent with many elements of $\mathcal{C}$.

The data structure DS consists of $\log N$ different "structures" which we refer to as $\mathcal{S}_0, \ldots, \mathcal{S}_{1+\log N}$. The i-th structure $\mathcal{S}_i$ is a set of (at most) $(2M)^i$ "i-substructures" $\{\mathcal{S}_{i,\mathsf{samp}}\}_{\mathsf{samp} \in \mathsf{SAMP}_i}$ which are indexed by elements of SAMP_i. Given an element $\mathsf{samp} = \{(x^1, y^1), \ldots, (x^i, y^i)\} \in \mathsf{SAMP}_i$, an i-substructure $\mathcal{S}_{i,\mathsf{samp}}$ has two parts: the "main part" $\mathsf{MAIN}(\mathcal{S}_{i,\mathsf{samp}})$ and one additional function (which we explain and give notation for below). The main part $\mathsf{MAIN}(\mathcal{S}_{i,\mathsf{samp}})$ is the subset of $\mathcal{C}$

that contains precisely those concepts in $\mathcal{C}$ that are consistent with samp, i.e. those functions $f \in \mathcal{C}$ that have $f(x^j) = y^j$ for all $j \in [i]$. The one additional function is $\mathsf{MAJ}(\mathsf{MAIN}(\mathcal{S}_{i,\mathsf{samp}}))$, the function that outputs, on any input $x \in X$, the majority vote over all the concepts in $\mathsf{MAIN}(\mathcal{S}_{i,\mathsf{samp}})$. This concludes the description of a generic i-substructure $\mathcal{S}_{i,\mathsf{samp}}$, and thus concludes the description of the i-th structure $\mathcal{S}_i$.

For example, the zeroth structure $\mathcal{S}_0$ consists of only one 0-substructure (since there is only one "empty sample" with no labeled pairs); call this 0-substructure $\mathcal{S}_{0,\mathsf{samp}_\emptyset}$. We have $\mathsf{MAIN}(\mathcal{S}_{0,\mathsf{samp}_\emptyset}) = \mathcal{C}$ (since every concept is consistent with the empty sample) and the one additional function is just the first hypothesis that the halving algorithm uses, the majority vote across $\mathcal{C}$.

Construction of DS. Suppose that $\mathcal{S}_{i-1}$, the $(i-1)$-st structure, has been constructed by the pre-processing procedure. The structure $\mathcal{S}_i$ is built from $\mathcal{S}_{i-1}$ as follows. $\mathcal{S}_i$ has exactly $2^i\binom{M}{i}$ substructures, one for each possible samp of size i. Consider such a $\mathsf{samp} = \{(x^1, y^1), \ldots, (x^i, y^i)\}$, and let samp' be $\{(x^1, y^1), \ldots, (x^{i-1}, y^{i-1})\}$, its length-$(i-1)$ prefix. Since $\mathsf{MAIN}(\mathcal{S}_{i-1,\mathsf{samp}'})$ has been constructed already as part of $\mathcal{S}_{i-1}$, given x^i it is easy to enumerate over the functions in $\mathsf{MAIN}(\mathcal{S}_{i-1,\mathsf{samp}'})$ and partition them into two groups; the functions f for which $f(x^i) = 0$ will go into $\mathsf{MAIN}(\mathcal{S}_{i-1,\mathsf{samp}'\cup\{(x^i,0)\}})$ and the ones for which $f(x^i) = 1$ will go into $\mathsf{MAIN}(\mathcal{S}_{i-1,\mathsf{samp}'\cup\{(x^i,1)\}})$. Finally once we have $\mathsf{MAIN}(\mathcal{S}_{i-1,\mathsf{samp}'\cup\{(x^i,y^i)\}})$ it is straightforward to read the corresponding matrix and construct the one additional function $\mathsf{MAJ}(\mathsf{MAIN}(\mathcal{S}_{i-1,\mathsf{samp}'\cup\{(x^i,y^i)\}}))$. It is not hard to see that the total size of $\mathcal{S}_i$, and the total time required to build it from $\mathcal{S}_{i-1}$, is at most $(2M)^i \cdot O(NM)$. Even when $i = \log N$ this is at most $O(N \cdot M^{\log N} \cdot NM) = M^{O(\log N)}$.

Using DS to run the halving algorithm efficiently. Now we describe how to emulate the halving algorithm in total time $\mathrm{poly}(\log N, \log M)$ given access to the structures $\mathcal{S}_0, \ldots, \mathcal{S}_{\log N}$. The initial hypothesis of the halving algorithm is the additional function $\mathsf{MAJ}(\mathsf{MAIN}(\mathcal{S}_{0,\mathsf{samp}_\emptyset}))$, i.e. the majority vote over all functions in $\mathcal{C}$, so the emulator for the halving algorithm need only "point to" this portion of DS to construct its initial hypothesis. On receiving a first labeled counterexample (x^1, y^1), the hypothesis of the halving algorithm is then precisely $\mathsf{MAJ}(\mathsf{MAIN}(\mathcal{S}_{1,\{(x^1,y^1)\}}))$; conveniently, this has been pre-computed as part of $\mathcal{S}_1$, so the emulator again only needs to point to this portion of DS to construct its second hypothesis. On receiving a second labeled counterexample (x^2, y^2), the emulator updates its hypothesis by pointing to $\mathsf{MAJ}(\mathsf{MAIN}(\mathcal{S}_{2,\{(x^1,y^1),(x^2,y^2)\}}))$ as its hypothesis, and so on. This goes on for at most $\log N$ stages, and it is clear that each stage requires time at most $\mathrm{poly}(\log N, \log M)$, giving (ii) as claimed. □

4.3 The general learning result based on Nečiporuk's method

With Lemma 4.3 in hand, we are ready to state and prove our general learning result based on Nečiporuk's method. As suggested earlier, the approach is to first do random sampling to identify the "important" subfunctions (ones which occur in many subcubes), then run the pre-processing procedure using these important subfunctions and the halving algorithm to efficiently learn over all subcubes containing important subfunctions, patching up the hypothesis on any subcubes that do not contain important subfunctions.

Lemma 4.4 (Learning based on few subfunctions)**.** *Fix any partition $S_1, S_2, \ldots, S_{n^{1-\delta}}$ of $[n]$ into equal-size subsets S_i of size n^δ each, where $\delta > 0$. Let $\mathcal{C}$ be a class of n-variable functions such that for each $f \in \mathcal{C}$ there is a set S_i such that* $\mathrm{Num}(f, S_i) \leq 2^{n^\beta}$, *where $\beta < 1$ and moreover $\beta + \delta < 1$. Then there is a randomized exact learning algorithm for $\mathcal{C}$ that uses membership and equivalence queries and achieves savings n^δ.*

Proof. We describe a randomized learning algorithm which works on S_1 and achieves the claimed runtime bound with high probability if $\mathrm{Num}(f, S_1) \leq 2^{n^\beta}$. If the algorithm runs for more than the claimed number of steps while working on S_1, it aborts and restarts, this time working on S_2, and so on. Hence in the following discussion we assume without loss of generality that $\mathrm{Num}(f, S_1) \leq 2^{n^\beta}$.

The learning algorithm works on S_1 in three stages as described below.

First stage: Identify important subfunctions. Recall that $\mathcal{R}_{S_1}$ is the set of all $2^{n-|S_1|} = 2^{n-n^\delta}$ restrictions that leave precisely the variables in S_1 free. Let $g_1, \dots, g_{\mathrm{Num}(f,S_1)}$ be the subfunctions that occur in $\{f \restriction \rho\}_{\rho \in \mathcal{R}_{S_1}}$. For $i \in [\mathrm{Num}(f, S_1)]$ let p_i denote the fraction of the 2^{n-n^δ} subcubes in $\mathcal{R}_{S_1}$ that have g_i as the subcube there. We say that a subfunction g_i is *important* if $p_i \geq \varepsilon/(10 \cdot \mathrm{Num}(f, S_1))$ (we will specify the value of ε later). Let $F' \subseteq \{g_1, \dots, g_{\mathrm{Num}(f,S_1)}\}$ be the set of all important subfunctions.

In the first stage we draw $A := 20n \cdot 2^{n^\beta}/\varepsilon$ independent uniform random elements of $\mathcal{R}_{S_1}$, and for each one we spend 2^{n^δ} many membership queries to exhaustively learn the associated truth table in time $\mathrm{poly}(n) \cdot 2^{n^\delta}$; let $F \subseteq \{g_1, \dots, g_{\mathrm{Num}(f,S_1)}\}$ be the set of all the subfunctions that are discovered in this way. For any given fixed important subfunction, the probability that it is not included in F is at most $(1 - \varepsilon/(10 \cdot \mathrm{Num}(f, S_1)))^A \leq (1 - \varepsilon/(10 \cdot 2^{n^\beta}))^A < 1/2^{2n}$, so a union bound over all (at most 2^{n^β}) important subfunctions gives that with probability at least $1 - 1/2^n$ the set F contains the set F' of important subfunctions (for the rest of the algorithm's analysis and execution we suppose that indeed F contains F'). We observe that by the definition of F', at most an $\varepsilon/10$ fraction of all $\rho \in \mathcal{R}_{S_1}$ are such that $f \restriction \rho$ do not belong to F', and hence at most an $\varepsilon/10$ fraction of all $\rho \in \mathcal{R}_{S_1}$ are such that $f \restriction \rho$ do not belong to F.

Note that the total running time for the first stage of the algorithm is $2^{n^\delta} \cdot \mathrm{poly}(n) \cdot 2^{n^\beta}/\varepsilon$.

Second stage: Do the pre-processing on F. Next, the algorithm performs the pre-processing described in the previous subsection on the $N = |F|$ functions in F, each of which is defined over the $M = 2^{n^\delta}$-size domain $\{0,1\}^{S_1}$. Since $N \leq A = 20n \cdot 2^{n^\beta}/\varepsilon$ we have that this takes time at most $M^{O(\log N)} < 2^{O(n^\beta \cdot n^\delta) + n^\beta \log(1/\varepsilon)}$.

Third stage: Run the halving algorithm in parallel over all $Z := 2^{n-n^\delta}$ subcubes in $\mathcal{R}_{S_1}$ using the data structure DS from Lemma 4.3. In the amortized analysis of the halving algorithm with pre-processing given in the previous subsection, we assumed that every execution of the halving algorithm was performed on a target function that actually belonged to the class $\mathcal{C}$. In our current setting there may be up to $(\varepsilon/10) \cdot 2^{n-n^\delta}$ many subcubes that contain functions that are not in F. When the halving algorithm is run over a subcube that contains a subfunction in F, it will correctly converge to the target subfunction over that subcube after at most $\log N \leq n^\beta$ many counterexamples, and the efficient implementation of the halving algorithm via pre-processing will work correctly over that subcube. This will happen on at least $1 - \varepsilon/10$ fraction of all 2^{n-n^δ} subcubes. For the remaining (at most $(\varepsilon/10) \cdot 2^{n-n^\delta}$) subcubes that have a subfunction not on our list, the halving algorithm may not work correctly. If, in a given subcube, the version space ever vanishes (note that if this happens it must happen after at most $\log N$ counterexamples from that subcube), or it has size greater than one after $\log N$ counterexamples, then it must be the case that that subcube's subfunction does not belong to F. In this case the algorithm uses 2^{n^δ} membership queries to "patch up" the hypothesis over this subcube (which can be done in time $\mathrm{poly}(n) \cdot 2^{n^\delta}$). This happens for at most $(\varepsilon/10) \cdot 2^{n-n^\delta}$ subcubes. Thus, the total running time of this stage will be at most

$$\begin{aligned}&(\text{time on subcubes with subfunctions in } F) + (\text{time on other subcubes})\\ &\le Z \cdot \mathrm{poly}(\log N, \log M)) + (\varepsilon/10) \cdot 2^{n-n^\delta} \cdot \mathrm{poly}(n) \cdot 2^{n^\delta}\\ &\le \mathrm{poly}(n) \cdot \left(2^{n-n^\delta} + (\varepsilon/10) \cdot 2^n\right).\end{aligned}$$

Hence the total running time for all stages can be upper bounded by

$$\overbrace{2^{n^\delta} \cdot \mathrm{poly}(n) \cdot 2^{n^\beta}/\varepsilon}^{\text{Stage 1}} + \overbrace{2^{O(n^\beta \cdot n^\delta) + n^\beta \log(1/\varepsilon)}}^{\text{Stage 2}} + \overbrace{\mathrm{poly}(n) \cdot \left(2^{n-n^\delta} + (\varepsilon/10) \cdot 2^n\right)}^{\text{Stage 3}}.$$

Recalling that by assumption $\beta + \delta < 1$, we may take $\varepsilon = 2^{-n^\delta}$, and the overall running time, which is dominated by Stage 3, is at most $\mathrm{poly}(n) \cdot 2^{n-n^\delta}$. □

4.4 Instantiating Lemma 4.4 using Nečiporuk's lower bounds

The proof of Corollary 4.2 extends unchanged to give our concrete learning results based on Nečiporuk's lower bounds:

Corollary 4.5. *There is a randomized exact learning algorithm, using membership and equivalence queries, to learn Boolean n-variable functions computable by computational model $\mathcal{A}$ of size S in time 2^{n-n^ε} for a fixed $\varepsilon > 0$, where $\mathcal{A}$ and S can be instantiated as in items (1)–(6) of Corollary 4.2.*

5 Conclusion

We initiated the study of learning algorithms with non-trivial savings and gave a range of such learning algorithms for various natural circuit classes. There are many intriguing problems left open by our work, we list a few.

- Our learning algorithms of Sections 3 and 4 are based on influential lower bound techniques in circuit complexity, namely the method of random restrictions and Nečiporuk's lower bound method. Can other proof techniques from circuit complexity, such as Razborov's method of approximations for monotone circuit lower bounds [Raz85, Raz89] or the "polynomial method" for various classes of constant-depth circuits [Bei93], similarly be leveraged to obtain non-trivial learning algorithms?

- Related to the previous item, there are several prominent circuit classes for which lower bounds are known but we do not yet have non-trivial learning algorithms; examples include intersections of poly(n) many LTFs, de Morgan formulas of size $n^{2.99}$, monotone formulas of significantly sublinear depth, monotone circuits of polynomial size, etc. Can we develop learning algorithms with nontrivial savings for these classes?

- Finally, we close with an ambitious question: in the spirit of recent celebrated work [Wil10, Wil11] wringing exciting new circuit lower bounds from non-trivial satisfiability algorithms, is it possible to leverage ideas from non-trivial learning algorithms to obtain new circuit lower bounds?

References

[Ang87] D. Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75(2):87–106, 1987. 1, 2

[Ang88] D. Angluin. Queries and concept learning. *Machine Learning*, 2:319–342, 1988. 2, 2, 4.2

[BBB$^+$96] A. Beimel, F. Bergadano, N. Bshouty, E. Kushilevitz, and S. Varricchio. On the applications of multiplicity automata in learning. In *Proceedings of the Thirty-Seventh Annual Symposium on Foundations of Computer Science*, pages 349–358, 1996. 1

[BEHW89] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. Warmuth. Learnability and the Vapnik-Chervonenkis dimension. *Journal of the ACM*, 36(4):929–965, 1989. 1

[Bei93] R. Beigel. The polynomial method in circuit complexity. In *Proceedings of the Eigth Conference on Structure in Complexity Theory*, pages 82–95, 1993. 5

[BF72] Ya M Barzdin and RV Freivald. Prediction of general recursive functions. *Doklady Akademii Nauk SSSR*, 206(3):521, 1972. 4.2

[BIS12] P. Beame, R. Impagliazzo, and S. Srinivasan. Approximating AC^0 by Small Height Decision Trees and a Deterministic Algorithm for $\#AC^0$-SAT. In *CCC*, pages 117–125, 2012. 1

[Bjö12] Andreas Björklund. Counting perfect matchings as fast as Ryser. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 914–921, 2012. 1

[Blu94] Avrim Blum. Separating distribution-free and mistake-bound learning models over the boolean domain. *SIAM J. Comput.*, 23(5):990–1000, 1994. 2

[Bsh95] N. Bshouty. Exact learning via the monotone theory. *Information and Computation*, 123(1):146–153, 1995. 1

[CIKK16] Marco L. Carmosino, Russell Impagliazzo, Valentine Kabanets, and Antonina Kolokolova. Learning Algorithms from Natural Proofs. In Ran Raz, editor, *31st Conference on Computational Complexity (CCC 2016)*, volume 50 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:24, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. 1, 1.1, 2.1, 2.1

[CKK$^+$15] Ruiwen Chen, Valentine Kabanets, Antonina Kolokolova, Ronen Shaltiel, and David Zuckerman. Mining circuit lower bound proofs for meta-algorithms. *Computational Complexity*, 24(2):333–392, 2015. 1, 1.1, 2, 2.1, 4

[FK10] Fedor V Fomin and Dieter Kratsch. *Exact exponential algorithms*. Springer Science & Business Media, 2010. 1

[FS92] P. Fischer and H. Simon. On learning ring-sum expansions. *SIAM Journal on Computing*, 21(1):181–192, 1992. 1

[GKM12] Parikshit Gopalan, Adam R. Klivans, and Raghu Meka. Learning functions of halfspaces using prefix covers. *Journal of Machine Learning Research - Proceedings Track*, 23:15.1–15.10, 2012. 1, 2.1

[GS10] P. Gopalan and R. Servedio. Learning and lower bounds for AC^0 with threshold gates. In *Proc. 14th Intl. Workshop on Randomization and Computation (RANDOM)*, pages 588–601, 2010. 1, 1.1, 3

[Hås86] Johan Håstad. *Computational Limitations for Small Depth Circuits*. MIT Press, Cambridge, MA, 1986. 1, 1.1, 3

[Hås14] Johan Håstad. On the correlation of parity and small-depth circuits. *SIAM Journal on Computing*, 43(5):1699–1708, 2014. 1.1, 3.1, 2

[HS07] L. Hellerstein and R. Servedio. On PAC learning algorithms for rich boolean function classes. *Theoretical Computer Science*, 384(1):66–76, 2007. 1, 3.1

[HSW92] D. Helmbold, R. Sloan, and M. Warmuth. Learning integer lattices. *SIAM Journal on Computing*, 21(2):240–266, 1992. 1

[IMP12] Russell Impagliazzo, William Matthews, and Ramamohan Paturi. A satisfiability algorithm for AC^0. In *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 961–972, 2012. 1, 3.1

[IW12] Taisuke Izumi and Tadashi Wadayama. A new direction for counting perfect matchings. In *53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012, New Brunswick, NJ, USA, October 20-23, 2012*, pages 591–598, 2012. 1

[Juk12] Stasys Jukna. *Boolean Function Complexity*. Springer, 2012. 4.1, 4.1

[Kan13] Daniel M. Kane. The correct exponent for the gotsman-linial conjecture. In *Proc. 28th Annual IEEE Conference on Computational Complexity (CCC)*, 2013. 1.1, 3

[KKO13] Adam Klivans, Pravesh Kothari, and Igor Carboni Oliveira. Constructing hard functions using learning algorithms. In *Proceedings of the 28th Conference on Computational Complexity, CCC 2013*, pages 86–97, 2013. 1

[KOS04] A. Klivans, R. O'Donnell, and R. Servedio. Learning intersections and thresholds of halfspaces. *Journal of Computer & System Sciences*, 68(4):808–840, 2004. 1, 3

[KS04] A. Klivans and R. Servedio. Learning DNF in time $2^{\tilde{O}(n^{1/3})}$. *Journal of Computer & System Sciences*, 68(2):303–318, 2004. 1

[KW93] M. Karchmer and A. Wigderson. On span programs. In *Proceedings of the Eighth Annual Structure in Complexity Theory Conference (San Diego, CA, 1993)*, pages 102–111. IEEE Comput. Soc. Press, Los Alamitos, CA, 1993. 4.1

[Lit88] N. Littlestone. Learning quickly when irrelevant attributes abound: a new linear-threshold algorithm. *Machine Learning*, 2:285–318, 1988. 2, 4.2

[LMN93] Nathan Linial, Yishay Mansour, and Noam Nisan. Constant depth circuits, fourier transform, and learnability. *Journal of the ACM*, 40(3):607–620, 1993. 1, 2.1, 3

[MT94] W. Maass and G. Turan. How fast can a threshold gate learn? In *Computational Learning Theory and Natural Learning Systems: Volume I: Constraints and Prospects*, pages 381–414. MIT Press, 1994. 3.1

[OS16] Igor Carboni Oliveira and Rahul Santhanam. Conspiracies between learning algorithms, circuit lower bounds and pseudorandomness. *CoRR*, abs/1611.01190, 2016. 1

[PPSZ05] Ramamohan Paturi, Pavel Pudlák, Michael E. Saks, and Francis Zane. An improved exponential-time algorithm for k-SAT. *J. ACM*, 52(3):337–364 (electronic), 2005. 1

[PPZ99] Ramamohan Paturi, Pavel Pudlák, and Francis Zane. Satisfiability coding lemma. *Chicago J. Theoret. Comput. Sci.*, pages Article 11, 19 pp. (electronic), 1999. 1

[Pud87] Pavel Pudlák. The hierarchy of boolean circuits. *Computers and artificial intelligence*, 6(5):449–468, 1987. 4.1

[Raz85] A. Razborov. Lower bounds on the monotone complexity of some boolean functions. *Dokl. Akad. Nauk SSSR*, 281:798–801, 1985. English translation in: Soviet Math. Dokl. 31:354–357, 1985. 5

[Raz89] Alexander A. Razborov. On the method of approximations. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washigton, USA*, pages 167–176, 1989. 5

[Rei11] Ben W. Reichardt. Reflections for quantum query algorithms. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 560–569. SIAM, Philadelphia, PA, 2011. 1

[Riv87] R. Rivest. Learning decision lists. *Machine Learning*, 2(3):229–246, 1987. 1

[Sch99] T Schoning. A probabilistic algorithm for k-sat and constraint satisfaction problems. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 410–414. IEEE, 1999. 1

[Sch05] Rainer Schuler. An algorithm for the satisfiability problem of formulas in conjunctive normal form. *J. Algorithms*, 54(1):40–44, 2005. 1

[Tal15] A. Tal. Tight Bounds on The Fourier Spectrum of AC^0. ECCC report TR14-174 Revision #1, available at http://eccc.hpi-web.de/report/2014/174/, 2015. 2.1, 3

[Val84] L. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984. 1

[Wil10] Ryan Williams. Improving exhaustive search implies superpolynomial lower bounds. In *STOC'10—Proceedings of the 2010 ACM International Symposium on Theory of Computing*, pages 231–240. ACM, New York, 2010. 1, 5

[Wil11] Ryan Williams. Non-uniform ACC circuit lower bounds. In *26th Annual IEEE Conference on Computational Complexity*, pages 115–125. IEEE Computer Soc., Los Alamitos, CA, 2011. 1, 5

[Wil15] Virginia V. Williams. Hardness of easy problems: basing hardness on popular conjectures such as the Strong Exponential Time Hypothesis. In *10th International Symposium on Parameterized and Exact Computation*, volume 43 of *LIPIcs. Leibniz Int. Proc. Inform.*, pages 17–29. Schloss Dagstuhl. Leibniz-Zent. Inform., Wadern, 2015. 1

A Proof of Corollary 3.2

Proof. We shall assume that the depth-$(d+1)$ circuit F is *layered*, meaning that for any gate g it contains, every directed path from an input variable to g has the same length (converting an unlayered circuit to a layered one increases its size only by a factor of d). We prove the corollary with a failure probability of $S \cdot 2^{-t}$ for such layered circuits. Let S_i denote the number of gates in layer i (at distance i from the inputs), so $S = S_1 + \cdots + S_d$.

We begin by trimming the bottom fan-in of F: applying Theorem 2 with $\mathscr{F}$ being the S_1 many bottom layer gates of F (viewed as depth-2 circuits of bottom fan-in $w = 1$) and $p_0 := 1/48$, we get that

$$\Pr_{\boldsymbol{\rho} \leftarrow \mathcal{R}_{p_0}} \left[F \restriction \boldsymbol{\rho} \text{ is not computed by a } (t, G \circ \mathsf{AC}^0(\text{depth } d, \text{ bottom fan-in } \log S))\text{-decision tree} \right] \leq S_1 \cdot 2^{-t}.$$

Let $F^{(0)}$ be any good outcome of the above, a $(t, G \circ \mathsf{AC}^0(\text{depth } d, \text{ bottom fan-in } \log S))$-decision tree. Note that there are at most 2^t many $\mathsf{AC}^0(\text{depth } d, \text{ fan-in } \log S)$ circuits at the leaves of the depth-t decision tree. Applying Theorem 2 to each of them with $p_1 := 1/(48 \log S)$ (and the 't' of Theorem 2 being $2t$) and taking a union bound over all 2^t many of them, we get that

$$\Pr_{\boldsymbol{\rho} \leftarrow \mathcal{R}_{p_1}} \left[F^{(0)} \restriction \boldsymbol{\rho} \text{ is not a } (t + 2t, G \circ \mathsf{AC}^0(\text{depth } d-1, \text{ fan-in } \log S))\text{-decision tree} \right]$$
$$\leq S_2 \cdot 2^{-2t} \cdot 2^t = S_2 \cdot 2^{-t}.$$

Repeat with $p_2 = \ldots = p_{d-1} := 1/(48 \log S)$, each time invoking Theorem 2 with its 't' being the one more than the current depth of the decision tree, so at the j-th invocation Theorem 2 is invoked with its 't' being 2^{j-1}. The claim then follows by summing the $S_1 2^{-t}$, $S_2 2^{-t}, \ldots, S_d 2^{-t}$ failure probabilities over all d stages and the fact that

$$\prod_{j=0}^{d-1} p_i = \frac{1}{48} \cdot \frac{1}{(48 \log S)^{d-1}}.$$

□