Training Sequences

by

Dana Angluin[†]
Department of Computer Science
Yale University
New Haven, CT 06520

and

William I. Gasarch
Department of Computer Science and
Institute for Advanced Computer Studies
The University of Maryland
College Park, MD 20742

and

Carl H. Smith[‡]
Department of Computer Science and
Institute for Advanced Computer Studies
The University of Maryland
College Park, MD 20742

I. Introduction

Computer scientists have become interested in inductive inference as a form of machine learning primarily because of artificial intelligence considerations, see [2,3] and the references therein. Some of the vast body of work in inductive inference by theoretical computer scientists [1,4,5,6,10,12,22,25,28,29] has attracted the attention of linguists (see [20] and the references therein) and has had ramifications for program testing [7,8,27].

[†] Supported in part by NSF Grant IRI 8404226.

[‡] Supported in part by NSA OCREAE Grant MDA904-85-H-0002. Currently on leave at the National Science Foundation.

To date, most (if not all) the theoretical research in machine learning has focused on machines that have no access to their history of prior learning efforts, successful and/or unsuccessful. Minicozzi [19] developed the theory of reliable identification to study the combination and transformation of learning strategies, but there is no explicit model of an agent performing these operations in her theory. Other than that brief motivation for reliable inference there has been no theoretical work concerning the idea of "learning how to learn." Common experience indicates the people get better at learning with practice. That learning is something that can be learned by algorithms is argued in [13].

The concept of "chunking" [18] has been used in the *Soar* computer learning system in such a way that chunks formed in one learning task can be retained by the program for use in some future tasks [15,16]. While the *Soar* system demonstrates that it is possible to use knowledge gained in one learning effort in a subsequent inference, this paper initiates a study in which it is demonstrated that certain concepts (represented by functions) can be learned, but *only* in the event that certain relevant subconcepts (also represented by functions) have been previously learned. In other words, the *Soar* project presents empirical evidence that learning how to learn is viable for computers and this paper proves that doing so is the only way possible for computers to make certain inferences.

We consider algorithmic devices called *inductive inference machines* (abbreviated: IIMs) that take as input the graph of a recursive function and produce programs as output. The programs are assumed to come from some acceptable programming system [17,23]. Consequently, the natural numbers will serve as program names. Program i is said to compute the function φ_i. M *identifies* (or explains) f iff when M is fed longer and longer initial segments of f it outputs programs which, past some point, are all i, where $\varphi_i = f$. The notion of identification (originally called "identification in the limit") was introduced formally by Gold [12] and presented recursion theoretically in [5]. If M does identify f we write $f \in \mathrm{EX}(M)$. The "EX" is short for "explains," a term which is consistent with the philosophical motivations for research in inductive inference [6]. The collection of inferrible

sets is denoted by EX, in symbols EX = $\{S \mid (\exists M)[S \subseteq \mathrm{EX}(M)]\}$. Several other variations of EX inference have been investigated [2].

The new notion of inference needed to show that, in some sense, machines can learn how to learn is one of inferring *sequences* of functions. Suppose that $\langle f_1, f_2, \ldots, f_n \rangle$ is a sequence of functions and M is an IIM. M can infer $\langle f_1, f_2, \ldots, f_n \rangle$ (written: $\langle f_1, f_2, \ldots, f_n \rangle \in \mathrm{S}^n\mathrm{EX}(M)$) iff

1. M can identify f_1 from the graph of f_1, with no information and
2. for $0 < i < n$, M can identify f_{i+1} from the graph of f_{i+1} if it is also provided with a sequence of programs e_1, e_2, ..., e_i, such that $\phi_{e_1} = f_1$, ..., $\varphi_{e_i} = f_i$.

$\mathrm{S}^n\mathrm{EX} = \{S \mid (\exists M)[S \subseteq \mathrm{S}^n\mathrm{EX}(M)]\}$.

A more formal definition appears in the next section. One scenario for conceptualizing how an IIM M can S^nEX infer some sequence like $\langle f_1, f_2, \ldots, f_n \rangle$ is as follows. Suppose that M *simultaneously* receives, on separate input channels, the graphs of $f_1, f_2, \ldots, f_n$. M is then free to use its most current conjectures for $f_1, f_2, \ldots, f_i$ in its calculation of a new hypothesis for f_{i+1}. If M changes its conjecture as to a program for f_i, then it also outputs new conjectures for $f_{i+1}, \ldots, f_n$. If f_{i+1} really somehow depends on $f_1, f_2, \ldots, f_i$, then no inference machine should be able to infer f_{i+1} without first learning $f_1, f_2, \ldots, f_i$. The situation where an inference machine is attempting to learn each of $f_1, f_2, \ldots, f_i$ simultaneously is discussed in the section on parallel learning below.

Another scenario is to have a "trainer" give an IIM M some programs as a preamble to the graph of some function. Our results on learning sequences of functions by single IIMs and teams of IIMs use this approach. In this case there is no guarantee that M has learned how to learn based on its own learning experiences. However, if the preamble is supplied by using the output of some other IIM, then perhaps M is learning based on some other machine's experience. If we restrict ourselves to a single machine and rule out magic, then there is no other possible source for the preamble of programs, other than what has been produced by M during previous learning efforts. In this case, M is assuming that

its preamble of programs is correct. The only way for M to know for certain that the programs it is initially given compute the functions it previously tried to learn is for M to be so told by some trainer.

Two slightly different models are considered below, one for each of the above scenarios. A rigorous comparison of the two notions reveals that the parallel learning notion is more powerful than learning with a training sequence.

For all n, $\mathrm{S}^n\mathrm{EX}$ is nonempty (not necessarily a profound remark). Consider any IIM M for which $\mathrm{EX}(M)$ is not empty. Let $S = \mathrm{EX}(M)$. Then $S \times S \in \mathrm{S}^2\mathrm{EX}$, $S \times S \times S \in \mathrm{S}^3\mathrm{EX}$, etc. The witness is an IIM M' that ignores the preamble of programs and simulates M. These are not particularly interesting members of $\mathrm{S}^n\mathrm{EX}$ since it is not necessary to learn a program for the first function in each sequence in order to learn a program for the second function, etc. One of our results is the construction of an interesting member of $\mathrm{S}^n\mathrm{EX}$.

We construct an $S \in \mathrm{S}^n\mathrm{EX}$, uniformly in n, containing only n-tuples of functions $\langle f_1, f_2, \ldots, f_n \rangle$ such that for each IIM M there is an $\langle f_1, f_2, \ldots, f_n \rangle \in S$ such that, for $1 \leq i \leq n$, M cannot infer f_i if it is not provided with a preamble of programs that contains programs for each of $f_1, f_2, \ldots, f_{i-1}$.

Let $S \in \mathrm{S}^n\mathrm{EX}$ be a set of n-tuples of functions. Suppose $\langle f_1, f_2, \ldots, f_n \rangle \in S$. $f_1, f_2, \ldots, f_{n-1}$ are the "subconcepts" that are needed to learn f_n. In a literal sense, $f_1, f_2, \ldots, f_{n-1}$ are encoded into f_n. The encoding is such that $f_1, f_2, \ldots, f_{n-1}$ can *not* be extracted from the graph of f_n. (If $f_1, f_2, \ldots, f_{n-1}$ could be extracted from f_n then an inference machine could recover programs for $f_1, f_2, \ldots, f_{n-1}$ and infer f_n without any preamble of programs, contradicting our theorem.) The constructed set S contains sequences of functions that must be learned in the presented order, otherwise there is no IIM that can learn all the sequences in S. Here $f_1, f_2, \ldots, f_{i-1}$ is the "training sequence" for f_i, motivating the title for this paper.

II. Definitions, Notation, Conventions and Examples

In this section we formally define concepts that will be of use in this paper. Most of our definitions are standard and can be found in [6]. Assume throughout that $\varphi_0, \varphi_1, \varphi_2, \ldots$ is

a fixed acceptable programming system of all (and only all) the partial recursive functions [17,23]. If f is a partial recursive function and e is such that $\varphi_e = f$ then e is called a *program* for f. $\mathbb{N}$ denotes the natural numbers, which include 0. $\mathbb{N}^+$ denotes the natural numbers without 0. Let $\langle \cdot, \cdot, \ldots, \cdot \rangle$ be a recursive bijection from $\bigcup_{i=0}^{\infty} \mathbb{N}^i$ to $\mathbb{N}$. We will assume that the empty sequence maps to 0.

Definition: Let f be a recursive function. An IIM M *converges* on input f to program i (written: $M(f)\downarrow = i$) iff almost all the elements of the sequence $M(\langle f(0) \rangle)$, $M(\langle f(0), f(1) \rangle)$, $M(\langle f(0), f(1), f(2) \rangle), \ldots$ are equal to i.

Definition: A set S of recursive functions is *learnable* (or *inferrible* or EX-identifiable) if there exists an IIM M such that for any $f \in S$, $M(f)\downarrow = i$ for some i such that $\varphi_i = f$. EX is the set of all subsets S of recursive functions that are learnable.

In the above we have assumed that each inference machine is viewing the input function in the natural, domain increasing order. Since we are concerned with total functions, we have not lost any of the generality that comes with considering arbitrarily ordered enumerations of the graphs of functions as input to IIM's. An order independence result that covers the case of inferring partial (not necessarily total) recursive functions can be found in [5]. The order that IIM sees its input can have dramatic effects on the complexity of performing the inference [9] but not on what can and cannot be inferred.

We need a way to talk about a machine learning a sequence of functions. Once the machine knows the first few elements of the sequence then it should be able to infer the next element. We would like to say that if the machine "knows" programs for the previous functions then it can infer the next function. In the next definition we allow the machine to know a subset of the programs for previous functions.

Definition: $M(\langle e_1 \ldots, e_m \rangle, f)\downarrow = e$ means that the sequence of outputs produced by M when given programs $e_1, \ldots, e_m$ and the graph of f converges to program e.

Definition: Let $n > 1$ be any natural number. Let $J = \langle J_1, \ldots, J_{n-1} \rangle$, where J_i $(1 \leq i \leq n-1)$ is a subset of $\{1, 2, \ldots, i-1\}$. (J_1 will always be $\emptyset$.) Let $J_i = \{b_{i1}, b_{i2}, \ldots, b_{im}\}$. A set S of sequences of n-tuples of recursive functions is *J-learnable* (or J-inferrible, or J-S^{n}EX-identifiable) if there exists an IIM M such that for all $\langle f_1, \ldots, f_n \rangle \in S$, for all $\langle e_1, \ldots, e_n \rangle$ such that e_j is a program for f_j $(1 \leq j < n)$, for all i $(1 \leq i \leq n)$, $M(\langle e_{b_{i1}}, e_{b_{i2}}, e_{b_{i3}}, \ldots, e_{b_{im}} \rangle, f_i)\downarrow = e$ where e is a program for f_i.

Note that f_1 has to be inferrible. Intuitively, if the machine knows programs for a subset of functions (specified by J_i) before f_i, then the machine can infer f_i. M is called a *Sequence IIM* (SIIM) for S. S^{n}EX is the set of n-tuples of recursive functions that are J-learnable with $J = \langle J_1, \ldots, J_{n-1} \rangle$, $J_i = \{1, 2, \ldots, i-1\}$ $(1 \leq i \leq n)$, i.e. the set of sequences such that any function in the sequence can be learned if *all* the previous ones have already been learned.

Convention: If an SIIM machine is not given any programs, but is given σ (σ is a subset of the input function) then we use the notation $M(\perp, \sigma)$. If an SIIM machine is given one program, e, and is given σ then we use the notation $M(e, \sigma)$ instead of the (formally correct) $M(\langle e \rangle, \sigma)$.

We are interested in the case where inferring the i^{th} function of a sequence *requires* knowing all the previous ones and some nonempty portion of the graph of the i^{th} function. The notion that is used in our proofs is the following.

Definition: A set S of sequences of n-tuples of recursive functions is *redundant* if there is an SIIM that can infer all f_n with a preamble of fewer than $n-1$ programs for f_1, f_2, ..., f_{n-1}. Every set S is either nonredundant or redundant.

Example: A set in S^3EX which is redundant.

$$S = \{\langle f_1, f_2, f_3\rangle \mid f_1(0) \text{ is a program for } f_1,$$
$$f_2(2x) = f_1(x)(\text{for } x \neq 0),$$
$$f_2(2x+1) \text{ is } 0 \text{ almost everywhere },$$
$$f_3(2x) = f_1(2x) + f_2(2x+1),$$
$$f_3(2x+1) = \ 0 \text{ almost everywhere, and}$$
$$f_1, f_2, f_3 \text{ are all recursive } \}$$

To infer f_2 a machine appears to need to know a program for f_1; to infer f_3 a machine appears to only need a program for f_1. Formally the set S is $\langle \emptyset, \{1\}, \{1\}\rangle$-learnable. Examples of nonredundant sets are more difficult to construct. In sections III and IV examples of nonredundant sets will be constructed.

The notion of nonredundancy that we are really interested in is slightly stronger. The definition is given below. It turns out to be easy to pass from the technically tractable definition to the intuitively interesting one.

Definition: A set S of sequences of n-tuples of recursive functions is *strictly redundant* if it is J-learnable for $J = \langle J_1, \ldots, J_{n-1}\rangle$ where there exists an i such that J_i is a proper subset of $\{1, 2, \ldots, i-1\}$.

The following technical lemma shows that the existence of certain nonredundant sets implies the existence of a strictly nonredundant set. This means that we can prove our theorems using the weaker, technically tractable definition of nonredundancy and our results will also hold for the more interesting notion of strict nonredundancy.

LEMMA 1. If there exists sets S_i $(2 \leq i \leq n)$ of nonredundant i-tuples of functions, then there exists a set S of n-sequences that is strictly nonredundant.

Proof:

Take S to be

$$\bigcup_{i=1}^{n}\{\langle f_1,\ldots,f_n\rangle \mid \exists\langle g_1,\ldots,g_i\rangle \in S_i$$
$$f_j(x) = \langle j, g_j(x)\rangle \qquad (1 \le j \le i)$$
$$f_j(x) = 0 \qquad (i+1 \le j \le n)\}$$

Suppose by way of contradiction that S is *not* strictly nonredundant. Then there exists i, J and M such that $J \subset \{1,\ldots,i-1\}$ and M can infer f_i from the indices of f_j, for $j \in J$, and the graph of f_i. The machine M can easily be modified to infer g_i from the indices of g_j, for $j \in J$, and the graph of g_i. Since J is a proper subset of $\{1,\ldots,i-1\}$, this contradicts the hypothesis. ☒

The following definitions are motivated by our proof techniques.

Definition: Suppose f is a recursive function and $n \in \mathbb{N}$. For $j < n$, the j^{th} n-ply of f is the recursive function $\lambda x[f(n \cdot x + j)]$.

n-plies of partial recursive functions were used in [25]. Clearly, any recursive function can be constructed from its n-plies. For the special case of $n = 2$ we will refer to the even and odd plies of a given function.

Often, we will put programs for constant functions along one of the plies of some function that we are constructing. For convenience, we let c_i denote the constant i function, e.g. $\lambda x[i]$. Also, p_i denotes a program computing c_i, e.g. $\varphi_{p_i} = c_i$.

As a consequence of the above lemma, we will state and prove our results in terms of redundancy with the implicit awareness that the results also apply with "redundancy" replaced by "strict redundancy" everywhere. This slight of notation allows us to omit what would otherwise be ubiquitous references to Lemma 1.

III. Learning Pairs of Functions

In this section we prove that there is a set of pairs of functions that can be learned sequentially by a single IIM but cannot be learned independently by any IIM. The technique used in the proof is generalized in the next section.

THEOREM 2. $S = \{\langle c_i, \varphi_i\rangle \mid \varphi_i \text{ is a recursive function}\}$ is a nonredundant member of $\mathrm{S^2EX}$.

Proof: First, we give the algorithm for an SIIM M which witnesses that $S \in \mathrm{S^2EX}$. M will view two different types of input sequences: one with an empty preamble of programs and one with a single program in the preamble. On input sequences with an empty preamble, M reads an input pair (x, y) and outputs a program for c_y and stops. Suppose M is given an input sequence with a preamble consisting of "i." Before reading any input other than the preamble, M evaluates $\varphi_i(0)$, outputs the answer (when and if the computation converges) and stops. Suppose $\langle f_0, f_1\rangle \in S$. Then f_0 is a constant function and will be inferred by M from its graph. Suppose $f_0 = \lambda x[e]$. By membership in S, $\varphi_e = f_1$. Hence, M will infer f_1, given a program for f_0.

To complete the proof we must show that S is not redundant. Suppose by way of contradiction that S is redundant. Then there is an IIM that can infer $R = \{f \mid \exists g$ such that $\langle g, f\rangle \in S\}$. Note that R is precisely the set of recursive functions, which is known to be not inferrible [12]. Hence, no such IIM can exist. ⊠

Note that the SIIM M defined above outputs a *single* program, independent of its input. For a discussion of inference machines and the number of conjectures they make, see [6]. We could modify the SIIM M above to make it *reliable* on the recursive functions in the sense that it will not converge unless it is identifying [5,19]. The notion of reliability used here is as follows: A SIIM M reliably identifies S if and only if for all $k > 0$, whenever $\langle e_1, \ldots, e_k\rangle$ is such that for some $\langle f_1, \ldots, f_n\rangle \in S$, $\varphi_{e_i} = f_i$ for $i = 1, \ldots, k$, and g is any recursive function, then $M(\langle e_1, \ldots, e_k\rangle, g)$ converges to a program j iff $\varphi_j = g$.

The modification to make M of the previous theorem reliable is as follows. After M outputs its only program, it continues (or starts) reading the graph of the input function looking for a counterexample to its conjecture. If M is given an empty preamble, the program produced as output computes a constant function, which is recursive. If M is given a nonempty preamble then, M assumes the program in the preamble computes some

constant function $\lambda x[i]$ where φ_i is a recursive function. Hence, the modified M will always be comparing its input with a program computing a recursive function. If a counterexample is found, M proceeds to diverge by outputting the time of day every five minutes.

A stronger notion of reliability would be to require that M converge correctly whenever its preamble contains only programs for recursive functions and the function whose graph is used as input is also recursive. Run time functions can be used to derive the same result for the stronger notion of reliability.

IV. Learning Sequences of Functions

In this section we will generalize the proof of the previous section to cover sequences of an arbitrary length. We start be defining an appropriate set of n-tuples of recursive functions. Intuitively, all but the last program in the sequence computes a constant function where the constant computed is a program for one of the plies of the last function in the sequence. Suppose $n \in \mathbb{N}^+$. Then

$$S_{n+1} = \{\langle f_0, f_1, \ldots, f_n\rangle \mid f_n \text{ is any recursive function and for each } i < n,\ f_i \text{ is the constant } j_i \text{ function where } \varphi_{j_i} \text{ is the } i^{\text{th}} n\text{-ply of } f_n\}$$

THEOREM 3. For all $n > 0$, S_n is a nonredundant member of $\mathrm{S}^n\mathrm{EX}$.

Proof: First we will show that there is an SIIM M_{n+1} such that if $\langle f_0, f_1, \ldots, f_n\rangle \in S_{n+1}$ and $i_0, \ldots, i_{n-1}$ are programs for $f_0, \ldots, f_{n-1}$, then $M_{n+1}(\langle i_0, \ldots, i_{n-1}\rangle, f_n)$ converges to a program for f_n. M_{n+1} first reads the preamble of programs $i_0, \ldots, i_{n-1}$ and runs $\varphi_{i_j}(0)$ to get a value e_j for each $j < n$. M_{n+1} then outputs a program for the following algorithm:

> On input x, calculate i such that $i \equiv x \bmod n$ and let $x' = (x - i)/n$. Output the value $\varphi_{e_i}(x')$.

If $i_0, \ldots, i_{n-1}$ are indeed programs for $f_0, \ldots, f_{n-1}$ then M_{n+1} will output a program for f_n. As in the previous proof, we could make M_{n+1} reliable on the recursive functions.

Let $J = \{i_1, \ldots, i_r\}$ be any proper subset of $\{0, \ldots, n-1\}$. Suppose by way of contradiction that there is an SIIM M such that whenever $\langle f_0, f_1, \ldots, f_n \rangle \in S_{n+1}$ and $e_{i_1}, \ldots, e_{i_r}$ are programs for $f_{i_1}, \ldots, f_{i_r}$ then $M(\langle e_{i_1}, \ldots, e_{i_r} \rangle, f_n)$ converges to a program for f_n. We complete the proof by showing how to transform M into M', an IIM that is capable of inferring all the recursive functions, a contradiction. Choose $j \in \{0, 1, \ldots, n-1\} - J$. Suppose the graph of f, a recursive function, is given to M' as input. Assume without loss of generality that the input is received in its natural domain increasing order $(0, f(0)), (1, f(1)), \ldots$. ¿From the values of f received as input it is possible to produce, again in domain increasing order, the graph of the following recursive function g:

$$g(x) = \begin{cases} f(i) & \text{if } x = ni + j; \\ 0 & \text{if } x \not\equiv j \bmod n. \end{cases}$$

Notice that the j^{th} n-ply of g is f and all the other n-plies of g are equal to $\lambda x[0]$. Let z be a program for the everywhere zero function ($\lambda x[0]$). M' now simulates M feeding M the input sequence:

$$\underbrace{\langle z, z, \ldots, z \rangle}_{r \text{ copies}}, g(0), g(1), \ldots$$

Whenever M outputs a conjectured program k, M' outputs a program $s(k)$ such that $\varphi_{s(k)} = \lambda x[\varphi_k(nx + j)]$. $s(k)$ is a program for the j^{th} n-ply of φ_k.

In summary, M' takes its input function and builds another function with the given input on the j^{th} n-ply and zeros everywhere else. M' then feeds this new function, with a preamble of r programs for the constant zero function, to M, which supposedly doesn't need the j^{th} n-ply. When M returns the supposedly correct program, M' builds a program that extracts the j^{th} n-ply. By our suppositions about the integrity of M, this program output by M' correctly computes f, its original input function. Since f was chosen to be an arbitrary recursive function, M' can identify all the recursive functions in this manner, a contradiction. ☒

The above proof is a kind of reduction argument. We know of no other use of reduction techniques in the theory of inductive inference. A set was constructed such that its redundancy would imply a contradiction to a known result. An alternate proof, using a direct construction, was discovered earlier by the authors [11]. The direct proof of the above theorem is more constructive but considerably more complex. The proof given above has the additional advantage of being easier to generalize.

V. Team Learning

Situations in which more than one IIM is attempting to learn the same input function were considered in [25]. In general, the learnable sets of functions are not closed under union [5]. For team learning, the team is successful if one of the members can learn the input function. The power of the team comes from its diversity as some IIMs learn some functions and others learn different functions, but when considered as a team, the team can learn any function that can be learned by any team member. This notion of team learning was shown to be precisely the same as probabilistic learning [21]. The major results from [25] and [21] are summarized, unified and extended in [22].

In some cases, teams of SIIMs can be used to infer nonredundant sets of functions from less information than a single SIIM requires. For example, consider the set S^3 from Theorem 3. Suppose $\langle c_i, c_j, f\rangle \in S^3$. In this case, the even ply of f is just φ_i and the odd ply is φ_j. Let M_1 be a SIIM that receives program p_i (computing c_i) prior to receiving the graph of f and M_2 is a similar SIIM that has p_j as its preamble. Each of these two SIIMs then uses its preamble program as an upper bound for the search for a program to compute the even ply of f and simultaneously as an upper bound for the search for a program to compute the odd ply of f. Since natural numbers name all the programs, one of the two preambles must contain a program (natural number) that bounds both p_i and p_j. The SIIM that receives the preamble with the larger (numerically) program will succeed in its search for a program for both the even and odd plies of f. Hence, the team of two SIIMs just described can infer, from a preamble containing a single program, all of

S^3. A stronger notion of nonredundancy is needed to discuss the relative power of teams of SIIMs.

In this section, for each $n > 1$, a nonredundant $\hat{S}_n \in \mathrm{S}^n\mathrm{EX}$ will be constructed with the added property that $\{f_n \mid \langle f_1, \ldots, f_n\rangle \in \hat{S}_n\}$ is not inferrible by any team of $n-1$ SIIM's that see a preamble of at most $n-2$ programs. This appears to be a stronger condition than nonredundancy, and, in fact, we prove this below. Not only can't $\hat{S}_n$ be inferred by any SIIM that sees fewer than $n-1$ programs in its preamble, it can't be inferred by any size $n-1$ team of such machines. Such sets $\hat{S}_n$ are called *super nonredundant*.

The fully general result involves some combinatorics that obscure the main idea of the proof. Consequently, we will present the $n = 3$ case first. We make use of the sets T_m constructed in [25] such that T_m is inferrible by a team of m IIMs but by no smaller team.

THEOREM 4. There is a set $\hat{S}_3 \in \mathrm{S}^3\mathrm{EX}$ that is super nonredundant.

Proof: Let $M_1, \ldots, M_6$ be the IIMs that can team identify T_6. Fix some coding C from $\{1,2\} \times \{1,2,3\}$ 1-1 and onto $\{1, \ldots, 6\}$. We can now define $\hat{S}_3$.

$$\hat{S}_3 = \{\langle f_1, f_2, f_3\rangle \mid f_1 \in \{c_1, c_2\},\ f_2 \in \{c_1, c_2, c_3\},\ \text{and } f_3 \in T_6 \text{ where } C(f_1(0), f_2(0)) \text{ is the least index of an IIM in } M_1, \ldots, M_6 \text{ that can infer } f_3\}$$

It is easy to see that $\hat{S}_3 \in \mathrm{S}^3\mathrm{EX}$. The first two functions in the sequence are always constant functions which are easy to infer. Given programs for f_1 and f_2 the SIIM figures out what constants these functions are computing and then uses the coding C to figure out which one of $M_1, \ldots, M_6$ to simulate.

Suppose that $\langle f_1, f_2, f_3\rangle \in \hat{S}_3$, e_1 a program for f_1, and e_2 a program for f_2. Suppose by way of contradiction that M_1' and M_2' are SIIMs and either $M_1'(e_1, f_3)$ or $M_2'(e_2, f_3)$ identifies f_3. The case where both M_1' and M_2' both see e_1 (or e_2) is similar. Let $[M, e]$ denote the IIM formed by taking an SIIM M and hard wiring its premable of programs to be "e". Recall that program p_i computes the constant i function c_i, for each i. One of the five machines $[M_1', p_1]$, $[M_1', p_2]$, $[M_2', p_1]$, $[M_2', p_2]$, or $[M_2', p_3]$ will infer each f_3 such that $\langle f_1, f_2, f_3\rangle \in \hat{S}_3$. This set is precisely T_6, contradicting the choice of T_6. ☒

THEOREM 5. For each $n \in \mathbb{N}$, there is a set $\hat{S}_n \in \mathrm{S}^n\mathrm{EX}$ that is super nonredundant.

Proof: For $n \leq 2$ the theorem holds vacuously. Choose $n > 2$. Let $g_i = 2^i$, for all i. Let P be the product of $g_1, g_2, \ldots, g_{n-1}$. Let C be a fixed coding from $\{1, \ldots, g_1\} \times \cdots \times \{1, \ldots, g_{n-1}\}$ 1-1 and onto $\{1, \ldots, P\}$. Let $M_1, \ldots, M_P$ be the IIMs that can team identify T_P, the set of recursive functions that is not identifiable by any team of size $P-1$. Now we can define $\hat{S}_n$.

$$\begin{aligned}\hat{S}_n = \{\langle f_1, \ldots, f_n\rangle \,|\, & f_j \in \{c_1, \ldots, c_{g_j}\}, \text{ for } 1 \leq j < n \text{ and } f_n \in T_P \\ & \text{where } C(f_1(0), \ldots, f_{n-1}(0)) \text{ is the least index of an IIM in} \\ & M_1, \ldots, M_P \text{ that can infer } f_n\}\end{aligned}$$

It is easy to see that $\hat{S}_n \in \mathrm{S}^n\mathrm{EX}$. The first $n-1$ functions in the sequence are always constant functions which are easy to infer. Given programs for $f_1, \ldots, f_{n-1}$ the SIIM figures out what constants these functions are computing and then uses the coding C to figure out which one of $M_1, \ldots, M_P$ to simulate.

Suppose $\langle f_1, \ldots, f_n\rangle \in \hat{S}_n$ and $e_1, \ldots, e_{n-1}$ are programs for $f_1, \ldots, f_{n-1}$, respectively. Suppose by way of contradiction that $M'_1, \ldots, M'_{n-1}$ are SIIMs such that if M'_j $(0 < j < n)$ is given the preamble of programs $e_1, \ldots, e_{n-1}$, except for program e_j, and the graph of f_n, then one of $M'_1, \ldots, M'_{n-1}$ will identify f_n. Actually, we need to suppose that the team $M'_1, \ldots, M'_{n-1}$ behaves this way on any n-tuple of functions in $\hat{S}_n$. This way we are considering the most optimistic choice for a collection of $n-1$ SIIMs. Any other association of machines to indices is similar.

As with the $n = 3$ case, we proceed by hard wiring various preambles of programs into the SIIMs M' to form a team of IIMs that can infer T_P. As long as the size of this team is strictly less than P, we will have a contradiction to the team hierarchy theorem of [25]. The remainder of this proof is a combinatorial argument showing that P was indeed chosen large enough to bound the number IIMs that could possibly arise by hard wiring in a preamble of $n-2$ programs into one of the M''s.

Since M'_j sees $e_1, \ldots, e_{j-1}, e_{j+1}, \ldots, e_{n-1}$ and there are g_i choices for e_i there are P/g_j different ways to hard wire in programs for relevant constant functions into M'_j. Hence, the total number of IIM's needed to form a team capable of inferring every f_n in $\hat{S}_n$ is:

$$\sum_{i=1}^{n-1} \frac{P}{g_i}.$$

The size of this team will be strictly bounded by P as long as:

$$\sum_{i=1}^{n-1} \frac{1}{g_i} < 1.$$

This inequality follows immediately from the definition of the g_i's. Hence, the theorem follows. ⊠

Note that the formula in the general case suggests using the set T_8 as a counterexample for the $n = 3$ case. In Theorem 4, the set T_6 was used. What this means is that the choice of the constants $g_1, g_2, \ldots$ was not optimal. We leave open the problem of finding the smallest possible values of the constants that suffices to prove the above result.

VI. Parallel Learning

In previous sections we examined the problem of inferring sequences of functions by SIIMs and teams of SIIMs. In this section, we show that there are sets of functions that are not inferrible individually, but can be learned when simultaneously presenting to a suitable IIM. First, we define identification by a *Parallel* IIM.

Definition: An *n-PIIM* is an inference machine that simultaneously (or by dovetailing) inputs the graphs of an n-tuple of functions $\langle f_1, f_2, \ldots, f_n \rangle$ and from time to time, outputs n-tuples of programs. An n-PIIM M converges on input from $\langle f_1, f_2, \ldots, f_n \rangle$ to $\langle e_1, e_2, \ldots, e_n \rangle$ if at some point while simultaneously inputting the graphs of $f_1, f_2, \ldots, f_n$, M outputs $\langle e_1, e_2, \ldots, e_n \rangle$ and never later outputs a different n-tuple of programs. An n-PIIM M *identifies* $\langle f_1, f_2, \ldots, f_n \rangle$ iff M on input $\langle f_1, f_2, \ldots, f_n \rangle$ converges to $\langle e_1, e_2, \ldots, e_n \rangle$

and $\varphi_{e_i} = f_i$ for all $1 \leq i \leq n$. $\mathrm{P}^n\mathrm{EX} = \{\langle f_1, f_2, \ldots, f_n\rangle \mid \exists M$ an n-PIIM such that M identifies $\langle f_1, f_2, \ldots, f_n\rangle\}$.

Notice that $\mathrm{P}^1\mathrm{EX} = \mathrm{EX}$. In order to somehow compare the classes $\mathrm{P}^n\mathrm{EX}$ as n varies, we need a way of compressing $\mathrm{P}^n\mathrm{EX}$ into $\mathrm{P}^m\mathrm{EX}$ for $m < n$. This will be accomplished via an *m-projection*. An m-projection of $\langle f_1, f_2, \ldots, f_n\rangle$ is given by an m-tuple $1 \leq i_1 < i_2 < \ldots < i_m \leq n$ such that the determined m-projection is $\langle f_{i_1}, f_{i_2}, \ldots, f_{i_m}\rangle$. Let $\binom{n}{m}$ denote "n choose m" $(n!/m!(m-n)!)$. For a given m and n with $m < n$ there are $\binom{n}{m}$ different m-projections possible. An m-projection of a set of n-tuples of recursive functions is the set of m-projections of all the tuples. The general theorem that we will prove below asserts that for every $m < n$ there is a set of n-tuples of recursive functions such that every m-projection of that set is in $\mathrm{P}^m\mathrm{EX}$ but no $(m-1)$-projection is in $\mathrm{P}^{m-1}\mathrm{EX}$. To illustrate the basic proof technique, without the combinatorics necessary for the general case, we prove the following special case.

A recursive function that is useful in the following proofs is one that computes two functions simultaneously. Define *ply* to be a program such that for all programs i and j:

$$\varphi_{ply(i,j)}(x) = \begin{cases} \varphi_i(x/2) & \text{if } x \text{ is even} \\ \varphi_j((x-1)/2) & \text{if } x \text{ is odd.} \end{cases}$$

So $\mathrm{ply}(i,j)$ is a program that computes φ_i on its even ply and φ_j on its odd ply.

THEOREM 6. There is a set $S \in \mathrm{P}^2\mathrm{EX}$ such that neither 1-projection of S is in EX.

Proof: First we define S.

$$S = \{\langle f_1, f_2\rangle \mid \text{ the even ply of } f_1 \text{ is any recursive function and the odd ply of } f_1 \text{ is the constant } e_2 \text{ function where } \varphi_{e_2} \text{ is the even ply of } f_2, \text{ and the even ply of } f_2 \text{ is any recursive function and the odd ply of } f_2 \text{ is the constant } e_1 \text{ function where } \varphi_{e_1} \text{ is the even ply of } f_1\}$$

The 2-PIIM M witnessing $S \in \mathrm{P}^2\mathrm{EX}$ is described as follows. M inputs values from f_1 and f_2 until it has received $f_1(1)$ and $f_2(1)$. Let $j = f_1(1)$ and $k = f_2(1)$. M then outputs $\langle ply(k, p_j), ply(j, p_k)\rangle$ and converges. Clearly, M suffices.

Suppose by way of contradiction that M is a 1-PIIM (an IIM) that can identify $S' = \{f \mid \exists g \text{ such that } \langle f, g\rangle \in S\}$. The case of the other 1-projection of S is similar. Suppose φ_i is an arbitrary recursive function. Let $k = ply(i, p_{p_0})$ and $j = ply(p_0, p_i)$. Then $\langle \varphi_k, \varphi_j\rangle \in S$ and $\varphi_k \in S'$. So every recursive function is the even ply of some member of S'. We now construct M' an IIM that can identify all the recursive functions, a contradiction. On input (x_0, y_0), $(x_1, y_1), \ldots$ M' simulates M with input $(2 \cdot x_0, y_0)$, $(1, p_0)$, $(2 \cdot x_1, y_1)$, $(3, p_0)$, $\cdots$. In other words, M' takes an arbitrary recursive function as input and transforms it into a member of S' that M can identify. If M outputs p, then M' outputs a program for the even ply of φ_p. M' then infers all the recursive functions. ☒

The general result is proven using a set of n-tuples of recursive functions whose even plies are arbitrary recursive functions and whose odd plies encode some information about the other functions in the n-tuple. Some combinatorial difficulty arises because complete information about the other functions in the n-tuple must be divided into enough pieces and distributed. This distribution will take place along the k-plies of the odd plies of the functions in the n-tuple, for some k. Some more notation is needed to conveniently describe the encoding. Let $f_1, f_2, \ldots, f_n$ be an n-tuple of functions. For $0 < i \leq n$ and $j < k$, let $PR^k_j(f_i)$ denote a program that computes the j^{th} k-ply of f_i. $PR^k_j(\langle f_1, f_2, \ldots, f_n\rangle)$ denotes the n-tuple of programs where each program computes the j^{th} k-ply of the corresponding f.

THEOREM 7. Suppose $0 < m < n$. Then there is a set of n-tuples of recursive functions such that every m-projection of that set is in $\mathrm{P}^m\mathrm{EX}$ but no $(m-1)$-projection is in $\mathrm{P}^{m-1}\mathrm{EX}$.

Proof: Choose $k = \binom{n}{m-1}$. Let $C_0, C_1, \ldots, C_{k-1}$ be all the size $m-1$ subsets of $\{1, 2, \ldots, n\}$. Now, we can define S, the desired set of n-tuples of recursive functions.

$$S = \{\langle g_1, \ldots, g_n\rangle \mid \exists \text{ recursive functions } f_1, f_2, \ldots, f_n \text{ such that for each } i \ (1 \leq i \leq n) \text{ the even ply of } g_i \text{ is } f_i \text{ and the odd ply is a constant function for some constant encoding the values } PR_j^k(\langle f_1, f_2, \ldots, f_n\rangle) \text{ for all } j \text{ such that } i \notin C_j\}.$$

Notice that if some PIIM receives the graph of g_{i_1} then it will have information about all the j^{th} k-plies of each of the f's for each j such that C_j does not contain i_1. Similarly, if this PIIM simultaneously receives the graphs of g_{i_1} and g_{i_2} then it will have information about all the j^{th} k-plies of each of the f's for each j such that C_j does not contain both i_1 and i_2. Consequently, if some PIIM simultaneously receives the graphs of g_{i_1}, g_{i_2}, ..., g_{i_m} then it will have information about all the j^{th} k-ply of each of the f's for each j such that C_j does not contain each of i_1, i_2, ..., i_m.

Since each of the C_j's has cardinality exactly $m-1$, *no* C_j contains each of i_1, i_2, ..., i_m. Hence, a PIIM receiving the graphs of g_{i_1}, g_{i_2}, ..., g_{i_m} will be able to recover programs for each of the k-plies of each of the f's. ¿From the k-plies of the f's, not only can programs for the f's be constructed (the even ply of the g's), but the encodings of $PR_j^k(\langle f_1, f_2, \ldots, f_n\rangle)$ for all subsets of j's from $\{0, \ldots, k-1\}$ as well. This latter information is all that is needed to figure out the constants that go on the odd ply of the g's. Hence, a program for each of the g's is constructible via the *ply* function. We have just informally described a m-PIIM that can infer any m-projection of S. Furthermore, this PIIM can actually infer the n-tuples of functions in S from any m-projection of S.

Suppose by way of contradiction that i_1, i_2, ..., i_{m-1} is an m-projection of S that is in $\mathrm{P}^{m-1}\mathrm{EX}$. Let M be the witnessing PIIM. Let j be such that $C_j = \{i_1, \ldots, i_{m-1}\}$. Then M, after seeing as input $g_{i_1}(1), \ldots, g_{i_{m-1}}(1)$, will know programs for all the k-plies of all the f's except the j^{th} ply. We will now show how to construct an IIM M' that, by simulating M, will be able to infer all the recursive functions.

Let h be an arbitrary recursive function. Let h' be another recursive function that has h as its j^{th} k-ply and has value zero everywhere else. h' can be constructed uniformly and effectively from h. Let $f_1 = h'$, $f_2 = h'$, $\ldots$, $f_n = h'$. For these f's there is a corresponding $\langle g_1, \ldots, g_n \rangle \in S$ such that, for $1 \leq i \leq n$, the even ply of g_i is f_i and suitable constants are on the odd ply. M', when given input from the graph of h constructs the m-projection (given by i_1, i_2, $\ldots$, i_{m-1}) of $g_1, \ldots, g_n$ described above and simulates M on that input. Recall, that M, without direct knowledge of the j^{th} k-ply of its input, can, by assumption, infer each function in the m-projection. If M conjectures a tuple of programs $\langle e_1, \ldots, e_{m-1} \rangle$, then M effectively finds a program e' that computes the even ply of φ_{e_1}. ¿From e', M effectively constructs, and outputs, a program that computes the j^{th} k-ply of $\varphi_{e'}$. By our construction, this program will compute the recursive function h that we started with. Since h was chosen arbitrarily, all the recursive functions can be inferred in this manner, a contradiction. ☒

VII. A Comparison of $\mathrm{S}^n\mathrm{EX}$ and $\mathrm{P}^n\mathrm{EX}$

In this section we show that parallel learning is strictly more powerful than sequence learning. Although this is generally true, our theorems will not hold for the $n = 1$ case since $\mathrm{S}^1\mathrm{EX} = \mathrm{EX} = \mathrm{P}^1\mathrm{EX}$.

THEOREM 8. For all $n \geq 2$, $\mathrm{S}^n\mathrm{EX} \subset \mathrm{P}^n\mathrm{EX}$.

Proof: Suppose $n \geq 2$. First we show inclusion. Suppose M is a SIIM witnessing $S \in \mathrm{S}^n\mathrm{EX}$. Let $\langle f_1, \ldots, f_n \rangle \in S$. We will uniformly and effectively transform M into an n-PIIM M' than simultaneously learns each of $f_1, \ldots, f_n$. To produce a conjecture for f_1, M' simulates $M(\perp, f_1)$ and outputs whatever guesses M outputs. To produce a conjecture for f_2, M' chooses e_1 to its most recent guess as to a program for f_1 and then simulates $M(\langle e_1 \rangle, f_2)$. In general, for $i < n$, M' produces conjectures for f_{i+1} by choosing $e_1, \ldots, e_1$ its most recent conjectures for $f_1, \ldots, f_i$ and then simulating $M(\langle e_1, \ldots, e_i \rangle, f_{i+1})$. Since

M will eventually succeed in inferring f_1, the choice of e_1 will eventually be sound allowing $M(\langle e_1 \rangle, f_2)$ to eventually produce a correct program for f_2. After that point, e_2 will be chosen correctly, enabling the inference of f_3. Continuing this line of argument verifies that M' will simultaneously learn $f_1, \ldots, f_n$. Hence, $S \in \mathrm{P}^n\mathrm{EX}$.

Next we show that the inclusion is proper. By Theorem 7, choose S, a set of n-tuples of functions, such that every n projection of S is in $\mathrm{P}^n\mathrm{EX}$ but no $(n-1)$-projection of S is in $\mathrm{P}^{n-1}\mathrm{EX}$. Let $\langle f_1, \ldots, f_n \rangle \in S$ and S' be the $(n-1)$-projection of S formed by omitting the last function of every n-tuple. For example, $\langle f_1, \ldots, f_{n-1} \rangle$ is a member of S'. By Theorem 7, $S' \notin \mathrm{P}^{n-1}\mathrm{EX}$. By Theorem 8, $S' \notin \mathrm{S}^{n-1}\mathrm{EX}$. If no SIIM can learn the sequence $\langle f_1, \ldots, f_{n-1} \rangle$, then it follows that no SIIM can learn the longer sequence $\langle f_1, \ldots, f_n \rangle$. Thus $S \notin \mathrm{S}^n\mathrm{EX}$. ⊠

Notice that the PIIM M' constructed in the above proof was aware of which function was the first one of the sequence, and which was the second, etc. The above argument breaks down (and indeed the theorem is false) without the assumption that the PIIM is cognizant of the position of each input function in in the original sequence. For example, let Z be the functions of finite support, e.g. the set of functions that map to 0 on all but finitely many arguments and I be the $\{0,1\}$ valued self-describing functions, e.g. the set of functions f such that if n is the least number such that $f(n) \neq 0$ then $\varphi_n = f$. Each of Z and I is in EX, but $Z \cup I$ is not [5].

Suppose by way of contradiction that M is a $2{-}PIIM$ that can identify $(Z \times I) \cup (I \times Z)$ (pairs of functions, one from I, one from Z, in any order). By the recursion theorem [14] there is a program e such that:

$$\varphi_e(x) = \begin{cases} 1 & \text{if } e = x; \\ 0 & \text{otherwise.} \end{cases}$$

Let $f = \varphi_e$, then $f \in Z \cap I$. A contradiction is obtained by constructing an M' that can EX identify $Z \cup I$. Suppose $g \in Z \cup I$. M', on input from g, simulates $M(f, g)$ and outputs

M's guesses for g. We have assumed M will infer $\langle f, g\rangle$. Consequently, M' will infer g, the desired contradiction.

VIII. Anomalies and Open Problems

The Blums [5], considered a form of inference by IIMs permitting the inference machine to converge to a program that only computed the input function correctly on all but finitely many arguments. More sets of recursive functions become inferrible under this relaxed criterion of correctness. Still, there is no single inference machine capable of inferring all the recursive functions. This notion was refined in [6] to give an upper bound on the number of points of disagreement (anomalies) between the function being used as input and the one computed by the final program produced by the inference machine. A version of the team hierarchy theorem used in the proofs above also holds for the inference of programs with anomalies [25].

The definitions of inference by SIIMs and PIIMs can easily be extended to consider the inference of sequences of programs with a few anomalies and the parallel inference of programs with some number of anomalies. Since our proofs are all by reduction to another inference problem and analogues of the problems we reduce to exist for anomalous inference, all of our results will "relativize" to the case of suitable inference with anomalies. The exact form of this relativization is an open problem. Consider a sequence $\langle f_1, f_2\rangle$ and a program e_1 that computes f_1 everywhere except on 2 anomalous inputs. Can the SIIM learn f_2 with respect to 2 anomalies, given e_1? Maybe the SIIM should be allowed 4 anomalies when trying to learn f_2?

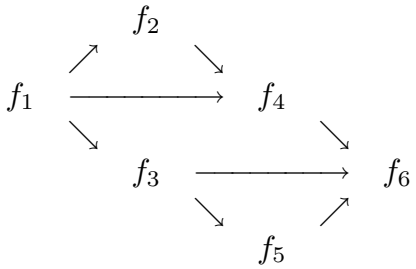
The consideration of anomalies raises several interesting questions. In [25] the trade-offs between the number of anomalies allowed and the size of the team performing the inference were investigated. What are the relationships between the number of anomalies and the number of SIIMs performing some inference? Between the number of anomalies and the number of functions that a PIIM sees?

Team inference by IIMs was equated with probabilistic inference [21]. The trade-offs found in [25] were generalized to include trade-offs with probabilities [22]. All the definitions in this paper and the ones alluded to above can be made with respect to probabilistic inference. There is probably a trade-off between the team size for sequence inference and the probability of the inference being successful. Similarly, we suspect there is a trade-off between probability and the number of functions a PIIM sees as input.

The notion of J-learnable can be used to cover much more general situations of using past knowledge to aid in the acquisition of new knowledge. Our discussion of sequence learning considered only cases where a programs for the first i functions were required to infer the $i+1^{\text{st}}$ function. A graph of the dependencies for a length 6 sequence of functions of the type considered above would look like:

$$f_1 \rightarrow f_2 \rightarrow f_3 \rightarrow f_4 \rightarrow f_5 \rightarrow f_6$$

Here an arc $x \rightarrow y$ means "knowledge of x is necessary in order to learn y." The notion of J-learnability can also be used to discuss more complicated learning dependencies, for example:



In the situation depicted above, programs for f_1 and f_2, but not f_3, are needed to learn f_4. We believe the techniques employed in this paper should be enough to answer questions concerning a finite, acyclic dependency structure.

IX. Conclusions

We have shown that, in some sense, computers can be taught how to learn how to learn. The mathematical result constructed sequences of functions that were easy to learn, provided they were learned one at a time in a specific order. Furthermore, the sequences

of functions constructed above are impossible to learn, by an algorithmic device, if the functions are not presented in the specified order. This result was extended to consider teams of inference machines, each trying to learn the same sequences of functions.

As with any mathematical model, there is some question as to whether or not our result accurately captures the intuitive notion that it was intended to. The types of models discussed were not intended to be an exhaustive list of models of learning how to learn. A fruitful avenue of research would be to clarify what are the most appropriate models with which to discuss the issues raised in this paper. Independently of how close our proof paradigm is to the intuitive notion of learning how to learn, if there were no formal analogue to the concept of machines that learn how to learn, then our result could not possibly be true. Our proof indicates not only that it is not impossible to program computers that learn based, in part, on their previous experiences, but that it is sometimes impossible to succeed without doing so.

The conclusion reached in [15,16] was that retaining knowledge learned in one learning effort could make the next learning effort less time consuming. Our result shows that sometimes first learning one function is a necessary step in order to infer some other function. A next step is to incorporate complexity theoretic concepts with our proof techniques to get theoretical results ontologically establishing the conclusions of Laird et al. It may also be possible to define similar notions of using knowledge from one learning effort in the next for Valiant's model of learning [26]. Techniques used in non-complexity theoretic inductive inference have played a fundamental role in subsequent studies of the complexity of inductive inference [9,24].

Also considered were inference machines that input several functions simultaneously with the hope that input from one function will help in the inference of another. Sets of n tuples of functions were constructed such that if a suitable inference machine saw any group of m of the functions from the tuple ($m < n$) then each of the m functions would be inferrible. Furthermore, no group of $m - 1$ functions from the tuple was sufficient to

admit the inference of those $m-1$ functions. Another view of this results is that there is some concept that is presented in n pieces such that any $m < n$ pieces are enough to figure out the concept, but no collection of $m-1$ pieces is sufficient. This is analogous to secret sharing in cryptography.

X. Acknowledgements

Our colleagues, Jim Owings and Don Perlis, made some valuable comments on the exposition. The second author wishes to thank C.W. and B.N. whose actions provided him with more time to work on this paper.

References

1. Angluin, D. Inference of reversible languages. *Journal of the Association for Computing Machinery 29* (1982), 741–765.

2. Angluin, D. and Smith, C. H. Inductive inference: theory and methods. *Computing Surveys 15* (1983), 237–269.

3. Angluin, D. and Smith, C. H. Inductive inference. In *Encyclopedia of Artificial Intelligence*, S. Shapiro, Ed., 1987.

4. Barzdin, J.A. and Podnieks, K. M. The theory of inductive inference. *Proceedings of the Mathematical Foundations of Computer Science* (1973), 9–15. Russian.

5. Blum, L. and Blum, M. Toward a mathematical theory of inductive inference. *Information and Control 28* (1975), 125–155.

6. Case, J. and Smith, C. Comparison of identification criteria for machine inductive inference. *Theoretical Computer Science 25,* 2 (1983), 193–220.

7. Cherniavsky, J. C. and Smith, C. H. Using telltales in developing program test sets. Computer Science Dept. TR 4, Georgetown University, Washington D.C., 1986.

8. Cherniavsky, J. C. and Smith, C. H. A recursion theoretic approach to program testing. *IEEE Transactions on Software Engineering SE-13,* 7 (1987), 777–784.

9. Daley, R. P. and Smith, C. H. On the complexity of inductive inference. *Information and Control 69* (1986), 12–40.

10. Freivalds, R. V. and Wiehagen, R. Inductive inference with additional information. *Electronische Informationsverabeitung und Kybernetik 15,* 4 (1979), 179–184.

11. Gasarch, W. I. and Smith, C. H. Learning concepts from subconcepts. Computer Science Department TR 1747, UMIACS TR 86-26, 1986.

12. Gold, E. M. Language identification in the limit. *Information and Control 10* (1967), 447–474.

13. Hutchinson, A. A data structure and algorithm for a self–augmenting heuristic program. *The Computer Journal 29,* 2 (1986), 135–150.

14. Kleene, S. On notation for ordinal numbers. *Journal of Symbolic Logic 3* (1938), 150–155.

15. Laird, J., Rosenbloom, P., and Newell, A. Towards chunking as a general learning mechanism. In *Proceedings of AAAI 1984*, Austin, Texas, 1984.

16. Laird, J., Rosenbloom, P., and Newell, A. Chunking in Soar: the anatomy of a general learning mechanism. *Machine Learning 1,* 1 (1986).

17. Machtey, M. and Young, P. *An Introduction to the General Theory of Algorithms.* North-Holland, New York, New York, 1978.

18. Miller, G. The magic number seven, plus or minus two: Some limits on our capacity for processing information. *Psychology Review 63* (1956), 81–97.

19. Minicozzi, E. Some natural properties of strong-identification in inductive inference. *Theoretical Computer Science 2* (1976), 345–360.

20. Osherson, D., Stob, M., and Weinstein, S. *Systems that Learn.* MIT Press, Cambridge, Mass., 1986.

21. Pitt, L. A Characterization of Probabilistic Inference. In *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*, Palm Beach, Florida, 1984.

22. Pitt, L. and Smith, C. Probability and plurality for aggregations of learning machines. *Information and Computation.* To appear.

23. Rogers, H. Jr. Gödel numberings of partial recursive functions. *Journal of Symbolic Logic 23* (1958), 331–341.

24. Shafer-Richter, G. Über eingabeabhängigkeit und komplexität von inferenzstrategien. Diplom–Mathematikkerin, Technische Hochschule, Aachen, Germany, 1984.

25. Smith, C. H. The power of pluralism for automatic program synthesis. *Journal of the ACM 29,* 4 (1982), 1144–1165.

26. Valiant, L. G. A theory of the learnable. *Communications of the ACM 27,* 11 (1984), 1134–1142.

27. Weyuker, E. J. Assessing test data adequacy through program inference. *ACM Transactions on Programming, Languages and Systems 5,* 4 (1983), 641–655.

28. Wiehagen, R. Characterization problems in the theory of inductive inference. *Lecture Notes in Computer Science 62* (1978), 494–508.

29. Wiehagen, R., Freivalds, R., and Kinber, E. K. On the power of probabilistic strategies in inductive inference. *Theoretical Computer Science 28* (1984), 111–133.