Learning DNFs and Circuits using Teaching Assistants

N. V. Vinodchandran

Department of Computer Science and Engineering
University of Nebraska-Lincoln
email: vinod@cse.unl.edu

Abstract. In this paper we consider the complexity of learning monotone DNF formulae and Boolean circuits in an exact learning model called the *teaching assistant model.* This model, introduced in one of our earlier works, uses teaching assistant classes for classifying learning problems. Teaching assistant classes are a learning-theoretic analog of complexity classes. We show that monotone DNFs are learnable using a teaching assistant in the class SPP. On the other hand, Boolean circuits and general DNFs are not learnable using an SPP teaching assistant unless NP $\subseteq$ SPP. We also show that learnability of Boolean circuits with an assistant in the class NP will improve the Karp-Lipton collapse result to P^{NP}.

1 Introduction

In this paper we study the complexity of learning certain concept classes in an exact learning model called the *teaching assistant* model. The teaching assistant model, developed in our earlier work [3, 4], is a refinement of Angluin's well-known exact learning model. In Angluin's model a learner learns a target concept by making either membership queries or equivalence queries (or both) to an honest teacher. In the teaching assistant model, a complexity-theoretic approach is taken towards learning.

The Teaching Assistant model: A learning model based on complexity classes. A central idea in this model is the notion of a *teaching assistant*; an intermediate agent between the learner and the teacher. The learner, instead of querying the teacher directly, queries a teaching assistant to learn the target concept. The teaching assistant in turn makes membership queries to the teacher to gather information about the target concept. The usefulness of this model for investigating the complexity of learning problems comes from the fact that we can classify teaching assistants into *teaching assistant classes*, which is similar to the way we classify computational problems into complexity classes.

This complexity-theoretic model for learning is closely related to Angluin's exact learning model. An equivalence query to a teacher can be replaced by a sequence of queries to an NP oracle, which in turn makes membership queries to the teacher. In particular, an NP-like set $\{\langle y, r\rangle \mid y \text{ is a prefix of a counter example to the hypothesis } r\}$ can be used to prefix-search for a counter example for an equivalence query r. This NP oracle is an example of a teaching assistant. Hence, we can say that classes learnable with equivalence queries in Angluin's model are also learnable in the teaching assistant model with an NP teaching assistant. Similarly, a membership query can be seen as a query to a teaching assistant in the class P. Therefore, any concept classes learnable with only membership queries in Angluin's model are also learnable with a P teaching assistant.

The main advantage of restating the learnability using queries in the language of NP and P is that we can generalize this idea to other complexity classes. For example, consider the complexity class UP. UP is the class of decision problems accepted by nondeterministic machines with a restriction that there is at most one accepting computation for any instance of the problem. The complexity of problems in UP is believed to be strictly between P and NP. Hence we can say that the concept classes that are learnable using a teaching assistant in the class UP have less learning complexity than concept classes that require assistants from the class NP. In a certain well-defined sense, the complexity of teaching assistants quantify the complexity of learning. In general, we can define a teaching assistant class analogous to any complexity class. Hence, introducing this intermediary called the teaching assistant in the exact learning model provides a framework which is suitable for analyzing the complexity of learning problems. We call this framework the teaching assistant model (TA model) for exact learning.

In this paper we are interested in the teaching assistant classes analogous to the complexity classes NP and SPP. While NP is a well-known class, some motivation needs to be given for studying the class SPP.

The complexity class SPP and its significance. SPP was introduced and studied by Fenner, Fortnow, and Kurtz in [8]. This class was also independently studied in [14] under the name XP and in [9] under the name $\mathcal{Z}$UP). SPP consists of decision problems that are accepted by nondeterministic polynomial time Turing machines such that; for positive instances, the difference in the number of accepting and rejecting paths of the machine is exactly one, and for negative instances, this difference is zero. SPP can be seen as the *gap* (difference between accepting and rejecting paths) analog of the class UP.

From a structural point of view, this class is interesting mainly because of its *lowness* property. A complexity class C_1 is said to be *low* for another (relativizable) complexity class C_2 if $C_2^{C_1} = C_2$. That is, any C_2-machine that uses a language in C_1 as an oracle can be converted into another C_2-machine with the same acceptance behavior without any oracle access: C_1 is powerless as oracle to C_2. It is shown in [8] that SPP is low for counting complexity classes such as PP, $\mathrm{C_{=}P}$, and $\mathrm{Mod}_k\mathrm{P}$ for $k \geq 2$. Thus, intuitively, SPP is a class of "low counting complexity".

Although SPP is a structurally defined class, it has been proved to be very useful in analyzing the complexity of "intermediate problems"− problems that are not known to be in P, but unlikely to be NP-complete. Membership of a problem in the class SPP can be seen as evidence that the problem is unlikely to be NP-complete: no known NP-complete problem has been shown to be in

SPP. In [13, 15] it is shown that many of the known candidate problems of intermediate complexity from the domain of algebra are in the class SPP. Very recently, Arvind and Kurur [2] presented an elegant SPP algorithm for the well-known Graph Isomorphism problem.

The exact power of SPP in relation to NP is not yet clear. Because of the strong restrictions on the computation tree of nondeterministic Turing machines accepting SPP languages, it seems surprising that NP could be in SPP. Furthermore, there are oracles separating NP from SPP [5]. On the other hand, results from derandomization and resource bounded measure theory give some evidence towards the containment of NP in SPP [12, 11].

Prior results on teaching assistant model. In [3], the teaching assistant framework was introduced primarily to distinguish the complexity of learning certain algebraic concepts such as vector spaces and permutation groups from the nonalgebraic concept class CNF. In particular, it was shown that vector spaces can be learned using teaching assistants in the class SPP. Learning CNFs using a teaching assistant in the class SPP is not possible, unless NP $\subseteq$ SPP. A subsequent work initiated a study of teaching assistant classes analogous to many well-studied complexity classes such as P, UP, NP, (UP$\cap$co-UP), (NP$\cap$co-NP), Σ_2^p, SPP, LWPP, and established their importance in classifying the complexity of learning [4]. In particular, [4] gave a fine separation among many of the teaching assistant classes mentioned above. For example, this paper presented a concept class that is learnable using a (UP$\cap$coUP) teaching assistant, but not learnable with teaching assistants in P. Note that, unlike in complexity theory where establishing proper separations among complexity classes is extremely difficult, we can actually show separations among TA classes.

New results in this paper. In this paper, we prove new results on the complexity of learning monotone DNFs and Boolean circuits in the TA model. First, we show that monotone DNFs are learnable using teaching assistants in the class SPP. On the other hand, general DNFs and Boolean circuits are not learnable using teaching assistants in the class SPP unless NP $\subseteq$ SPP. We also show that if Boolean Circuits are learnable with an NP assistant then we can improve the current best upper bound of the Karp-Lipton collapse result; under this assumption we show that SAT $\in$ P/poly $\Rightarrow$ PH $\subseteq$ P^{NP}. The current best upper bound is S_2^p; a complexity class that contains P^{NP}. Previously, it was known that SAT $\in$ P/poly $\Rightarrow$ PH $\subseteq$ $\mathrm{ZPP}^{\mathrm{NP}}$ (attributed to Watanabe in [6]). More recently, Cai showed that $\mathrm{S}_2^\mathrm{p} \subseteq \mathrm{ZPP}^{\mathrm{NP}}$ [7]. Cai's result together with the result that SAT $\in$ P/poly $\Rightarrow$ PH $\subseteq$ S_2^p (attributed to Sengupta in [7]) improved the Karp-Lipton upper bound from $\mathrm{ZPP}^{\mathrm{NP}}$ to S_2^p. There are relativized worlds where SAT has polynomial size circuits and PH $\not\subseteq$ P^{NP} [10, 16]. Therefore, it appears that in order to prove learnability of Boolean circuits with NP-assistants, we need non-relativizing techniques.

Because of the space limitations, we only give sketches of the proofs in this extended abstract. Some of the proofs are omitted completely.

In complexity theory, the class SPP has been used successfully in studying computational problems with intermediate complexity. The results in this paper help to further establish the role of the class SPP in understanding the complexity of learning problems.

2 Notations, definitions, and technical preliminaries

We follow standard language-theoretic notations in this paper. For sets X_1 and X_2, $X_1 \setminus X_2$ denotes their set difference. Let λ denote the empty string. For $y_1, y_2 \in \Sigma^*$, $y_1 \leq y_2$ means that y_1 is lexicographically smaller than or equal to y_2. Let $\mathbf{Z}$ denote the set of integers and $\mathbf{N}$ the set of natural numbers.

Complexity theory. We use standard definitions and notations from complexity theory. In particular, oracle Turing machines play a central role in many of the notions in this paper. Please refer to standard text books for these definitions and notations. For a nondeterministic polynomial time (in short, NP) machine M, $acc_M : \Sigma^* \rightarrow \mathbf{N}$ denotes the function defined as: $acc_M(x) =$ the number of accepting paths of M on input x. #P denotes the class of functions $\{acc_M | M \text{ is an NP machine}\}$. $gap_M : \Sigma^* \rightarrow \mathbf{Z}$ denotes the function defined as: $gap_M(x)=$ (number of accepting paths of M on input x − the number of rejecting paths of M on input x). GapP denotes the class of functions $\{gap_M | M \text{ is an NP machine}\}$ [8]. GapP is the closure of #P under subtraction. We will make implicit use of the following closure properties of GapP: a) the product of a polynomial number of GapP functions is a GapP function, b) the sum of a polynomial number of GapP functions is a GapP function (see [8] for details).

Using the notion of gaps we can define the complexity class SPP as follows. A language L is in SPP if there is a function f in GapP such that; $f(x) = 1$ if $x \in L$ and $f(x) = 0$ if $x \notin L$. For the purpose of comparison, we give the definition of the class UP. A language L is in UP, if there is a function f in #P such that; $f(x) = 1$ if $x \in L$ and $f(x) = 0$ if $x \notin L$. Thus, SPP generalizes the class UP. The inclusions P $\subseteq$ UP $\subseteq$ SPP are a direct consequence of the definitions.

Learning theory. We only deal with the learnability of finite concepts. A *representation class* $\mathcal{P}$ is a tuple $\langle R, \mu, p \rangle$ where p is a polynomial, $R \subseteq \Sigma^*$ and μ is a collection $\mu = \{\mu_n\}_{n \in \mathbf{N}}$ such that $\mu_n : R \rightarrow 2^{\Sigma^{p(n)}}$ is a many-one mapping. Any element in the range of μ is called a *concept* of $\mathcal{P}$. The set of all concepts of $\mathcal{P}$ is called the *concept class* represented by $\mathcal{P}$. When there is no confusion, we denote the concept class represented by $\mathcal{P}$ also by $\mathcal{P}$. For any n, let $\mathcal{P}_n$ denote the concepts of $\mathcal{P}$ in $\Sigma^{p(n)}$. For any concept $c \in \mathcal{P}_n$ let *size*(c) denote $\min\{|r| : \mu_n(r) = c\}$.

Although Angluin's query model is very closely related to our model, we omit the definitions for this model since we do not use it in any essential manner. Instead, we directly give formal definitions related to the teaching assistant model. Please refer to Angluin's original paper for definitions related to this query model [1]. We start with the abstract definition of a teaching assistant.

Definition 1. *Let $\mathcal{P}$ be a representation class. Let $Q : \mathbf{N}\times\mathbf{N}\times\Sigma^*\times 2^{\Sigma^*} \to \{0, 1\}$ be a predicate. A tuple $\langle n, l, x, c\rangle \in \mathbf{N} \times \mathbf{N} \times \Sigma^* \times 2^{\Sigma^*}$, is said to be $\mathcal{P}$-valid if $c \in \mathcal{P}_n$ and size$(c) \leq l$. The* teaching assistant *defined by Q w. r. t. the representation class $\mathcal{P}$ is the set $L(\mathcal{P}) = \{\langle n, l, x, c\rangle | \langle n, l, x, c\rangle$ is $\mathcal{P}$-valid and $Q(n, l, x, c)\}$.*

Let $\mathcal{P} = \langle R, \mu, p\rangle$ be a representation class and $L(\mathcal{P})$ be a teaching assistant for $\mathcal{P}$. A learner is a deterministic oracle Turing machine transducer. The learning algorithm for $\mathcal{P}$ comprises of a pair − a learner α, and a teaching assistant $L(\mathcal{P})$. The teacher selects a target concept $c \in \mathcal{P}_n$. The learner α, on input $\langle 0^n, 0^l\rangle$, has to output a representation $r \in R$ such that $\mu_n(r) = c$, provided that *size*$(c) \leq l$. α can query the teaching assistant $L(\mathcal{P})$. When α wants to query the teaching assistant about $\langle n, l, x, c\rangle$, it writes x on its oracle tape. Notice that other parameters, n, l and c, are fixed prior to learning and are implicitly communicated to the teaching assistant. For such a query x by the learner, the learner receives the answer 'Yes' if $\langle n, l, x, c\rangle \in L(\mathcal{P})$ and 'No' if $\langle n, l, x, c\rangle \notin L(\mathcal{P})$. After a finite number of steps, α outputs a representation of the target concept c. $\mathcal{P}$ is said to be learnable with an assistant $L(\mathcal{P})$ if there is a learner α for $\mathcal{P}$ with queries to $L(\mathcal{P})$ as teaching assistant. We say α learns $\mathcal{P}$ with teaching assistant $L(\mathcal{P})$.

Let $\mathcal{P}$ be a representation class. Let α be a learner learning $\mathcal{P}$ with a teaching assistant $L(\mathcal{P})$. For an input $\langle 0^n, 0^l\rangle$, the *time complexity* of a learner is the number of steps it takes before it writes down a representation of the target concept. $\mathcal{P}$ is said to be deterministic polynomial-time learnable with a teaching assistant $L(\mathcal{P})$, if there is a polynomial q and a deterministic learner α learning $\mathcal{P}$ with $L(\mathcal{P})$ such that for all n, and for all $c \in \mathcal{P}_n$, on input $\langle 0^n, 0^l\rangle$, the running time of learner α learning c, is bounded by $q(n + l)$. Adapting the complexity-theoretic notation FP for functional polynomial time, we denote polynomial-time learnability by FP-learnability.

In order to quantify the complexity of teaching assistants we can define classes of teaching assistants with respect to a given representation class. These classes can be defined analogous to some well-studied complexity classes. In this paper we are interested in the teaching assistant class analogous to the complexity classes SPP and NP. We define the teaching assistant class SPP. The class NP can be defined analogously.

Definition 2. *Let $\mathcal{P}$ be a representation class and $L(\mathcal{P})$ be a teaching assistant. $L(\mathcal{P})$ is said to be in the class* SPP$(\mathcal{P})$ *if there exists a polynomial-time non-deterministic oracle Turing machine M which for any $\mathcal{P}$-valid tuple $\langle n, l, x, c\rangle$, on input $\langle 0^n, 0^l, x\rangle$, uses c as oracle and produces a gap=1 if $\langle n, l, x, c\rangle \in L(\mathcal{P})$ and gap=0 if $\langle n, l, x, c\rangle \notin L(\mathcal{P})$. The behavior of machine M is not specified on inputs that are not $\mathcal{P}$-valid.*

We have the following definition of learnability with teaching assistant classes.

Definition 3. *Let $\mathcal{C}$ denote a teaching assistant class. We say a representation class $\mathcal{P}$ is* FP*-learnable with a $\mathcal{C}$-assistant if there exists a teaching assistant $L(\mathcal{P}) \in \mathcal{C}(\mathcal{P})$ and an* FP*-learner α, such that α learns $\mathcal{P}$ with the teaching assistant $L(\mathcal{P})$.*

The following theorem relates the TA model to Angluin's model.

Theorem 1 ([4]). *Let $\mathcal{P}$ be a representation class. Then $\mathcal{P}$ is* FP*-learnable with membership queries if and only if it is* FP*-learnable with a* P*-assistant. If $\mathcal{P}$ is* FP*-learnable with membership and equivalence queries, then it is* FP*-learnable with an* NP*-assistant.*

3 Learning using SPP assistants

In this section, we give a learning algorithm for monotone DNF using a teaching assistant in the class SPP. We also note that general DNFs and Boolean circuits are not learnable using SPP unless NP $\subseteq$ SPP. Note that, from Theorem 1 and the fact that monotone DNFs are learnable with membership and equivalence queries, they are also learnable with an NP-assistant.

Let mDNF denotes the concept class of monotone functions represented by monotone DNF formulae. Let CIRCUITS denote the representation class of boolean circuits. For ease of presentation, we define the size of a concept c in mDNF as the number of prime implicants of c. This is only polynomially different from the regular notion of size and does not affect our algorithm.

Theorem 2. *The representation class* mDNF *is* FP*-learnable with an* SPP*-assistant.*

Proof. (*Sketch*) We will first present a teaching assistant $L(\text{mDNF})$ and show that there exists an oracle machine M with the required gap behavior accepting $L(\text{mDNF})$. We then present a learner Learner-mDNF which uses $L(\text{mDNF})$ to output a monotone formula for the target concept.

The learning algorithm that we give is an adaptation of Angluin's algorithm for learning monotone DNFs using membership and equivalence queries. Our algorithm works as follows. Let c be the concept represented by a monotone formula ϕ. As in the case of Angluin's algorithm for mDNF, the learning algorithm will find out each of the prime implicants of ϕ one by one. This is done by doing a prefix-search with the aid of the teaching assistant. In order to make the TA in SPP, the prime implicants are prefix-searched in the order of increasing number of variables in them. The teaching assistant that we use for the learner is defined as:

$L(\text{mDNF}) = \{\langle n, l, \phi, k, y, c\rangle | \phi$ is a disjunction of monotone terms; there are no $x \in (c \setminus \phi)$ having less than k 1s; there exists $z \in (c \setminus \phi)$ having exactly k 1s; y is a prefix of lex-first such $z\}$

Claim. $L(\text{mDNF}) \in \text{SPP}(\text{mDNF})$

Proof (of claim). The nondeterministic oracle machine M that witnesses $L(\text{mDNF})$ in SPP(mDNF) is defined below. M uses an auxiliary oracle nondeterministic machine N_2 as subroutine, which in turn uses N_1. We define these two auxiliary machines first.

MACHINE$N_1^c(\langle n, l, \phi, i, k, y\rangle)$
1. **Guess** in lex-first order strings $z_1, z_2, \ldots, z_i$
2. **Accept** if all the following conditions are met
3. y is a prefix of z_1
4. $z_1, z_2, \ldots, z_i$ have exactly k $1s$
5. $z_1, z_2, \ldots, z_i$ are in c (using membership queries to c)
6. $z_1, z_2, \ldots, z_i$ are not in ϕ
7. **Else Reject**

MACHINE$N_2^c(\langle n, l, \phi, k, y\rangle)$
1. Produce a gap of $\sum_{i=1}^{l} acc_{N_1^c}(\langle n, l, \phi, i, k, y\rangle) \prod_{j=i+1}^{l} \left(1 - acc_{N_1^c}(\langle n, l, \phi, j, k, y\rangle)\right)$

MACHINE $M^c(\langle n, l, \phi, k, y\rangle)$
1. If ϕ is not a disjunction of monotone terms produce a gap=0
2. Else produce a gap of $gap_{N_2^c}(\langle n, l, \phi, k, y\rangle) \prod_{k'=1}^{k-1} \left(1 - gap_{N_2^c}(\langle n, l, \phi, k', y\rangle)\right)$

Let c be a target concept fixed by the teacher with $size(c) \leq l$. Let k_0 be the number of variables in the smallest (in terms of the number of variables) prime implicant of c that is not in ϕ. Let i_0 be the number of such prime implicants with k_0 variables.

We will argue that M produces a gap=1 if $(\langle n, l, \phi, k, y, c\rangle) \in L(\text{mDNF})$ and produces a gap=0 if $(\langle n, l, \phi, k, y, c\rangle) \notin L(\text{mDNF})$. Since if ϕ is not a disjunction of monotone terms M produces a gap=0, we will assume otherwise for the rest of the proof. $\langle n, l, \phi, k, y, c\rangle \in L(\text{mDNF})$ if and only if $k = k_0$ and y is a prefix of lex-first string $z \in (c \setminus \phi)$ with exactly k ones. We need to show that M^c produces a gap=1 in this case and produces a gap=0 otherwise. For brevity of presentation, we only present the case when y is the prefix of the lex-first string $z \in (c \setminus \phi)$ with exactly k ones. The other case can be argued similarly.

We first observe certain acceptance behavior of N_2 which is stated as the next claim. The proof follows from the descriptions of N_1 and N_2.

Claim (subclaim).

1. If $k < k_0$, then $gap_{N_2^c}(\langle n, l, \phi, k, y\rangle) = 0$.
2. If $k = k_0$, then $gap_{N_2^c}(\langle n, l, \phi, k, y\rangle) = 1$.

*Proof (***sketch for subclaim).**

1 $k < k_0$. In this case N_1^c will have 0 accepting paths. Hence, whatever be the value of i, $gap_{N_1^c}(\langle n, l, \phi, i, k, y\rangle) = 0$. Hence, the total gap produced by $N_2^c = 0$.

2 $k = k_0$. Split the sum in line 1 of the description of N_2^c into three cases: $i < i_0$, $i = i_0$, and $i > i_0$. For $i > i_0$, N_1^c will have 0 accepting paths and hence that part of the sum contributes 0 to the overall gap. For $i = i_0$, N_1^c will have exactly 1 accepting path and for this case each of terms in the product contributes 1. Thus the contribution to the overall gap in this case is 1. For the case $i < i_0$, consider the term $\prod_{j=i+1}^{l} \left(1 - acc_{N_1^c}(\langle n, l, \phi, j, k, y\rangle)\right)$.

Since $i < i_0$, there is term in this product with $j = i_0$. For this value of $j = i_0$, $acc_{N_1^c}(\langle n, l, \phi, i_0, k, y\rangle) = 1$ and hence $\left(1 - acc_{N_1^c}(\langle n, l, \phi, i_0, k, y\rangle)\right) = 0$. Hence the contribution from this case to the overall gap =0. Therefore, the total gap=1. □

Case 1: $k = k_0$. In this case we need to show that $gap_{M^c} = 1$. Consider the gap expression in the line 2 of the description of M^c. From the above claim, $gap_{N_2^c}(\langle n, l, \phi, j, k, y\rangle) = 1$ and for all $k' \le k - 1$, $gap_{N_2^c}(\langle n, l, \phi, j, k', y\rangle) = 0$. Hence the overall gap produced is 1.
Case 2: $k < k_0$. We need to show that $gap_{M^c} = 0$. From the above claim, the term $gap_{N_2^c}(\langle n, l, \phi, j, k, y\rangle) = 0$ and hence the overall gap produced is 0.
Case 3: $k > k_0$. (This case is similar to the case $i < i_0$ in the above claim) In this case we need to show that $gap_{M^c} = 0$. Consider the product term $\prod_{k'=1}^{k-1} \left(1 - gap_{N_2^c}\left(\langle n, l, \phi, k', y\rangle\right)\right)$ of the gap expression in the line 2 of the description of M^c. Since k' runs from 1 to $k - 1$, if $k > k_0$ there is a term $\left(1 - gap_{N_2^c}\left(\langle n, l, \phi, k_0, y\rangle\right)\right)$ that appear in this product. Now from the subclaim, for this value of k', $\left(1 - gap_{N_2^c}\left(\langle n, l, \phi, k_0, y\rangle\right)\right) = 0$ and the product term, and consequently the overall gap, is 0.

The following algorithm learns mDNF in polynomial time by making only $O(ln)$ queries to the teaching assistant L(mDNF). In the algorithm, for a binary string $y \in \{0, 1\}^n$, Y denotes the the corresponding monotone term.

Learner-mDNF($0^n, 0^l$)
1. $\phi \leftarrow$ **F**
2. $k \leftarrow 1$
3. **while** $\langle 0^n, \phi, k, \lambda, c\rangle \notin L$
4. $\quad k \leftarrow k + 1$
5. **if** $k = n + 1$
6. $\quad$ **then** output ϕ and stop
7. $y \leftarrow \lambda$
8. **for** $j = 1$ to n **do**
9. $\quad$ **if** $\langle 0^n, \phi, k, y1, c\rangle \in L$
10. $\quad\quad$ **then** $y \leftarrow y1$
11. $\quad\quad$ **else** $y \leftarrow y0$
12. $\phi \leftarrow \phi \vee Y$
13. **goto** 2

Finally we note that any representation class for which the equivalence problem (checking whether two strings represent the same concept) is co-NP-complete cannot be learned using an SPP assistant unless NP $\subseteq$ SPP. We state this as our next theorem for the case of CIRCUITS and general DNFs. The proof essentially follows the line of argument for the proof of a similar result ([4], Theorem 7.4).

Theorem 3. *If the representation classes of* CIRCUITS *or* DNF *are learnable with an* SPP*-assistant then* NP $\subseteq$ SPP*.*

4 Learning with NP assistants

In this section we consider learnablity using NP-assistants. Since learning with NP-assistants generalizes learning in Angluin's model with equivalence + membership queries, it is one of the most interesting teaching assistant classes.

Theorem 4. *If* CIRCUITS *are* FP*-learnable with an* NP*-assistant then* SAT $\in$ P/poly $\Rightarrow$ PH $\subseteq$ P^{NP}

Proof. (*Sketch*) We will present the proof sketch here. We will use the self-reducibility structure of SAT to prove the theorem. Under this assumption we will show how to construct a circuit for SAT using P^{NP} machine. The construction is done inductively. For every n let C_n be the circuit that accepts SAT with n variables.

For the sake of simplicity, we omit the size parameter l in this proof. Let α be a learner, which on input 0^n learns a polynomial size circuit for SAT with n variables with an NP-assistant $L(\text{CIRCUITS}) = \{\langle n, y, c\rangle | y \in \Sigma^*; Q(n, y, c)\}$, where Q is the predicate defining the NP-assistant. Let the oracle machine N accept $L(\text{CIRCUITS})$. We will focus on α's computation when the target circuit is the circuit for SAT.

Consider the following machine N' (now without oracles) which on input $\langle 0^n, y, C\rangle$, simulates N with target concept SAT. Whenever N makes an oracle query f to SAT, it constructs two formulae f_0 and f_1 where f_i for $i \in \{0, 1\}$ is the formula f with the n^{th} variable replaced with i. Then, it checks whether $C(f_0) = 1$ or $C(f_1) = 1$. If yes then it continues simulation as if the query f is answered "YES"; otherwise it proceeds as if the query f is answered "NO". Let L be the NP language accepted by this machine N'.

Now consider the following FP machine M that constructs the circuit C_n for SAT inductively using the NP language L. When n is a constant, M exhaustively searches for the circuit for SAT. Now assume that M has constructed C_{n-1}. M simulates the learner α with SAT as target concept. Whenever α makes a query y to the teaching assistant, M makes a query $\langle 0^n, y, C_{n-1}\rangle$ to L and continues simulation. It is easy to see that M constructs the circuit for SAT. Therefore if CIRCUITS is learnable using an NP-assistant, then if SAT $\in$ P/poly, SAT has P^{NP}-uniform circuits. Now, using standard techniques, it can be shown that $\Sigma_2^{\mathrm{P}} \subseteq \mathrm{P}^{\mathrm{NP}}$, which implies PH $\subseteq \mathrm{P}^{\mathrm{NP}}$.

It is known that for any class $\mathcal{C}$ which contains NP, SAT $\in$ P/poly $\Rightarrow$ PH $\subseteq \mathcal{C}$, then $\mathcal{C}$ has languages with polynomial circuit complexity for any fixed polynomial. We have the following corollary whose proof is omitted.

Corollary 1. *If* CIRCUITS *is learnable with an* NP*-assistant, then for any* k*,* P^{NP} *has languages that does not have circuits of size* n^k*.*

Perhaps the most important concept class that is worth investigating in the TA model is general DNF formulae. An open problem regarding DNF learning in Angluin's model is whether equivalence + membership queries are powerful enough for learning DNF formulae. A natural step towards progress in this problem is to investigate it in a more general learning framework than Angluin's model. The TA model with NP teaching assistants provides exactly such a framework. Are general DNF formulae learnable using an NP assistant?

Acknowledgments. I thank Chris Bourke for suggestions which helped me improve the presentation. I thank John Hitchcock for discussions. I thank the anonymous referees for comments which improved the presentation of the paper.

References

1. D. Angluin. Queries and concept learning. *Machine Learning*, 2:319–342, 1988.
2. V. Arvind and P. Kurur. Graph isomorphism is in SPP. In *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science*, pages 743 –750, 2002.
3. V. Arvind and N. V. Vinodchandran. The complexity of exactly learning algebraic concepts. In *Proceedings of the 7th International Workshop on Algorithmic Learning Theory*, volume 1160 of *Lecture Notes in Artificial Intelligence*, pages 100–112, 1996.
4. V. Arvind and N. V. Vinodchandran. Exact learning via teaching assistants. *Theoretical Computer Science*, 241:51–81, 2000. Special issue devoted to the 7th International Workshop on Algorithmic Learning Theory.
5. R. Beigel. Perceptrons, PP, and the polynomial hierarchy. *Computational Complexity*, 4(4):339–349, 1994.
6. N. Bshouty, R. Cleve, R. Gavaldà, S. Kannan, and C. Tamon. Oracles and queries that are sufficient for exact learning. *Journal of Computer and System Sciences*, 52:421–433, 1996.
7. J-Y. Cai. $\mathrm{S}_2^{\mathrm{P}} \subseteq \mathrm{ZPP}^{\mathrm{NP}}$. In *Proceedings of the 42nd Annual IEEE Symposium on Foundations of Computer Science*, pages 620–629, 2001.
8. S. Fenner, L. Fortnow, and S. Kurtz. Gap-definable counting classes. *Journal of Computer and System Sciences*, 48:116–148, 1994.
9. S. Gupta. Closure properties and witness reduction. *Journal of Computer and System Sciences*, 50(3):412–432, 1995.
10. H. Heller. On relativized exponential and probabilistic complexity classes. *Information and Control*, 71:231–243, 1986.
11. J. M. Hitchcock. The size of SPP. *Theoretical Computer Science*. To Appear.
12. A. Klivans and D. van Melkebeek. Graph nonisomorhism has subexponential size proofs unless the polynomial-time hierarchy collapses. *SIAM Journal on Computing*, 31:1501–1526, 2002.
13. J. Köbler, U. Schöning, and J. Torán. Graph isomorphism is low for PP. *Journal of Computational Complexity*, 2:301–330, 1992.
14. M. Ogiwara and L. Hemachandra. A complexity theory of feasible closure properties. *Journal of Computer and System Sciences*, pages 295–325, 1993.
15. N. V. Vinodchandran. Counting complexity of solvable group problems. *SIAM Journal on Computing*, 2004. To Appear.
16. C. B. Wilson. Reltivized circuit complexity. *Journal of Computer and System Sciences*, 31(2):169–181, 1985.