Learning Arithmetic Circuits via Partial Derivatives

Adam R. Klivans[1] and Amir Shpilka[2]

[1] Division of Engineering and Applied Sciences, Harvard
Cambridge, MA 02139, USA
klivans@eecs.harvard.edu
[2] DEAS, Harvard
LCS, MIT
Cambridge, MA, USA
amirs@deas.harvard.edu

Abstract. We present a polynomial time algorithm for learning several models of algebraic computation. We show that any arithmetic circuit whose partial derivatives induce a "low"-dimensional vector space is exactly learnable from membership and equivalence queries. As a consequence we obtain the first polynomial time algorithm for learning depth three multilinear arithmetic circuits. In addition, we give the first polynomial time algorithms for learning restricted algebraic branching programs and noncommutative arithmetic formulae. Previously only restricted versions of depth-2 arithmetic circuits were known to be learnable in polynomial time. Our learning algorithms can be viewed as solving a generalization of the well known *polynomial interpolation problem* where the unknown polynomial has a succinct representation. We can learn representations of polynomials encoding *exponentially* many monomials. Our techniques combine a careful algebraic analysis of arithmetic circuits' partial derivatives with the "multiplicity automata" techniques due to Beimel et al [BBB+00].

Keywords: PAC learning, learning with queries.

1 Introduction

Let p be an unknown multivariate polynomial over a fixed field. Given random input/output pairs chosen from some distribution D, can a computationally bounded learner output a hypothesis which will correctly approximate p with respect to future random examples chosen from D? This problem, known as the multivariate polynomial learning problem, continues to be a fundamental area of research in computational learning theory. If the learner is allowed to query the unknown polynomial at points of his choosing (instead of receiving random examples) and is required to output the exact polynomial p, then this problem is precisely the well-known polynomial interpolation problem. Both the learning and the interpolation problem have received a great deal of attention from the theoretical computer science community. In a learning context, multivariate polynomials are expressive structures for encoding information (sometimes referred to as the "algebraic" analogue of DNF formulae) while polynomial interpolation has been studied in numerous contexts and has important applications in complexity theory, among other fields [ALM+98, STV01].

Previous research on this problem has focused on giving algorithms whose running time is polynomial in the number of terms or monomials of the unknown polynomial. This is a natural way to measure the complexity of learning and interpolating polynomials when the unknown polynomial is viewed in the usual "sum of monomials" representation. That is to say, given that the polynomial $p = \sum_{i=1}^{t} m_i$ is the sum of t monomials, we may wish to output a list of these monomials (and their respective coefficients), hence using at least t time steps simply to write down the list of coefficients. Several researchers have developed powerful interpolation and learning algorithms for a variety of contexts which achieve time bounds polynomial in all the relevant parameters, including t (see for example [BBB+00, BM02, GKS94, HR99, KS01, SS96]).

1.1 Arithmetic Circuits

In this paper we are concerned with learning succinct representations of polynomials via alternate algebraic models of computation, most notably *arithmetic circuits*. An arithmetic circuit syntactically represents a multivariate polynomial in the obvious way: a multiplication (addition) gate outputs the product (sum) of the polynomials on its inputs. The input wires to the circuit correspond to the input variables of the polynomial and thus the output of the circuit computes some polynomial of the input variables. We measure the size of an arithmetic circuit as the number of gates. For example, the standard "sum of monomials" representation of a polynomial $p = \sum_{i=1}^{t} \alpha_i x_{i_1} \cdots x_{i_n}$ (α_i is an arbitrary field element) corresponds precisely to a depth-2 arithmetic circuit with a single sum gate at the root and t product gates feeding into the sum gate (each product gate multiplies some subset of the input variables). To rephrase previous results on learning and interpolating polynomials in terms of arithmetic circuits, we could say that depth-2 arithmetic circuits with a sum gate at the root are learnable in time polynomial in the size of the circuit.

Moving beyond the standard "sum of products" representation, we consider the complexity of learning higher depth arithmetic circuits. It is easy to see that there exist polynomial size depth-3 (or even depth-2 with a product gate at the root) arithmetic circuits capable of computing polynomials with *exponentially* many monomials. For example, let $\{L_{i,j}\}, 1 \leqslant i, j \leqslant n$ be a family of n^2 distinct linear forms over n variables. Then $\sum_{i=1}^{n} \prod_{i=1}^{n} L_{i,j}$ is a polynomial size depth-3 arithmetic circuit but cannot be written as a sum of polynomially many monomials. Although arithmetic circuits have received a great deal of attention in complexity theory and, more recently, derandomization, the best known result for learning arithmetic circuits in a representation other than the depth-2 sum of products representation is due to Beimel et al. [BBB+00] who show that depth-2 arithmetic circuits with a product gate of fan in $O(\log n)$ at the root and addition gates of unbounded fan-in in the bottom level are learnable in polynomial time.

1.2 Our Results

We learn various models of algebraic computation capable of encoding exponentially many monomials in their input size. Our algorithms work with respect to any distribution and require membership query access to the concept being learned. More specifically we show that any class of polynomial size arithmetic circuits whose partial derivatives induce a vector space of dimension polynomial in the circuit size are learnable in polynomial time. This characterization generalizes the work of Beimel et al. [BBB+00] and yields the following results:

- The first polynomial time algorithm for learning the class of depth-3 multilinear circuits: polynomials $p = \sum_{i=1}^{n} \prod_{j=1}^{m} L_{i,j}(X_j)$ where each $L_{i,j}$ is a linear form and the X_j's are a partition of the input variables.
- An algorithm for learning general depth-3 arithmetic circuits with s product gates each of fan in at most d in time polynomial in s, 2^d, and n, the number of variables.
- The first polynomial time algorithm for learning polynomial size noncommutative formulae computing polynomials over a fixed partition of the variables (note there are no depth restrictions on the size of the formula).
- The first polynomial time algorithm for learning polynomial size read once, oblivious algebraic branching programs.

We also show that polynomial size depth-3 circuits with a product gate at the root are learnable in polynomial time (more generally we can learn the product of a polynomial number of depth-3 multilinear circuits in polynomial time). Finally we show that, with respect to known techniques, it is hard to learn polynomial size depth-3 homogeneous arithmetic circuits in polynomial time. This indicates that our algebraic techniques give a fairly tight characterization of the learnability of arithmetic circuits with respect to current algorithms.

1.3 Our Techniques

We use as a starting point the work on multiplicity automata due to Beimel et al. [BBB+00]. A multiplicity automaton is a nondeterministic finite automaton where each transition edge has an integer weight (for a precise definition see Section 2). On input x, $f(x)$ is equal to to the sum over all accepting paths of the automaton on input x of the product of the edge weights on that accepting path. In [BBB+00], the authors, building on work due to [OSK94], show that multiplicity automata can be learned in polynomial time and that these multiplicity automata can compute polynomials in their standard sum of products representation[1]. Their analysis centers on the Hankel matrix of a multiplicity automata (see section 2 for a definition).

We generalize their results and show that any polynomial whose partial derivatives induce a low dimensional vector space has a low rank Hankel matrix. We conclude that any arithmetic circuit or branching program whose partial derivatives form a low dimensional vector space can be computed by polynomial size multiplicity automata and are amenable to the learning algorithms developed by Beimel et al. As such, we output a multiplicity automaton as our learner's hypothesis.

Our next task is to show which circuit classes have partial derivatives that induce low dimensional vector spaces. At this point we build on work due to Nisan [Nis91] and Nisan and Wigderson [NW97] (see also [SW01]) who analyzed the partial derivatives of certain arithmetic circuit classes in the context of proving lower bounds and show that a large class of algebraic models have "well-behaved" partial derivatives. For example we show that the dimension of the vector space of partial derivatives induced by a multilinear depth-3 arithmetic circuit is polynomial in the size of the circuit.

We feel that partial derivatives are the "correct" algebraic quantity to analyze for learning irreducible polynomials[2]. By considering partial derivatives, we can show that there are depth-3 polynomial size homogeneous arithmetic circuits which are not computed by polynomial size multiplicity automata and are thus not learnable by these techniques. As depth-3 homogeneous circuits would be the natural next step in this line of research, we feel that we have captured precisely which circuits are learnable with respect to known techniques. Finally, we feel that that a function's partial derivatives are more easily understood than its Hankel matrix and hence may lead to more improvements in learning polynomials.

1.4 The Relationship to Lower Bounds

In the case of learning Boolean functions, the ability to prove lower bounds against a class of Boolean circuits usually coincides with the ability to give strong learning algorithms for those circuits. For example the well known lower bounds of Håstad [Has86] against constant depth Boolean circuits are used heavily in the learning algorithm due to Linial, Mansour, and Nisan [LMN93]. Jackson et al. [JKS02] have shown that constant depth circuits with a majority gate, one of the strongest circuit classes for which we can prove lower bounds (see [ABFR91]), also admit nontrivial learning algorithms. Furthermore Jackson et al. [JKS02] show that we will not be able to learn more complicated Boolean circuits unless certain cryptographic assumptions are false.

We believe our work furthers this relationship in the algebraic setting. The models of algebraic computation we can learn correspond to a large subset of the models of algebraic computation for which lower bounds are known. For example Nisan [Nis91] gives lower bounds for noncommutative formulae. Nisan and Wigderson [NW97] prove lower bounds against depth-3 multilinear circuits. Moreover, in both papers the authors prove lower bounds by considering the partial derivatives spanned by the circuit and the function computed by it, a method similar to ours. No exponential lower bounds are known for general depth-3 arithmetic circuits (see [GR98, SW01]). As in the Boolean case, we exploit many of the insights from the lower bound literature to prove the correctness of our learning algorithms.

1.5 Organization

In Section 2 we review relevant learning results for multiplicity automata as well as state some basic facts from algebraic complexity. In Section 3 we prove our main theorem, characterizing the learnability of arithmetic circuits via their partial derivatives. In Sections 4, 5, and 6 we state our main learning results for various arithmetic circuits and algebraic branching programs. In Section 7 we give an algorithm for learning polynomial size depth-3 circuits with a product gate at the root which uses only factorization and interpolation algorithms (the results and techniques from Section 7 are self-contained and are unrelated to the learning algorithms in previous sections).

---

[1] More generally they can learn any polynomial p of the form $p = \sum_{i=1}^{n} \prod_{j=1}^{m} p_{ij}(x_j)$ where $p_{ij}(x_j)$ is a univariate polynomial of low degree.

[2] Reducible polynomials can first be factored.

2 Preliminaries

2.1 The Learning Model

We will work in the model of exact learning from membership and equivalence queries, first introduced by Angluin [Ang87]. In this model a learner begins with some candidate hypothesis h for an unknown concept f and is allowed access to both a *membership query* oracle and an *equivalence query* oracle. The membership query oracle takes as input x and outputs $f(x)$. The equivalence query oracle takes as input the learner's current hypothesis h and outputs a counterexample, namely an input y such that $h(y) \neq f(y)$. We assume that making a membership or equivalence query of length k takes time k. If no such counterexample exists then we say that the learner has exactly learned f. We say that a concept f is exactly learnable in time t if there exists a exact learner for f whose running time is bounded by t. A concept class is considered to be exactly learnable in polynomial time if for every f in the concept class there exists an exact learner for f running in time polynomial in the size of the smallest description of f. Known transformations imply that if a concept class is exactly learnable in polynomial time then it is also learnable in Valiant's PAC model in polynomial time with membership queries.

2.2 Multiplicity Automata

A multiplicity automaton is a nondeterministic automaton where each transition is assigned a weight, and the output of the automaton for any input x is the sum over all accepting paths of x of the products of the weights on each path.

Definition 1. *Fix Σ an alphabet with $s = |\Sigma|$ elements. A multiplicity automaton A of size r over alphabet Σ is a vector of integers $\bar{\gamma} = \{\gamma_1, \ldots, \gamma_r\}$ and a set of s matrices μ_σ, one for each $\sigma \in \Sigma$, where each μ_σ is an $r \times r$ matrix of integers. The output of A on input $x = (x_1, \ldots, x_n)$ is defined to be the inner product of $(\prod_{i=1}^{n} \mu_{x_i})_1$ and $\bar{\gamma}$ where $(\prod_{i=1}^{n} \mu_{x_i})_1$ equals the first row of $\prod_{i=1}^{n} \mu_{x_i}$. In other words the output is the first coordinate of the vector $(\prod_{i=1}^{n} \mu_{x_i}) \cdot \bar{\gamma}$.*

Intuitively each matrix μ_σ corresponds to the transition matrix of the automaton for symbol $\sigma \in \Sigma$. Iterative matrix multiplication keeps track of the weighted sum of paths from state i to state j for all $i, j \leqslant r$. The first row of the iterated product corresponds to transition values starting from the initial state and $\bar{\gamma}$ determines the acceptance criteria.

Next we define the Hankel matix of a function:

Definition 2. *Let Σ be an alphabet and $f : |\Sigma|^n \to \mathbb{F}$. Consider an ordering of all strings in $\Sigma^{\leqslant n}$ and let $x \in \Sigma^d$ and $y \in \Sigma^{n-d}$ for some $d \leqslant n$. Construct a matrix H where the (x, y) entry of H is equal to $f(x \cdot y)$. For any other pair of strings (a, b) such that $|a| + |b| \neq n$ let $H_{a,b} = 0$. H is called the Hankel matrix of f for strings of length n. We define H_k to be the kth "block" of H, i.e. H_k is the submatrix defined by all rows of H indexed by strings of length exactly k and all columns of H indexed by strings of length exactly $n - k$.*

The following key fact relates the rank of the Hankel matrix of a function for strings of length n with the size of multiplicity automaton computing f on inputs of length n:

Theorem 1. *[CP71, Fli74, BBB$^+$00] Let $f : \Sigma^n \to \mathbb{F}$ for some fixed field $\mathbb{F}$. Then the rank of the Hankel matrix of f (over $\mathbb{F}$) is equal to the size of the smallest multiplicity automaton computing f on inputs of length n.*

Previous learning results have computed the rank of the Hankel matrices of particular polynomials yielding a bound on the size of their multiplicity automata. In fact, Beimel et al. [BBB$^+$00], improving on [OSK94], learn functions computed by multiplicity automata by iteratively learning their corresponding Hankel matrices:

Theorem 2. *[BBB$^+$00] Let $f : \Sigma^n \to \mathbb{F}$ for some fixed field $\mathbb{F}$. Let r be the rank of the Hankel matrix of f for strings of length n. Then there exists an exact learning algorithm for f running in time polynomial in n, r, and $|\Sigma|$. Furthermore the final hypothesis output by the learning algorithm is a multiplicity automaton of size r over alphabet Σ.*

Our main technical contribution is to show that the rank of a function's Hankel matrix is bounded by (and in most cases equal to) the dimension of the vector space of a function's partial derivatives. Thus we reduce the problem of learning a polynomial to bounding the dimension of the vector space of its partial derivatives.

2.3 Learning Functions Over Large Fields

Our algorithms run in time polynomial in both the size of the arithmetic circuit to be learned as well as the size of the field the circuit is defined over. As long as the size of the field grows polynomially with the input length our running times remain polynomial. In the case that we wish to learn an arithmetic over a large (or infinite) field, it will suffice to learn that circuit over a subset of the field of size polynomial in the input length. This is because the arithmetic circuits we wish to learn correspond to polynomials where the degree of each variable is bounded by a polynomial in n (although the polynomial will have exponentially many monomials, each monomial will be of bounded degree), and we can apply standard techniques to interpolate the value of the circuit on an input chosen from a large field from several inputs chosen from a subset of the field (see [BBB$^+$00]).

2.4 Structured Polynomials

In this paper we will work primarily with polynomials that respect a fixed partition of the input variables:

Definition 3. *Let $X = \dot\cup_{i=1}^d X_i$ be a partition of the variables into d sets each of size n. A polynomial over the variables X is called* structured *and has degree d if every monomial m is of the form $y_1 \cdot y_2 \cdots y_d$ where each y_i is some variable from X_i. Thus any structured f is also homogeneous and multilinear.*

For example, the class of depth-3 multilinear circuits, as defined by Nisan and Wigderson [NW97], computes only structured polynomials. To see this note that any polynomial computed by a depth-3 multilinear circuits is of the form $p = \sum_{i=1}^n \prod_{j=1}^m L_{i,j}(X_j)$ where each $L_{i,j}$ is a linear form and the X_j's are a partition of the input variables. In later sections we will show that certain algebraic branching programs also compute structured polynomials and will therefore be amenable to our learning techniques.

Another notation that we use is the following:

Definition 4. *Let $X = \dot\cup_{i=1}^d X_i$. For any $1 \leqslant k \leqslant d$ define*

$$\mathbb{M}[X_1, ..., X_k] = \{\ M \mid M = \prod_{i=1}^k x_i\ , x_i \in X_i\ \}.$$

Thus $\mathbb{M}[X_1, ..., X_k]$ is the set of all structured monomials of degree k.

2.5 Partial Derivatives

In this subsection we introduce some notation for computing partial derivatives.

Definition 5. *Let $d = \sum_{i=1}^k d_i$. For a function $f(x_1, ..., x_n)$ and a a monomial $M = \prod_{i=1}^k {x_i}^{d_i}$ we define*

$$\frac{\partial f}{\partial M} = \frac{1}{M!} \cdot \frac{\partial^d f}{\prod_{i=1}^k \partial {x_i}^{d_i}}$$

Where $M! = \prod_{i=1}^k (d_i!)$.

Example 3 *Let $f(x, y, z) = x^2y + z$, $M(x, y, x,) = xy$. We have that*

$$\frac{\partial f}{\partial M} = 2x,\ M! = 1$$

Definition 6. *For a function $f(x_1, ..., ..., x_n)$ and $k \leqslant n$ define*

$$\partial_k(f) = \{ \ \frac{\partial f}{\partial M} \mid \text{monomials M} \in \mathbb{F}[\text{x}_1, ..., \text{x}_\text{k}] \ \}$$

We define

$$\text{rank}_k(f) = \dim\left(\text{span}\left(\partial_k(f)\right)\right)$$

In the case that f is a sturctured multilinear polynomial we define $\partial_k(f)$ in a slightly different way.

Definition 7. *Let f be a structured polynomial of degree d over $X = \dot\bigcup_{i=1}^{d} X_i$. We define*

$$\partial_k(f) = \{ \ \frac{\partial f}{\partial M} \mid M \in \mathbb{M}[X_1, ..., X_k] \ \}$$

As before we define

$$\text{rank}_k(f) = \dim\left(\text{span}\left(\partial_k(f)\right)\right)$$

Notice that for every $M \in \mathbb{M}[X_1, ..., X_k]$ we have that $\frac{\partial f}{\partial M}$ is a stuctured polynomial in $X_{k+1}, ..., X_d$.

We now show how to compute $\text{rank}_k(f)$ by considering the rank of the partial derivative *evaluation* matrix and *coefficient* matrix:

Definition 8. *(Evaluation Matrix) Let f be a structured polynomial over $X = \dot\bigcup_{i=1}^{d} X_i$. Fix a field $\mathbb{F}$ and lexicographically order all elements of $\mathbb{F}^{d-k}$. Construct a matrix E_k where the rows are indexed by elements of $\mathbb{M}[X_1, ..., X_k]$ and the columns are indexed by elements of $\mathbb{F}^{d-k}$. Then the M, ρ entry of E_k is equal to*

$$\frac{\partial f}{\partial M}(\rho)$$

That is the value of the partial derivative of f with respect to M at the point ρ.

Definition 9. *(Coefficient Matrix) Let f be a structured polynomial over $X = \dot\bigcup_{i=1}^{d} X_i$. Lexicographically order the sets $\mathbb{M}[X_1, \dots, X_k]$ and $\mathbb{M}[X_{k+1}, \dots, X_d]$. Construct a matrix N_k where the rows are indexed by elements of $\mathbb{M}[X_1, \dots, X_k]$ and the columns are indexed by elements of $\mathbb{M}[X_{k+1}, \dots, X_d]$. Then the M_1, M_2 entry of N_k is equal to the coefficient of monomial M_2 in $\frac{\partial f}{\partial M_1}$.*

The following is a standard fact from linear algebra:

Fact 4 *Let f be a structured polynomial over $X = \dot\bigcup_{i=1}^{d} X_i$. Then $\text{rank}_k(f)$ is equal to both the rank of E_k and the rank of N_k where E_k and N_k are defined as above.*

Although we use $\text{rank}_k(f)$ both for structured and unstructured polynomials it will be clear from the context which is the case.

3 Characterizing Learnability via Partial Derivatives

In this section we present our main criterion for determining whether an arithmetic circuit or algebraic branching program is learnable. We prove that any structured polynomial whose partial derivatives form a low degree vector space induce low rank Hankel matrices. To relate the rank of the Hankel matrix of C to its partial derivatives we will need the following multivariate version of Taylor's theorem:

Fact 5 *Let $X = \dot\bigcup_{i=1}^{d} X_i$ and let $f(X_1, \dots, X_d)$ be a polynomial in nd variables respecting the partition. Let $\rho = (\rho_1, \dots, \rho_d)$ be an assignment to the variables. For a monomial M define $M(\rho)$ be value of M on assignment ρ. Then*

$$f(\rho_1, \dots, \rho_d) = \sum_{M \in \mathbb{M}[X_1, \dots, X_k]} M(\rho) \frac{\partial f}{\partial M}(\rho)$$

Now we can state the main technical theorem of the paper:

Theorem 6. *Let $X = \dot\cup_{i=1}^{d} X_i$ be a partition of at most nd variables and let $f(X_1, \ldots, X_d)$ be a polynomial respecting this partition. Then for every $k \leqslant d$,*

$$\dim(H_k(f)) = \mathrm{rank}_k(f)$$

Proof of Theorem 6:

We will define two matrices V and D such that the rank of D will equal $\mathrm{rank}_k(f)$ and that $H_k = V \cdot D$. Then we will show that the rank of $H_k = \mathrm{rank}(D) = \mathrm{rank}_k(f)$.

- **Construction of V**:
  Fix a lexicographic ordering of all monomials in $\mathbb{M}[X_1, \ldots, X_k]$ and all assignments ρ to the variables $X_1, \ldots, X_k$ (note that there might be infinitely many such assignments). Create a generalized Vandermonde matrix V where the rows are indexed by all assignments ρ and the columns are indexed by all elements of $\mathbb{M}[X_1, ..., X_k]$. For an assignment ρ and a monomial $M \in \mathbb{M}[X_1, ..., X_k]$ the (ρ, M) entry of V is defined to be $M(\rho)$.

- **Construction of D**:
  Again fix a lexicographic ordering of all monomials in $\mathbb{M}[X_1, \ldots, X_k]$ and all assignments σ to the variables $X_{k+1}, \ldots, X_d$. Create a matrix D where the rows are indexed by all elements in $\mathbb{M}[X_1, ..., X_k]$ and the columns are indexed by all assignments σ. For a monomial M and an assignment σ the (M, σ) entry of D is defined to be $\frac{\partial f}{\partial M}(\sigma)$.

Claim. $\mathrm{rank}(D) = \mathrm{rank}_k(f)$

Proof. Row M of D is the evaluation of $\frac{\partial f}{\partial M}$ on all inputs. Hence each vector corresponds to the "truth table" of a particular partial derivative of f. By Fact 4, the rank of this matrix equals the dimension of the span of the partial derivatives of f with respect to all $M \in \mathbb{M}[X_1, ..., X_k]$.

Claim. $V \cdot D = H_k(f)$

Proof. Consider $V \cdot D$. Notice that for $\rho = \rho_1, \ldots, \rho_k$ and $\sigma = \sigma_{k+1}, \ldots, \sigma_d$, the (ρ, σ) entry of $V \cdot D$ is equal to

$$(V \cdot D)_{\rho,\sigma} = \sum_{M \in \mathbb{M}[X_1, \ldots, X_k]} M(\rho) \frac{\partial f}{\partial M}(\sigma)$$

which by Fact 5 equals $f(\rho_1, \ldots, \rho_k, \sigma_{k+1}, \ldots, \sigma_d)$.

Thus H_k is exactly equal to $V \cdot D$. Furthermore it is easy to check that V is a generalized Vandermonde matrix and has full rank. Therefore we can conclude

$$\mathrm{rank}(H_k) = \mathrm{rank}(D) = \mathrm{rank}_k(f).$$

□

By summing over all values of k we obtain

Corollary 1. *Let f be a polynomial respecting a partition $X = \dot\cup_{i=1}^{d} X_i$. Then*

$$\dim(H(f)) = \sum_{i=1}^{d} \mathrm{rank}_k(f).$$

Using a similar argument we can prove an analogous theorem for polynomials that are not necessarily structured:

Theorem 7. *Let $f : \mathbb{F}^n \to \mathbb{F}$ be a polynomial in n variables where each variable has degree less than d. Then $\dim(H(f)) \leqslant \sum_{i=1}^{n} \mathrm{rank}_k(f)$. If $|\mathbb{F}|$ is larger than d then the previous equation is an equality.*

4 Learning Multilinear Depth-3 Arithmetic Circuits

In this section we learn a form of depth-3 arithmetic circuits first studied by Nisan and Wigderson [NW97] called multilinear depth-3 arithmetic circuits. Although strong lower bounds have been proved for this circuit class (see [NW97]), previously no learning algorithms were known. We begin by defining general depth-3 arithmetic circuits:

Definition 10. *A depth 3 arithmetic circuit is a layered graph with 3 levels and unbounded fan-in. At the top we have either a sum gate or a product gate. A depth 3 arithmetic circuit $\mathcal{C}$ with a sum (product) gate at the top is called a $\Sigma\Pi\Sigma$ ($\Pi\Sigma\Pi$) circuit and has the following structure:*

$$\mathcal{C} = \sum_{i=1}^{s} \prod_{j=1}^{d} L_{i,j}(X)$$

where each $L_{i,j}$ is a linear function in the input variables $X = x_1, \ldots, x_n$. The size of the circuit is the number of gates, in this case $O(sd)$.

A $\Sigma\Pi\Sigma$ circuit is a homogeneous circuit if all the linear forms are homogeneous linear forms and all the product gates have the same fan in (or degree). In other words every gate of the circuit computes a homogeneous polynomial. To define a multilinear depth-3 circuit we need to impose a partition on the variables:

Definition 11. *A $\Sigma\Pi\Sigma$ circuit is called multilinear with respect to $X = \dot{\bigcup}_{i=1}^{d} X_i$ if every linear function computed at the bottom is a homogeneous linear form in one of the sets X_i, and each multiplication gate multiplies d homogeneous linear forms $L_1, \ldots, L_d$ where every L_i is over a distinct set of variables X_i. That is to say a depth-3 multilinear circuit $\mathcal{C}$ has the following structure:*

$$\mathcal{C} = \sum_{i=1}^{s} \prod_{j=1}^{d} L_{i,j}(X_j)$$

where $L_{i,j}$ is an homogeneous linear form.

The following lemma characterizes the dimension of a multilinear circuit's partial derivatives:

Lemma 1. *If a polynomial f is computed by a multilinear depth 3 circuit with s product gates then for every $1 \leqslant k \leqslant d$,*

$$\mathrm{rank_k(f)} \leqslant \mathrm{s}.$$

Proof. First notice that for every product gate

$$P = \prod_{i=1}^{d} L_i(X_i)$$

we have $\mathrm{rank_k(P)} \leqslant 1$. Indeed for any monomial $M \in \mathbb{M}[X_1, ..., X_k]$ we have that

$$\frac{\partial P}{\partial M} = \alpha_M \cdot \prod_{i=k+1}^{d} L_i(X_i)$$

for some constant α_M depending on M and P. The proof of the lemma now follows from the linearity of the partial derivative operator.

Applying Lemma 1, Theorem 6, and Theorem 2 we obtain the following learning result:

Theorem 8. *Let $\mathcal{C}$ be a multilinear depth-3 circuit with s product gates over n variables with coefficients from a field $\mathbb{F}$. Then $\mathcal{C}$ is learnable in time polynomial in s,n, and $|\mathbb{F}|$.*

4.1 Learning General Depth 3 Circuits

In this subsection we give a learning algorithm for general depth-3 arithmetic circuits. Unlike the algorithm in the multilinear case, our algorithm runs in time exponential in the degree of the circuit (and polynomial in the other parameters). Thus we can learn in subexponential time any depth-3 circuit of sublinear degree. Our algorithm follows from the following lemma:

Lemma 2. *If a polynomial $f : \mathbb{F}^n \to \mathbb{F}$ be a polynomial over a variable set X of size n computed by a depth-3 circuit with s product gates each of degree at most d then for every $1 \leqslant k \leqslant d$,*

$$\mathrm{rank_k(f)} \leqslant \binom{\mathrm{d}}{\mathrm{k}} \cdot \mathrm{s}$$

Proof. The proof is similar to the case of multilinear depth-3 circuits. Notice that for every product gate

$$P = \prod_{i=1}^{d} L_i(X)$$

we have $\mathrm{rank_k(P)} \leqslant \binom{\mathrm{d}}{\mathrm{k}}$. For any monomial M of degree k we have that

$$\frac{\partial P}{\partial M} \in \operatorname{span}\left\{ \prod_{i \in T} L_i(X) \mid T \subset [d],\ |T| = d - k \right\}$$

Since there are at most s product gates we obtain the claimed bound.

Applying the above lemma with Theorem 7 and Theorem 2 we obtain the following learning result:

Theorem 9. *Let $f : \mathbb{F}^n \to \mathbb{F}$ be computed by a depth-3 arithmetic circuit with s product gates each of fan in at most d. Then f is learnable in time polynomial in $n, 2^d$, and s.*

4.2 Discussion

The fact that the rank of f was bounded by the number of product gates is unique to multilinear depth-3 circuits. For example consider the following depth-2 $\Pi\Sigma$ circuit:

$$f(z, x_1, ..., x_n) = \prod_{i=1}^{n}(z + x_i).$$

For every ordering of the variables the dimension of the span of the partial derivatives of f, and hence the rank of the Hankel matrix of f, is exponential in n; this follows from the observation that the coefficient of z^d is the $n - d$ symmetric polynomial whose partial derivatives have dimension $2^{\Omega(n-d)}$ (see [SW01]). Thus it is no surprise that Beimel et al. [BBB$^+$00] only considered depth-2 $\Pi\Sigma$ circuits where the product gate at the root has fan in at most $O(\log n)$; fan in larger than $O(\log n)$ would correspond to Hankel matrices of superpolynomial dimension and thus would not be learnable by multiplicity automata techniques.

On the other hand, in Section 7, we show how to learn $\Pi\Sigma$ circuits where the product gate has large fan in (our learning results hold only over large enough finite fields) using black box factorization techniques. To show the limits of current learning techniques we point out that the following homogeneous depth-3 arithmetic circuit

$$C' = \prod_{i=1}^{n}(z + x_i) + \prod_{i=1}^{n}(v + u_i)$$

is both irreducible and has exponentially many independent partial derivatives. As its degree is n we can only learn it in time exponential in n. We leave open the problem of learning homogeneous depth-3 arithmetic circuits (as well as the more difficult problem of learning general depth-3 arithmetic circuits) of super logarithmic degree.

5 Learning Classes of Algebraic Branching Programs

Algebraic and Boolean branching programs have been an intense area of study for complexity theorists; it has been a particularly fruitful model for proving lower bounds. Considerably less is known in the learning scenario– Bshouty et al. [BTW96] and Bergadano et al. [BBTV97] have shown some partial progress for learning restricted width Boolean branching programs. In this section we will show how to learn any polynomial size algebraic branching program that is both read once and oblivious. As such we will be able to show that multiplicity automata are essentially equivalent to read once, oblivious algebraic branching programs, a characterization that may be of independent interest. We begin with a general definition of algebraic branching programs:

Definition 12. *An algebraic branching program (ABP), first defined by Nisan [Nis91], is a directed acyclic graph with one vertex of in-degree zero, which is called* source, *and one vertex of out-degree zero, which is called the* sink. *The vertices of the graph are partitioned into levels numbered* $0, ..., d$. *Edges are labeled with a homogeneous linear form in the input variables and may only connect vertices from level* i *to vertices from level* $i+1$. *The source is the only vertex at level* 0 *and the sink is the only vertex at the level* d. *Finally the size of the ABP is the number of vertices in the graph.*

The polynomial that is computed by an ABP is the sum over all directed paths from the source to the sink of the product of linear functions that labeled the edges of the path. It is clear that an ABP with $d+1$ levels computes a homogeneous polynomial of degree d.

In this section we will show how to learn a natural restriction of an algebraic branching program as mentioned above: the read once, oblivious algebraic branching program or ROAB.

Definition 13. *Let* $X = \dot\bigcup_{i=1}^{d} X_i$ *be a partition of the input variables into* d *disjoint sets. An ABP is* oblivious *if for every level* i *only one set of variables* X_j *appears. A function is a ROAB, a read once, oblivious algebraic branching program, if it is an oblivious ABP and every set of variables* X_j *appears in at most one level.*

Without loss of generality we will assume that for a ROAB f and partition $X = \dot\bigcup_{i=1}^{d} X_i$ the variables X_i appear on edges from level $i-1$ to level i. It is straightforward to check that a ROAB computes a structured multilinear polynomial in X_1,...,X_d. In this section we measure the complexity of a polynomial in terms of its smallest ROAB:

Definition 14. *For a polynomial* f *we define* $B(f)$ *to be the size of the smallest ABP for* f. *For a structured multilinear polynomial* f *we denote* $OB(f)$ *to be the size of the smallest ROAB for* f.

The main theorem of this section shows that for structured polynomials, the size of its smallest ROAB is equal to the dimension of the vector space induced by its partial derivatives:

Theorem 10. *For a structured multilinear polynomial* $f(X_1, ..., X_d)$ *we have that*

$$\sum_{k=1}^{d} \mathrm{rank_k(f)} = \mathrm{OB(f)}.$$

To prove Theorem 10 we will need the following theorem which is implicit in Nisan [Nis91]:

Theorem 11. *[Nis91] Let* f *be a structured multilinear polynomial. For each* $0 \leqslant k \leqslant d$ *define a matrix* $\mathcal{M}_k(f)$ *as follows:*

- *Each row is labeled with a monomial* $M_1 \in \mathbb{M}[X_1, \ldots, X_k]$.
- *Each column is labeled with a monomial* $M_2 \in \mathbb{M}[X_{k+1}, \ldots, X_d]$.

If $k = 0$ *then* $M_1 = 1$ *and if* $k = d$ *then* $M_2 = 1$. *The* (M_1, M_2) *entry of* $\mathcal{M}_k(f)$ *is equal to the coefficient of the monomial* $M_1 \cdot M_2$ *in* f. *We have that*

$$OB(f) = \sum_{k=0}^{d} \mathrm{rank}(\mathcal{M}_\mathrm{k}(\mathrm{f})).$$

Proof of Theorem 10:

We will show that $\text{rank}(\mathcal{M}_k(f)) = \text{rank}_k(f)$ which, combined with Theorem 11, completes the proof. Consider a row of $\mathcal{M}_k(f)$ corresponding to some monomial $M \in \mathbb{M}[X_1, \ldots, X_k]$. Since f is a structured polynomial with respect to X, $\frac{\partial f}{\partial M}$ is equal to $\sum_t \alpha_t M_t$ where each α_t is an element of the field and $M_t \in \mathbb{M}[X_{k+1}, \ldots, X_d]$ for all t. Notice, however that row M of $\mathcal{M}_k(f)$ is precisely equal to the row vector $(\alpha_1, \ldots, \alpha_t)$. Hence row M of $\mathcal{M}_k(f)$ is equal to the coefficients of the partial derivative of f viewed as a polynomial in $X_{k+1}, \ldots, X_d$. Applying Fact 4 completes the proof. □

Combining Theorem 10 and Theorem 2 we obtain

Theorem 12. *Let f be computed over a field $\mathbb{F}$ by a ROAB of size s and set of input variables X. Then f is learnable in time polynomial in $s, |\mathbb{F}|$, and $|X|$.*

5.1 Equivalence of ROABs and Multiplicity Automata

We can now prove that ROABs are essentially equivalent to multiplicity automata. Since our learning algorithm outputs as a hypothesis a multiplicity automata, Theorem 12 implies that every ROAB of size s is computed by a multiplicity automata of size s. We cannot show that every multiplicity automaton is computed by a ROAB, but we can show that every multiplicity automaton is computed by a ROAB which computes higher degree polynomials at each edge.

Definition 15. *Define a ROAB of degree d to be a ROAB where every edge is labelled with a polynomial of degree d.*

Lemma 3. *Let f be any polynomial over n variables computed by an algebraic multiplicity automata of size r. Assume also the the degree of each variable in f is bounded by d. Then f can be computed by a ROAB of size nr and degree d.*

Proof. Let $S \subset \Sigma$ be a subset of the alphabet of size $d+1$. Let f be computed by a multiplicity automaton A of size r. Construct a matrix T where the i, j entry of T is a degree d univariate polynomial, $T_{i,j}$, interpolating the (i, j) entry of μ_σ for every $\sigma \in S$. That is, $T_{i,j}(\sigma)$ is the (i, j) entry of μ_σ. Consider a ROAB with n levels each of size r where every level has a copy of the r states of A. Connect every vertex at level j to every vertex at level $j+1$ and label edge (a, b) with the polynomial in the (a, b) entry of T. The Schwartz-Zippel lemma [Sch80, Zip79] implies that this ROAB of degree d computes A.

6 Learning Noncommutative Formulae

In this section we will show how to learn another type of arithmetic circuits: polynomial size noncommutative formulae. A noncommutative formula is an arithmetic formula where multiplication does not necessarily commute; i.e. different orderings of inputs to a product gate result in different outputs. This may seem to be a strange restriction, but it is very natural in the context of function computation where an ordering is enforced on groups of variables. For example the product of k matrices $M_1, \ldots, M_k$ where matrix M_i uses variables from a set X_i can be viewed as a structured noncommutative formula over an ordering of the variables $X = \dot{\cup} X_i$ (changing the order of the matrices will result in a different output).

Nisan proved the first lower bounds for noncommutative formulae in [Nis91]; here we will give the first learning algorithm for structured polynomials computed by noncommutative formulae. Previously only algorithms for learning read-once arithmetic formulae were known (see [BB98]). We begin with a general definition for arithmetic formulae:

Definition 16. *An arithmetic formula is a tree whose edges are directed towards the root. The leafs of the tree are labeled with input variables. Any inner vertex is labeled with one of the arithmetic operations $\{+, *\}$. Every edge is labeled with a constant from the field in which we are working. The size of the formula is defined to be the number of vertices.*

An arithmetic formula computes a polynomial in the obvious manner. A formula is noncommutative if for any two input variables x_i and x_j, $x_i x_j - x_j x_i \neq 0$.[3] Intuitively this restriction makes it difficult for a formula to use the power of cancellation. For a polynomial f we denote $F(f)$ to be the size of the smallest noncommutative formula computing f.

In [Nis91] Nisan proved exponential lower bounds on the size of noncommutative formula computing the permanent and the determinant. An important ingredient of Nisan's result is the following lemma relating noncommutative formula size to algebraic branching program size:

Lemma 4. *[Nis91] For any structured polynomial f of degree d we have that*

$$B(f) \leqslant d(F(f)+1)$$

Using this we can give the following relationship between noncommutative formulae and ROABs:

Theorem 13. *Let f be a structured polynomial of degree d computed by a noncommutative formula f of size s, then f is computed by a ROAB of size $d(s+1)$.*

Proof. Applying Lemma 4 we see that f is computed by an algebraic branching program B of size $d(s+1)$. We will show that B is also computed by a ROAB of size $d(s+1)$. Consider the set of edges in B from level $i-1$ to i. Assume that two different sets of variables appear from level $i-1$ to level i say X_i and X_j. Then the output of B will contain a monomial of the form and Yx_jZ where $x_j \in X_j$, Y is a set of variables appearing in levels less than $i-1$ in B, and Z is the set of variables appearing in levels greater than i in B. But by assumption f is a noncommutative formula enforcing a fixed ordering of variables, so the coefficient of any monomial Yx_jZ must be zero. As such, we can substitute the constant 0 for all of the variables X_j appearing on these edges and obtain an oblivious branching program B' computing the same polynomial as B. B' can be made read-once in a similar fashion.

Combining Theorem 13 with Theorem 12 we obtain

Theorem 14. *Let f be a structured polynomial of degree d over a variable set X computed by a noncommutative formula of size s with coefficients from a field $\mathbb{F}$. Then f is learnable in time polynomial in d, s, $|X|$, and $|\mathbb{F}|$.*

7 Learning $\Pi\Sigma\Pi$ Circuits

In Beimel et. al [BBB+00], the authors give a polynomial time learning algorithm for depth-2 arithmetic circuits with a product gate at the root where the product gate has fan in $O(\log n)$ and the fan in of each sum gate is unbounded. Here we observe that depth-3 $\Pi\Sigma\Pi$ arithmetic circuits where every product gate has fan in at most t over n variables can be learned in time polynomial in t and n. Our algorithm will only make membership queries to the unknown arithmetic circuit, but it will have some small probability of failure. Therefore we consider our learning result to hold in the PAC model with membership queries (as opposed to other results in the paper which hold for the model of exact learning with membership and equivalence queries). Similar methods show that we can also learn a product of multilinear $\Sigma\Pi\Sigma$ circuits. We will need to use a "black-box" factorization result originally due to Kaltofen and Trager [KT90] and refined by Huang and Rao [HR99]. The following is implicit in [HR99]:

Theorem 15. *[KT90, HR99] Let p be a polynomial with n variables over a finite field $\mathbb{F} = GF(q)$ such that $p = \prod_{i=1}^{t} p_i$ where each p_i is an irreducible polynomial in at most n variables. Then there exists a probabilistic algorithm querying p as a black box and running in time polynomial in $n, t,$ and $\log q$ which outputs t programs $B_1, \ldots, B_t$ with the following properties:*

- *With high probability $p = \prod_{i=1}^{t} B_i$.*
- *Every B_i can be evaluated in time polynomial in n, t and $\log q$.*

[3] Evaluation of the polynomial over a field corresponds to evaluation in the normal sense, i.e. multiplication of field elements *is* commutative. Here the polynomial $x_i x_j - x_j x_i$ is a non-zero formula even though it evaluates to zero on every point.

Remark: For the above algorithm to succeed, the results of [HR99] require that we can evaluate the polynomial p at an extension field $GF(q^k)$ where $k \geqslant c(n \log n)$ for some sufficiently large constant c. The algorithm still works in time polynomial in $\log q^k$, but if the polynomial cannot be queried at such an extension then our learning algorithm will hold only for sufficiently large finite fields.

The factorization algorithm above outputs an implicit representation or program computing each factor of p. At this point we can apply the sparse interpolation algorithm due to Klivans and Spielman [KS01] to explicitly output the linear forms contained in each B_i:

Theorem 16. *[KS01] Let p be a polynomial in n variables with at most m monomials of total degree d over a field $\mathbb{F} = GF(q)$. Then there exists a deterministic algorithm for identifying p exactly using only membership queries which runs in time polynomial in $n, m, d,$ and $\log q$.*

Again the above theorem requires q to be larger than some fixed polynomial in n, and d. Combining Theorem 15 and Theorem 16 we obtained the claimed learning result:

Theorem 17. *Let p be a polynomial with n variables over a field $GF(q)$ computed by a $\Pi\Sigma\Pi$ depth-3 arithmetic circuit where each product gate has fan in t. Then p is PAC learnable from membership queries only in time polynomial in $t, n,$ and $\log q$ assuming either q is large enough to satisfy the hypothesis of Theorem 15 or we can evaluate p from a suitably large extension field.*

Similar arguments give

Theorem 18. *Let p be a polynomial with n variables over a field $GF(q)$ computed by a $\Pi\Sigma\Pi\Sigma$ depth-4 arithmetic circuit where the top product gate multiplies multilinear depth-3 $\Sigma\Pi\Sigma$ circuits. Then p is PAC learnable from membership queries only in time polynomial in $t, n,$ and $\log q$ assuming either q is large enough to satisfy the hypothesis of Theorem 15 or we can evaluate p from a suitably large extension field.*

8 Acknowledgments

We thank Ran Raz for many helpful discussions in all stages of this work. We also thank Eli Ben-Sasson for important conversations at an early stage of this research.

References

[ABFR91] J. Aspnes, R. Beigel, M. Furst, and S. Rudich. The expressive power of voting polynomials. In Baruch Awerbuch, editor, *Proceedings of the 23rd Annual ACM Symposium on the Theory of Computing*, pages 402–409, New Orleans, LS, May 1991. ACM Press.

[ALM+98] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and the hardness of approximation problems. *Journal of the ACM*, 45(3):501–555, May 1998.

[Ang87] Dana Angluin. Queries and concept learning. *Machine Learning*, 2:319, 1987.

[BB98] Daoud Bshouty and Nader H. Bshouty. On interpolating arithmetic read-once formulas with exponentiation. *Journal of Computer and System Sciences*, 56(1):112–124, February 1998.

[BBB+00] Amos Beimel, Francesco Bergadano, Nader H. Bshouty, Eyal Kushilevitz, and Stefano Varricchio. Learning functions represented as multiplicity automata. *J. ACM*, 47(3):506–530, 2000.

[BBTV97] F. Bergadano, N. H. Bshouty, C. Tamon, and S. Varricchio. On learning branching programs and small depth circuits. In Shai Ben-David, editor, *Proceedings of the 3rd European Conference on Computational Learning Theory*, volume 1208 of *LNAI*, pages 150–161, Berlin, March 17–19 1997. Springer.

[BM02] N. Bshouty and Y. Mansour. Simple learning algorithms for decision trees and multivariate polynomials. *SICOMP: SIAM Journal on Computing*, 31, 2002.

[BTW96] Nader H. Bshouty, Christino Tamon, and David K. Wilson. On learning width two branching programs. In *Proc. 9th Annu. Conf. on Comput. Learning Theory*, pages 224–227. ACM Press, New York, NY, 1996.

[CP71] J. W. Carlyle and A. Paz. Realizations by stochastic finite automata. *Journal of Computer and System Sciences*, 5(1):26–40, February 1971.

[Fli74] Michel Fliess. Matrices de Hankel. *J. Math. Pures et Appl.*, 53:197–224, 1974.

[GKS94] Dima Grigoriev, Marek Karpinski, and Michael F. Singer. Computational complexity of sparse rational interpolation. *SIAM Journal on Computing*, 23(1):1–11, February 1994.

[GR98] D. Grigoriev and A. A. Razborov. Exponential complexity lower bounds for depth 3 arithmetic circuits in algebras of functions over finite fields. In IEEE, editor, *39th Annual Symposium on Foundations of Computer Science: proceedings: November 8–11, 1998, Palo Alto, California*, pages 269–278, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1998. IEEE Computer Society Press.

[Has86] Johan Hastad. Almost optimal lower bounds for small depth circuits. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 6–20, Berkeley, California, 28–30 May 1986.

[HR99] M. Huang and T. Rao. Interpolation of sparse multivariate polynomials over large finite fields with applications. *ALGORITHMS: Journal of Algorithms*, 33, 1999.

[JKS02] Jeffrey C. Jackson, Adam R. Klivans, and Rocco A. Servedio. Learnability beyond AC$\hat{0}$. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC-02)*, pages 776–784, New York, May 19–21 2002. ACM Press.

[KS01] Adam R. Klivans and Daniel Spielman. Randomness efficient identity testing of multivariate polynomials. In ACM, editor, *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing: Hersonissos, Crete, Greece, July 6–8, 2001*, pages 216–223, New York, NY, USA, 2001. ACM Press.

[KT90] Erich Kaltofen and Barry M. Trager. Computing with polynomials given by black boxes for their evaluations: Greatest common divisors, factorization, separation of numerators and denominators. *Journal of Symbolic Computation*, 9(3):300–320 (or 301–320??), March 1990.

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

[Nis91] Noam Nisan. Lower bounds for non-commutative computation (extended abstract). In *Proceedings of the Twenty Third Annual ACM Symposium on Theory of Computing*, pages 410–418, New Orleans, Louisiana, 6–8 May 1991.

[NW97] N. Nisan and A. Wigderson. Lower bounds on arithmetic circuits via partial derivatives. *CMPCMPL: Computational Complexity*, 6, 1997.

[OSK94] H. Ohnishi, H. Seki, and T. Kasami. A polynomial time learning algorithm for recognizable series. *TIEICE: IEICE Transactions on Communications/Electronics/Information and Systems*, 1994.

[SS96] Robert E. Schapire and Linda M. Sellie. Learning sparse multivariate polynomials over a field with queries and counterexamples. *Journal of Computer and System Sciences*, 52(2):201–213, April 1996.

[Sch80] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM*, 27(4):701–717, October 1980.

[STV01] M. Sudan, L. Trevisan, and S. Vadhan. Pseudorandom generators without the XOR lemma. *JCSS: Journal of Computer and System Sciences*, 62, 2001.

[SW01] A. Shpilka and A. Wigderson. Depth-3 arithmetic circuits over fields of characteristic zero. *CMPCMPL: Computational Complexity*, 10, 2001.

[Zip79] R. Zippel. Probabilistic algorithms for sparse polynomials. In *ISSAC '79: Proc. Int'l. Symp. on Symbolic and Algebraic Computation,* Lecture Notes in Computer Science, Vol. 72. Springer-Verlag, 1979.