Learning from Different Teachers

DANA ANGLUIN

dana.angluin@yale.edu

Computer Science Department, Yale University, P.O. Box 208285, New Haven, CT 06520-8285, USA

MĀRTIŅŠ KRIKIS

martins.krikis@intel.com

Intel Corporation, 75 Reed Rd., Hudson, MA 01749, USA

Editor: Lisa Hellerstein

Abstract. We introduce a new model of a learner learning an unknown concept from examples with a teacher's help. In such models, "outright coding" refers to a situation in which the teacher sends the learner a representation of the concept, either directly or encoded via the examples. Previous models have used adversarial learners or adversarial teachers to try to prevent outright coding. Our model is an attempt to reflect more directly some of the reasons that outright coding is not a common mode of human learning.

We model the learner as a Turing machine with oracle access to another programming system, called its "function box." The programming system in its function box is initially unknown to the learner. The target concept is a partial recursive function and the goal of the learner is to find in its function box a function that is equal to or extends the target concept. We exhibit a class of learner/teacher pairs in which the learner can learn any partial recursive function, provided that the learner's function box is "not too much slower" than the teacher's. This result is shown not to hold if the learner's function box can contain an arbitrary acceptable programming system.

Keywords: concept learning, teacher, programming system, hint, complexity bound, oracle Turing machine

1. Introduction

In this paper we introduce and explore a new model for the situation of a teacher attempting to teach a concept to a learner. Several different formal models of teaching have been studied, addressing different aspects of the problem, for example, the work of Goldman and Kearns (1995) and Shinohara and Miyano (1991) on teaching dimension, and the work of Goldman and Mathias (1996), Jackson and Tomkins (1992), and Mathias (1995) on more helpful teachers. The basic paradigm for these models may be described as follows. The teacher and learner are each modeled as computers, and the teacher is assumed to have a program p representing the concept to be taught to the learner. The teaching protocol involves examples of the concept and possibly other information from which the learner is to develop a program p' that also represents the target concept.

In this situation, the term *outright coding* describes a protocol in which the teacher transmits (using an encoding via examples or other information) the text of the program p to the learner. This, most authors agree, does not seem to involve learning in an interesting sense, and it is generally prevented by deliberately chosen features of the model. For example, the learner may be required to learn from a variety of more or less adversarial teachers, or, conversely, the teacher may be required to teach a variety of more or less adversarial learners.

One of the main goals of our new model is to try to reflect in a more direct way the reasons that outright coding is not a common mode of human learning. Our analysis of this issue involves two related ideas.

- The “hardware and software environment” differs quite substantially from one person to the next.
- In many human endeavors, simulation does not give a feasible solution.

In support of the first point, it appears that the interconnections in human neural circuitry involve a large degree of randomness and influence by external stimuli, making a neuron-by-neuron isomorphism of two human nervous systems very unlikely. The fantasy of somehow transferring the patterns of your neural activation to my neurons to allow me to experience exactly what you are experiencing is quite improbable. Taking the neural level as “hardware” and the pattern of activations as “program,” our hardware is just different—I cannot meaningfully run your program. In the case of humans, learners and teachers are just “too different” for outright coding to work.

Of course, there are other more abstract levels of human cognitive functioning where we could make analogies to hardware and program, but it does not seem plausible that there is any level for which the analogy of transferring a program between two identical computers is very accurate. That analogy is perhaps becoming less accurate for real computers as well. As computers have become more complex, the process of porting a program from one machine to another has been complicated by differences between processors, buses, networks, peripherals, communications protocols, programming languages, operating systems, and other installed software, as well as the threat of computer viruses. As a result, techniques for transferring functionality between computers may have to borrow more and more heavily from biological and human models, including learning.

This brings us to the second point, whether simulation can help. Simulation has been a powerful tool in theoretical computer science, in a large variety of settings. Consequently, a theoretical computer scientist’s almost immediate reaction to the situation of two different computational systems is to ask if they can efficiently simulate each other. If so, then for many theoretical purposes they are equivalent. If we take this approach then even if we assume that the teacher and the learner are modeled by non-identical computers, the learner can use the teacher’s program by simulating it, and a version of outright coding is still possible. But it is instructive to see what simulation means in a concrete setting.

Suppose we map the idea of simulation to the situation of someone trying to teach another person to juggle; what problems arise? The desired behavior is an extended process that interacts physically with the world of muscles, juggling balls, air, light, and so on, in a time-critical fashion. What must the learner do to simulate the teacher’s juggling program? The perceptual signals of vision, touch, and pressure from the learner must be translated into equivalent ones for the teacher’s program. Motor signals generated by the teacher’s program for the teacher’s muscles must be translated to equivalent motor signals for the learner’s muscles. And all this must be done before the juggling balls hit the floor. There are several issues here: the real-time constraint, the fact that most of the process is not open to introspection, and the complexity of specifying the process in terms of its inputs and

outputs. It seems extremely unlikely that people have the capacity to simulate one another in anything like this sense; nonetheless, juggling is quite teachable.

In addition to the obvious constraints on memory and speed, one major obstacle to simulation in this sense for humans is the fact that many of a person's capabilities are only partly and poorly known to that person. This is most evident in embodied capabilities, for example, the sequence of motor signals and responses that allow a person's hands to type a word, but it probably holds equally of more strictly cognitive capabilities, like the ability to picture a friend's face. Many of our capabilities are, in effect, "black boxes" in the sense that we learn to produce appropriate control signals to achieve certain effects, but the actual details of how the signals lead to the effects are opaque to us.

We have attempted to reflect some of these considerations in the model we present. We are aware that it is only a first approach to the issues, with many drawbacks of its own, but we hope that it will inspire others to think more about models of teaching.

2. Preliminaries

Since we are considering issues that lie at the foundation of theoretical computer science, we work in the realm of computable functions, complexity measures, and identification in the limit. We follow Machtey and Young (1978) or Smith (1994) in our definitions concerning programming systems and complexity measures, and start from the model of inductive inference introduced by Blum and Blum (1975).

$\mathbf{N}$ denotes the set of natural numbers, $\{0, 1, 2, 3, \dots\}$. We consider partial functions from $\mathbf{N}$ to $\mathbf{N}$. The subset of $\mathbf{N}$ on which a function f is defined is its *domain*, denoted $\text{Dom}(f)$. The *graph* of f is the set of all ordered pairs $(x, f(x))$ such that $x \in \text{Dom}(f)$. The function g *extends* the function f if the graph of f is a subset of the graph of g ; this is denoted by $g \supseteq f$. The *composition* of functions f and g is the function h such that $h(x) = f(g(x))$ if both $g(x)$ and $f(g(x))$ are defined, otherwise $h(x)$ is undefined.

We fix a particular listing $t_0, t_1, t_2, \dots$ of all Turing machines, which we term the *standard listing*. Each Turing machine computes a partial function from $\mathbf{N}$ to $\mathbf{N}$; for those inputs for which the Turing machine does not halt, the corresponding function is undefined. The *partial recursive functions* are those partial functions from $\mathbf{N}$ to $\mathbf{N}$ that can be computed by a Turing machine. The *total recursive functions* are those partial recursive functions that are total, that is, defined for every input. We also consider functions of pairs or triples of natural numbers. Formally, we assume a fixed, easily-computed one to one correspondence mapping an ordered pair of natural numbers (x, y) to a single natural number $\langle x, y \rangle$. The i -th projection function for this mapping is π_i ; therefore, $\pi_1(\langle x, y \rangle) = x$ and $\pi_2(\langle x, y \rangle) = y$. The projection functions are also assumed to be easily computable. Formally, $u(i, x)$ and $g(i, x, s)$ are interpreted as $u(\langle i, x \rangle)$ and $g(\langle \langle i, x \rangle, s \rangle)$, but we generally prefer the informal notation.

2.1. Programming systems and complexity measures

A *programming system* is any sequence $\phi_0, \phi_1, \phi_2, \dots$ of partial recursive functions that contains all the partial recursive functions. The *universal function* u for this programming

system is $u(i, x) = \phi_i(x)$. The programming system is *universal* if its universal function is partial recursive.

A *composition function* c for such a system is a total function c such that for any i and j if $k = c(i, j)$ then ϕ_k computes the same function as ϕ_i composed with ϕ_j . A universal programming system is *acceptable* if it has a recursive composition function. We assume that our standard listing of Turing machines, $t_0, t_1, t_2, \dots$ is chosen so that it is an acceptable programming system; this will be true for essentially any natural method of listing all Turing machines.

The following basic result (from Machtey & Young (1978)) implies that any two acceptable programming systems are equivalent in the sense that they can simulate each other.

Lemma 1. *Suppose ϕ is a universal programming system and ψ is an acceptable programming system. Then there exists a total recursive translation function r such that $\phi_i = \psi_{r(i)}$ for all i .*

We now give the definition of computational complexity measures introduced and studied by Blum (1967). Let $\phi_0, \phi_1, \phi_2, \dots$ be any acceptable programming system. A sequence $\Phi_0, \Phi_1, \Phi_2, \dots$ of partial recursive functions is a *computational complexity measure* (on the given acceptable programming system) if it satisfies the following conditions.

1. For all i and x , $\phi_i(x)$ is defined if and only if $\Phi_i(x)$ is defined.
2. $\Phi_i(x) \leq y$ is a total recursive predicate of i, x , and y .

An important example of an acceptable programming system with a computational complexity measure is given by taking the standard listing of Turing machines $t_0, t_1, t_2, \dots$ together with the functions $T_0, T_1, T_2, \dots$, where $T_i(x)$ is the number of steps taken by the i th Turing machine on input x until it halts, or undefined, if it doesn't halt.

For readers unfamiliar with these notions, we briefly indicate why the definitions are satisfied in this case. Every partial recursive function is computed by some Turing machine, so the sequence $t_0, t_1, t_2, \dots$ is a programming system. The universal function $u(i, x)$ for this system can be computed by simulating the action of the i th Turing machine on input x , so the programming system is universal. A composition function $c(i, j) = k$ can be computed by taking the i th and j th Turing machines and combining them into another Turing machine, the k th, that runs the i th machine and, if it halts, runs the j th machine on the output of the i th machine. This shows that the programming system is acceptable.

To see that T_i is partial recursive, we can compute $T_i(x)$ by simulating the i th Turing machine on input x , counting the simulated steps. If the i th machine on input x does not halt, then neither does the simulation; otherwise, the simulation outputs the total number of simulated steps until the i th machine halts on input x . This shows both that T_i is partial recursive and that t_i and T_i have the same domains, satisfying condition (1). Finally, given i, x, y , we can compute whether $T_i(x) \leq y$ by simulating the i th Turing machine for (at most) y steps. If the machine halts at or before the y th step, then the predicate is true, otherwise, the simulation is stopped and the predicate is false. This shows that the predicate is total recursive, satisfying condition (2).

There are many other examples of acceptable programming systems and computational complexity measures, based on other models of computation and other computational resources. The recursive equivalence of two acceptable programming systems can be extended to complexity measures as follows (from Machtey & Young (1978)).

Lemma 2. *If ϕ and ψ are acceptable programming systems with computational complexity measures Φ and Ψ , respectively, and r is a recursive translation of ϕ into ψ then there is a recursive bounding function b such that for all i and for all but finitely many x , $\Psi_{r(i)}(x) \leq b(x, \Phi_i(x))$.*

Thus, not only can programs in one system be effectively translated into programs in the other, there is also a single recursive function bounding the complexity of the translated programs in terms of the complexity of the original programs.

The ideas of computational complexity measures found elegant application in the seminal paper on inductive inference by Blum and Blum (1975). Our results are in large part inspired by their paper; we now explain their model of inductive inference.

2.2. Inductive inference machines

In Blum and Blum's model, the learner is a Turing machine M , the thing to be learned is a partial recursive function f , and the examples from which it is learned are input/output pairs from the graph of f .

A *presentation* of f is an infinite sequence I of elements, each either the delay symbol $*$ or a pair (x, y) such that $y = f(x)$. For every x in the domain of f , there must be at least one element (x, y) of I . Thus, I presents every element of the graph of f , in any order, with repetitions allowed, interspersed arbitrarily with the symbol $*$. (Note that the only presentation of the everywhere undefined function is an infinite sequence of $*$'s.) We assume that M has an input tape, the *examples tape*, with a presentation I of f written on it.

M is allowed to run indefinitely; from time to time it may write a conjecture about the function f on a special output tape. Each conjecture is an index j in a fixed programming system $\phi_0, \phi_1, \phi_2, \dots$. As M runs for longer and longer, it produces a finite or infinite sequence of conjectures. The sequence is said to *converge* to the index j if either the sequence is finite and its last element is j , or the sequence is infinite and equal to j except at a finite number of places. Each point in the sequence at which a conjecture is followed by a different conjecture is a *mind change*. If the sequence of conjectures does not converge to any value j , it must either be the empty sequence or contain an infinite number of mind changes.

M *identifies f in the limit from I* if the sequence of M 's conjectures converges to some j such that ϕ_j extends f . Note that if f is a total recursive function, then ϕ_j must be equal to f , but if f is partial, we consider as correct a conjecture that agrees with f on the domain of f , but may also be defined elsewhere. M *identifies f in the limit* provided that for all presentations I of f , M identifies f in the limit from I . If S is any set of partial recursive functions, M *identifies S in the limit* provided that for all $f \in S$, M identifies f in the limit.

2.3. An example inductive inference machine

Let C_0 denote the class of partial recursive functions f such that there is some total extension f' of f that is constant except at finitely many inputs. We describe an inductive inference machine M_0 that successfully identifies every function in C_0 . M_0 uses Turing machines as its programming system.

Initially, M_0 conjectures an index i of a Turing machine that outputs 0 for every input. M_0 initializes a table W to be empty and a value v to be 0, and then reads in successive entries from the presentation on the examples tape.

Whenever the entry is a pair (x, y) , M_0 checks to see if this pair is in the table W . If so, it does not output anything new, and goes on to the next input. Otherwise, M_0 adds (x, y) to the table W and compares y with the value v . If they are the same, it does not output anything new, and goes on to the next input. If they are different, M_0 sets v to y and outputs an index i for a Turing machine defined as follows. On input x' , if (x', y') is in W , then output y' , otherwise, output y .

To see that M_0 identifies C_0 , let f be any element of C_0 . If f is nowhere defined, then M_0 outputs an index for the constant function 0, which is an extension of f , and never changes its mind.

If f is defined for at least one input and at most finitely many inputs, then there is some first point in the presentation when all of the pairs in the graph of f have been presented. If (x, y) is the pair presented at this point, then M_0 will conjecture an index for the function that is constantly equal to y , except on those inputs x' such that (x', y') is in the graph of f , where the value of the function will be y' . This is a total recursive extension of f , and M_0 does not change its mind after this.

If f is defined for infinitely many inputs, then it must give the same constant output y at all but finitely many of them. There is some first point in the presentation that all the pairs (x', y') in the graph of f with $y' \neq y$ have been presented. There must be (infinitely many) pairs (x, y) not yet in the table W presented after this. On the first one, M_0 conjectures an index for the function that agrees with f where it is defined, and gives the output y everywhere else, which is an extension of f . Subsequently, M_0 gets no new table entries (x', y') with $y' \neq y$, and therefore does not change its conjecture.

Thus, in each case, M_0 identifies f in the limit. Note that although M_0 successfully identifies every $f \in C_0$ in the limit, it does not “know” that its conjecture is correct at any finite stage in the process. This phenomenon is typical of identification in the limit.

3. A new model of a learner

In this section we give our new model of a learner and some basic results for the model; in the next section we add a teacher to the model. In Blum and Blum’s definition, the learner is modeled as an inductive inference machine that is assumed to be fully cognizant of the programming system it uses to express its conjectures. For instance, in the example given in the preceding section, the inductive inference machine M_0 is presumed to be able to convert its current table W and value v into the index i of a Turing machine that looks up the input in the table W and outputs the associated value if the lookup succeeds and outputs v otherwise.

The key point in our model is to weaken this assumption, to restrict the learner to a more limited knowledge of the programming system in which it must make its conjectures.

We assume that the learner is a Turing machine with oracle (or “black box”) access to an acceptable programming system ϕ with a complexity measure Φ . As in Blum and Blum’s definition, the concept to be learned is a partial recursive function f given by a presentation on the examples tape and the goal of the learner is to find in the limit a correct program for f , but now the conjectures must come from the programming system ϕ . Approximately speaking, the Turing machine component represents the purely cognitive operations of the learner, and the programming system ϕ represents the repertoire of possible actions of the learner, or, more succinctly, “thinking” versus “doing”. Again speaking approximately, the learner must learn to “do” the exemplified thing, and not just “imagine” doing it. That is, having a correct program for f in the standard Turing machine system is not enough; the learner must find a correct program in the system ϕ .

If the system ϕ is known to be the same as the standard Turing machine system, or if the learner knows a program in the system ϕ to simulate programs in the standard Turing machine system, then this distinction collapses. We would like to maintain the distinction and avoid solutions that involve simulation. For this purpose, we model lack of self-knowledge on the part of the learner. We assume that the programming system ϕ is more or less arbitrary and unknown to the learner, and require that a learning algorithm (for the Turing machine component of the learner) work correctly when ϕ is any one of a whole class of possible programming systems. We can characterize this aspect of the model as treating the action-space of the learner as part of the initially unknown environment. A new-born animal may be in a somewhat analogous situation, having to discover how to focus its eyes or move its limbs partly by experiment. By analogy to expert and naive users of computers, we term our new model of a learner a *naive learner*. We now give a formal specification of the model.

3.1. Function boxes

The unknown programming system ϕ with its complexity measure Φ will be presented to the learner via oracle access to a function box. A *function box* is an arbitrary total recursive function $g(i, x, s)$ of three arguments, taking values in $\mathbb{N} \cup \{?\}$. We could re-code the co-domain to be just the natural numbers, but we prefer human intelligibility.

Let g be an arbitrary function box and i and x natural numbers. Define $G_i(x)$ to be the least value of s such that $g(i, x, s)$ is a natural number, if any; otherwise, $G_i(x)$ is undefined. If $G_i(x)$ is defined, then define $g_i(x)$ to be $g(i, x, G_i(x))$; otherwise, $g_i(x)$ is undefined. In other words, if we fix i and x , and examine the successive values of $g(i, x, s)$ for $s = 0, 1, 2, \dots$, we observe a mixture of natural numbers and ?’s. If there are no natural numbers in the sequence, then $G_i(x)$ and $g_i(x)$ are both undefined. Otherwise, there is a first natural number y in the sequence, and we define $g_i(x)$ to be y and $G_i(x)$ to be the value of s when y first appears.

We say that the function box g *contains* the functions $g_0, g_1, g_2, \dots$. These functions form the space of possible hypotheses for the learner. Note that all these functions must be partial recursive, because we can compute $g_i(x)$ by evaluating $g(i, x, s)$ for $s = 0, 1, 2, \dots$ until we find the first value $\neq ?$, if any. However, there may be some partial recursive functions

not contained in a given function box.¹ A function box g is *full* if it contains all the partial recursive functions, i.e., if the functions $g_0, g_1, g_2, \dots$ form a programming system.

For every function box g , its universal function $U_g(i, x) = g_i(x)$ is necessarily partial recursive. If a function box is full, then the programming system it contains is necessarily universal, and we shall refer to the function box as *universal*. A universal function box is *acceptable* if the programming system it contains is acceptable.

What of the associated functions $G_0, G_1, G_2, \dots$? They satisfy the two conditions in the definition of a computational complexity measure with respect to $g_0, g_1, g_2, \dots$. To see this, note that G_i has the same domain as g_i and is partial recursive, computed by a program that evaluates $g(i, x, s)$ for $s = 0, 1, 2, \dots$ and outputs the least value of s such that $g(i, x, s) \neq ?$, if any. To decide whether $G_i(x) \leq y$ given i, x , and y , compute $g(i, x, s)$ for $s = 0, 1, \dots, y$. If all these values are $= ?$, then the predicate is false, otherwise, true.

We think of G_i as the complexity function corresponding to g_i and call it the *measure* of g_i . We say that g contains measures $G_0, G_1, G_2, \dots$. This slightly generalizes Blum's definition of a computational complexity measure, because we do not require g to contain a programming system. In the case that g is an acceptable function box, we get the original definition.

Lemma 3. *If g is an acceptable function box, then $g_0, g_1, g_2, \dots$ is an acceptable programming system, and $G_0, G_1, G_2, \dots$ is a computational complexity measure for it.*

In the opposite direction, we can start with a sequence of functions and measures satisfying certain properties and transform them into a function box.

Lemma 4. *Suppose $h_0, h_1, h_2, \dots$ and $H_0, H_1, H_2, \dots$ are two sequences of partial recursive functions such that (1) the universal function $u(i, x) = h_i(x)$ is partial recursive, (2) for all i , the domains of h_i and H_i are the same, and (3) there is a total recursive function to decide whether $H_i(x) \leq y$ given i, x , and y as inputs. Then there is a function box g containing the functions $h_0, h_1, h_2, \dots$ with the measures $H_0, H_1, H_2, \dots$.*

Proof: Define $g(i, x, s) = h_i(x)$ if $H_i(x) \leq s$ and $g(i, x, s) = ?$ otherwise. Then the conditions on $h_0, h_1, h_2, \dots$ and $H_0, H_1, H_2, \dots$ imply that g is a total recursive function, and that $g_i = h_i$ and $G_i = H_i$ for all i . Thus g is a function box that contains the functions $h_0, h_1, h_2, \dots$ with the measures $H_0, H_1, H_2, \dots$ as claimed. $\square$

Note that function boxes obtained by this transformation have the special property that for each i and x , the values $g(i, x, s)$ for $s = 0, 1, 2, \dots$ are either all equal to $?$ or consist of an initial segment of $?$'s followed by an infinite constant sequence. Arbitrary function boxes are not necessarily of this form. If we apply this transformation to an acceptable programming system with a computational complexity measure, we get an acceptable function box.

We define a relation of being “not more than b slower” between two function boxes as follows. Let $b(x, s)$ be a total recursive function. A function box g is *b -related* to a function box g' if for every i there exists a j satisfying the following properties.

1. g_j extends g'_i .
2. $G_j(x) \leq b(x, G'_i(x))$ for all but finitely many x in the domain of g'_i .

This is analogous to the bounding relation between two acceptable programming systems with complexity measures (Lemma 2) except that we do not require the function boxes to be universal or acceptable, we do not assume a recursive translation function r , and we allow the function in g to extend the corresponding function in g' . However, as a corollary of Lemmas 1 and 2, we have the following.

Lemma 5. *For any acceptable function boxes g and g' there exists a total recursive function $b(x, s)$ such that g is b -related to g' .*

3.2. Naive learners

A *naive learner*, usually denoted by L , is an oracle Turing machine with the usual work tapes and certain other specialized tapes. For input, the learner has read-only access to the *examples tape*, which contains a presentation of some partial recursive function f , referred to as the *target function*. For output, the learner has write-only access to the *guess tape*, where it writes its successive conjectures about f . The learner has oracle access to an arbitrary function box g using two additional tapes and two special states. The learner may write particular values of i , x , and s on the *function box query tape* and enter the function box query state. The value of $g(i, x, s)$ is placed on the *function box answer tape*, and the learner is resumed in the function box answer state. In other words, the learner has subroutine access to g . The naive learner L with oracle access to the function box g is denoted $L(g)$.

Given a naive learner L , a function box g , a partial recursive function f , and a presentation of f on the examples tape, we say that the learner $L(g)$ *converges to j* if the sequence of outputs on the guess tape finitely converges to j . We say that $L(g)$ *converges correctly* if $L(g)$ converges to j and g_j extends the function f . We say that $L(g)$ *converges incorrectly* if $L(g)$ converges to j and g_j does not extend the function f . If there is no index j such that $L(g)$ converges to j , then either $L(g)$ never outputs any conjecture, or $L(g)$ changes its mind infinitely often.

Given a naive learner L , a function box g , and a partial recursive function f , we say that $L(g)$ *learns f* if for every presentation I of f on the examples tape $L(g)$ converges correctly. If S is a set of partial recursive functions, $L(g)$ *learns S* if for every function $f \in S$, $L(g)$ learns f . Clearly, $L(g)$ cannot learn a function f unless there is a function g_i that extends f . In this paper we primarily consider two cases: the learnability of a single function, or the learnability of the class of all partial recursive functions.

The primary difference between an inductive inference machine and a naive learner is that the former is an “expert” in the programming system used to express the conjectures, while the latter has very limited knowledge of that programming system. We now present results comparing inductive inference machines and naive learners.

3.3. Inductive inference machines and naive learners

First, we observe that any class of partial recursive functions that can be learned by a naive learner with a fixed universal function box can be identified in the limit by an inductive inference machine.

Lemma 6. *Let L be a naive learner, g a universal function box, and S a set of partial recursive functions such that $L(g)$ learns S . Then there exists an inductive inference machine M_g that identifies S in the limit.*

Proof: We describe M_g , which operates by simulating L . If $f \in S$ and I is a presentation of f on the examples tape of M_g , then M_g simulates L with I on its examples tape. When L requests the value of $g(i, x, s)$, M_g computes it (recall that g is a total recursive function) and returns the value to L . When L produces a conjecture j , M_g must translate the index into an equivalent index in the Turing machine programming system $t_0, t_1, t_2, \dots$. This is possible because g contains a universal programming system and the Turing machine programming system is acceptable, which implies by Lemma 1 that there is a recursive translation program r such that for all i , $g_i = t_{r(i)}$. Thus, M_g uses a program for r to translate L 's conjecture j into $r(j)$, which it writes to its output tape.

Because $f \in S$ and L learns S , the sequence of conjectures of L must finitely converge to a value j such that g_j extends f . Thus, the outputs of M_g finitely converge to $r(j)$, and $t_{r(j)}$ extends f . Thus, M_g identifies in the limit every function in S . $\square$

Next, we show that certain inductive inference techniques based on computational complexity carry over to naive learners. Let $h(x, y)$ be any total recursive function. A partial recursive function f is *h -honest* with respect to a given function box g if there is an index i such that the following two conditions hold.

1. The function g_i extends f .
2. For all but finitely many x in the domain of f , $G_i(x) \leq h(x, f(x))$.

This definition extends the definition of Blum and Blum (1975) to the case of function boxes. Note that the definition includes the case of a complexity class such as n^3 time, but also allows for running times that depend on the output as well as the input to the program.

For each recursive function h , Blum and Blum give an inductive inference machine that learns all the h -honest functions with respect to a fixed acceptable programming system with a complexity measure. The basic idea is to search through the fixed programming system for an index that agrees with the pairs (x, y) on the examples tape. The bound h allows the inductive inference machine to decide that the current program is taking “too long” (i.e., the complexity is more than $h(x, y)$) to produce an output for input x . The possibility of a finite number of inputs that violate the bound is handled by returning infinitely often to try the same index, with a different running-time bound for the exceptions.

This same algorithm can be implemented essentially without change by a naive learner, giving the following result.

Lemma 7. *For every total recursive function h there exists a naive learner L_h such that for every function box g , $L_h(g)$ learns all the functions that are h -honest with respect to g .*

Proof: Let h be a total recursive function and g a function box. Let f be a partial recursive function and I a presentation of f on the examples tape of the naive learner L_h . The key

point is to verify that $L_h(g)$ can implement Blum and Blum's algorithm with only oracle access to the function box g . $L_h(g)$ operates in stages, $n = 0, 1, 2, \dots$.

At the n th stage, let $n = \langle i, k \rangle$. At the beginning of the stage, write the conjecture i on the guess tape. For $j = 1, 2, \dots$, examine the j th entry on the examples tape. If it is $*$, then go on to the next value of j . Otherwise, the entry is a pair (x, y) . Test whether $G_i(x) \leq \max\{k, h(x, y)\}$. If not, then go on to stage $n + 1$. If so, then test whether $g_i(x) = y$. If not, then go on to stage $n + 1$. If so, then go on to the next value of j in stage n .

We argue that if f is any partial recursive function that is h -honest with respect to g , $L_h(g)$ learns f . Clearly, all the steps specified in the above algorithm are effective. In particular, we do not attempt to find $g_i(x)$ in the function box until we have verified that $G_i(x)$ is defined.

By the definition of h -honest with respect to g , there exists an index i such that g_i extends f and for all but finitely many x in the domain of f , $G_i(x) \leq h(x, f(x))$. Because g_i extends f , $G_i(x)$ is defined for every x in the domain of f . Thus, if we consider the finitely many x in the domain of f such that $G_i(x) > h(x, f(x))$, we can choose k sufficiently large that $G_i(x) \leq k$ for each such x . Then we have for all x in the domain of f ,

$$G_i(x) \leq \max\{k, h(x, f(x))\}.$$

Let n' be the smallest natural number such that for $n' = \langle i', k' \rangle$, $g_{i'}$ extends f and $G_{i'}(x) \leq \max\{k', h(x, f(x))\}$ for all x in the domain of f .

At every stage $n < n'$, if $n = \langle i, k \rangle$ then there must be some x in the domain of f such that either $G_i(x) > \max\{k, h(x, f(x))\}$ or $G_i(x) \leq \max\{k, h(x, f(x))\}$ and $g_i(x) \neq f(x)$. In either case, the algorithm will eventually go to the next stage. When the algorithm reaches stage n' , it outputs the conjecture i' to the guess tape, and never proceeds to the next stage, and therefore never changes its mind. Because $g_{i'}$ extends f , this means that $L_h(g)$ learns f . $\square$

As Blum and Blum proved, L_h is *reliable* on the set of partial functions from $\mathbf{N}$ to $\mathbf{N}$. This means that if the examples tape holds a presentation of any partial function (not necessarily partial recursive), the learner L_h will not converge incorrectly. In other words, if L_h converges at all, it converges correctly.

Blum and Blum go on in their paper to develop a more powerful *a posteriori* method of inductive inference. The inductive inference machine that implements this method must be rather "expert" in its knowledge of the programming system of its conjectures. In particular, the inductive inference machine must be able to apply both padding and the recursion theorem to the system, which means that the method is not available to a naive learner.

3.4. What a naive learner cannot do

Suppose the task of the naive learner is to learn the constant zero function, intuitively a very simple function. An inductive inference machine to accomplish this task can simply output an index for any Turing machine that computes the constant zero function and halt. However, the naive learner must locate (in the limit) an index in its function box for the constant zero function.

Theorem 1 below shows that for every naive learner, there is some acceptable function box for which the learner will fail at this task. The explanation for this rather counter-intuitive result is that there are acceptable programming systems in which the constant zero function is not at all simple to compute; thus the task of coping with all acceptable programming systems is quite daunting. Let $Z(x) = 0$ for all $x \in \mathbb{N}$ be the constant zero function.

Theorem 1. *There is no naive learner L such that for all acceptable function boxes g , $L(g)$ learns the constant zero function Z .*

Proof: Assume to the contrary that such a naive learner L exists. By our assumption, for every acceptable function box g and for every presentation of the constant zero function Z on the examples tape, $L(g)$ correctly converges to an index for Z , that is, an index j such that $g_j(x) = 0$ for all $x \in \mathbb{N}$. We show how to use L to construct an inductive inference machine M that identifies in the limit every total recursive function, which is impossible (Gold, 1967).

On its examples tape M has a presentation of an arbitrary total recursive target function f . The entries on the tape are pairs $(x, f(x))$, possibly intermixed with $*$'s. Because f is total, every $x \in \mathbb{N}$ appears at least once on the tape as the first component of a pair.

Let $t_0, t_1, t_2, \dots$ denote the standard Turing machine programming system, with the step-counting complexity measure $T_0, T_1, T_2, \dots$. The goal of the inductive inference machine M is to converge correctly to an index i for f in the Turing machine programming system, that is, an index i such that $t_i = f$.

The inductive inference machine M “builds” another acceptable programming system ψ using t and the values of f on the examples tape. In the new system ψ , the constant zero function Z has the same indices as function f has in t . That is, $t_i = f$ if and only if $\psi_i = Z$. M simulates the naive learner L with some enumeration of the constant zero function on L 's examples tape and a function box g containing the programming system ψ with complexity measure T . According to our assumption, $L(g)$ converges to an index for Z in the programming system ψ , which is an index for f in the Turing machine programming system t . We now describe the construction of ψ .

Let $\ominus$ be a binary operator on natural numbers defined by $x \ominus y = |x - y|$. The new programming system ψ is defined by

$$\psi_i(x) = t_i(x) \ominus f(x).$$

If $t_i(x)$ is undefined, so is $\psi_i(x)$. The complexity measure for ψ is defined to be the Turing machine complexity measure, i.e., $\Psi_i = T_i$ for all i .

The definition of ψ makes it clear that i is an index of f in the Turing machine system if and only if i is an index of Z in ψ . In addition, we must verify that ψ is an acceptable programming system, and that Ψ is a computational complexity measure for it. Let g be the function box obtained from ψ and Ψ using the transformation of Lemma 4. We must also verify that M can simulate $L(g)$.

In order to simulate $L(g)$, M must supply the answer to each of L 's queries $g(i, x, s)$. To do this, M reads its own examples tape, containing a presentation of f , until it finds the pair $(x, f(x))$. Because f is total, this pair must appear somewhere on the tape. M then

simulates the Turing machine with index i on input x for s steps. If the simulation does not halt in that time, M supplies ? as the answer to the query $g(i, x, s)$. Otherwise, the simulation halts with output value $t_i(x)$, and M supplies the value of $t_i(x) \ominus f(x)$ as the answer to the query $g(i, x, s)$.

To see that ψ is an acceptable programming system, we note that because the standard Turing machine system is acceptable and f is total recursive, there are total recursive functions h and k such that

$$t_{h(i)}(x) = t_i(x) + f(x)$$

and

$$t_{k(i)}(x) = t_i(x) \ominus f(x).$$

Thus,

$$\psi_{h(i)}(x) = (t_i(x) + f(x)) \ominus f(x) = t_i(x)$$

and

$$\psi_i(x) = t_{k(i)}(x).$$

The partial recursive function t_i appears as $\psi_{h(i)}$, so ψ is a programming system. The universal function

$$u(i, x) = \psi_i(x) = t_{k(i)}(x)$$

is clearly partial recursive. Let c be the composition function for the standard Turing machine system and let

$$c'(i, j) = h(c(k(i), k(j))).$$

Then c' is total recursive and is a composition function for ψ because

$$\psi_{c'(i,j)}(x) = t_{c(k(i),k(j))}(x) = t_{k(i)}(t_{k(j)}(x)) = \psi_i(\psi_j(x)).$$

Thus, ψ is an acceptable programming system. To see that Ψ is a complexity measure for ψ , note that ψ_i has the same domain as $t_{k(i)}$, which in turn has the same domain as t_i and $T_i = \Psi_i$. Hence g is an acceptable function box and Theorem 1 is proved. $\square$

4. Modeling teachers

We now add a teacher to our model. The teacher has knowledge of a program for the target function f , and attempts to use it to help the learner learn f . The question arises of how

to model the programming system used by the teacher: as a fixed, known system or as an unknown system in a function box, analogous to the learner's situation. For uniformity, we prefer the latter alternative. Under this assumption, an eventual extension of the theory might permit the learner to go on to become a teacher.

Because knowledge of the programming system cannot make teaching harder, this choice makes for stronger positive results. For the same reason, non-teachability results might be weakened. However, all the negative results we present in this paper still hold when the teacher uses a fixed, known programming system.

In our model there are two agents, the teacher T and the learner L . Each one is an oracle Turing machine with its own work tapes and oracle access to its own function box. $L(g)$ denotes L with oracle access to the function box g , and $T(g')$ denotes T with oracle access to the function box g' . The teacher and learner share read-only access to the *examples tape*, which contains a presentation of a partial recursive function f . They also share read/write access to a *message tape* for communication.

The teacher has an additional input tape, the *answer tape*, which is read-only and contains a value i such that g'_i extends f . That is, the teacher "knows" the index of a program in its function box g' that computes an extension of the target function f . The learner has an additional output tape, the *guess tape*, on which it writes its conjectures about the target function f .

The teacher and learner alternate in their use of the message tape to communicate. Specifically, we assume that L has exclusive access to the message tape until it enters a special *send* state, at which time its computation is suspended. At this point, T 's (initially or previously) suspended computation is resumed until T enters its *send* state. Then T 's computation is suspended again and L 's is resumed, and so on. There are no other restrictions on the communication; in particular, the teacher is free to transmit the contents of its answer tape to the learner. Because the function boxes g and g' may be different, this information will not necessarily be helpful to the learner.

Given a learner L , a teacher T , a function box g for the learner, a function box g' for the teacher, a partial recursive function f , a presentation I of f on the examples tape, and an index i such that g'_i extends f on the answer tape, we run the communicating computations of $L(g)$ and $T(g')$, and consider the sequence of natural numbers output by $L(g)$ on the guess tape. This sequence may finitely converge to some value i . If g_i extends f , we say that $L(g)$ *converges correctly*. If g_i does not extend f , we say that $L(g)$ *converges incorrectly*. If the sequence of outputs does not finitely converge to any value, we say that $L(g)$ *does not converge*; this may be because it outputs nothing, or because it makes infinitely many mind changes.

Given a learner L , a teacher T , a function box g for L , a function box g' for T , and a partial recursive function f , we say that $L(g)$ *learns* f from $T(g')$ if there is at least one index i such that g'_i extends f and for every presentation I of f on the examples tape and every index i such that g'_i extends f on the answer tape, $L(g)$ converges correctly. Note that $L(g)$ cannot learn f from $T(g')$ unless g' contains at least one extension of f .

We extend this definition to the case of learning a class S of partial recursive functions in the usual way. $L(g)$ *learns* S from $T(g')$ if for every function $f \in S$, $L(g)$ learns f from $T(g')$. Note that S cannot be learned by $L(g)$ from $T(g')$ if T 's function box fails to

contain an extension of some function in S . In this paper we primarily consider two cases: the learnability of a single function, or the learnability of the class of all partial recursive functions. In the second case, we generally assume that the teacher's function box is at least universal.

Inspired by the work of Minicozzi (1976) and Blum and Blum (1975) on reliable (or strong) identification, we are interested in designing learning protocols that “fail gracefully” in certain situations other than those for which the protocol is specifically intended. The safety condition we consider is that the learner should avoid converging incorrectly. That is, if the learner does not converge correctly, then it does not converge at all, either by never producing any conjecture, or by changing its mind infinitely often. The possible “unanticipated situations” we consider are the following.

1. A function from outside the intended target class.²
2. A teacher other than the intended one.
3. A function box from outside the intended class for the learner.

In this paper we focus mainly on (2) and (3); our positive results are for the target class of all the partial recursive functions, though obviously there is room for generalization by parameterizing the target class of functions as well.

For “a teacher other than the intended one” we permit any infinite sequence of finite messages, $m_1, m_2, \dots$, which is used in place of a teacher as follows. When the learner enters its send state for the i -th time, the message m_i is placed on the message tape and the learner's computation is resumed. This models a large class of possible behaviors for teachers. However, the situation of the teacher causing the learner's computation to remain indefinitely suspended is not modeled; we are assuming that the teacher always makes some response, though not necessarily a helpful one. A learner L with function box g is *teacher-proof* if for any presentation of any partial recursive function f on the examples tape and any sequence of messages $m_1, m_2, \dots$ used as responses on the message tape, $L(g)$ does not converge incorrectly.

For “a function box outside the intended class for the learner” we permit any function box. Given a learner L , a teacher T , and a universal function box g' for the teacher, we say that L and $T(g')$ are *box-proof* if for any function box g , for any partial recursive function f , for any presentation of f on the examples tape, and for any index i on the answer tape such that g'_i extends f , $L(g)$ does not converge incorrectly. Any particular function box g for L might be “defective” in a variety of ways, for example, it might contain a very peculiar collection of functions, possibly only everywhere undefined functions, but none of these problems will cause $L(g)$ to converge incorrectly if L and $T(g')$ are box-proof.

Finally, we define what it means for a learner L to be box-and-teacher-proof. This is not simply a conjunction of box-proof and teacher-proof, because L must cope with combinations of “bad” boxes and “bad” teachers simultaneously. L is *box-and-teacher-proof* if for any partial recursive function f , any presentation of f on the examples tape, any function box g , and any sequence $m_1, m_2, \dots$ of messages, $L(g)$ run with the given examples tape and sequence of messages as responses on the message tape does not converge incorrectly.

5. Complexity-bounded learning

Since we began with the issue of outright coding, it is instructive to examine a protocol for outright coding in this new setting. Consider the teacher T_0 that copies the contents of its answer tape to the message tape each time control is passed to T_0 . Consider the learner L_0 that initially passes control to the teacher, and, when control returns to the learner, copies the contents of the message tape to its guess tape and halts. Intuitively speaking, when T_0 and L_0 are run together, T_0 sends the index i to L_0 , and L_0 guesses i and halts.

Clearly, for every universal function box g , $L_0(g)$ learns every partial recursive function from $T_0(g)$. That is, if the function boxes of the learner and teacher are exactly the same, our model permits a straightforward version of outright coding. However, this protocol is very sensitive to changes in the learner's function box or in the teacher. In particular, for no universal function box g' are L and $T(g')$ box-proof and for no function box g is $L(g)$ teacher-proof.

However, by appealing again to the idea of using a computational complexity bound to guide learning, we can move away from using outright coding and achieve a learner and teacher pair such that the learner is box-and-teacher proof and learns all the partial recursive functions from the teacher, provided that the function box of the teacher is "not too much faster" than the function box of the learner. The notion of being "not too much faster" is formally specified using the concept of b -relatedness of two function boxes, defined in Section 3.1. The learner is denoted by L^* . The teacher is parameterized by the choice of the function b and is denoted by T_b^* .

Theorem 2. *There exists a learner L^* such that for every total recursive function $b(x, s)$ there is a teacher T_b^* such that for every universal function box g' and every function box g that is b -related to g' , $L^*(g)$ learns all the partial recursive functions from $T_b^*(g')$. Furthermore, L^* is box-and-teacher-proof.*

Proof: We fix a total recursive bounding function $b(x, s)$. For example, if $b(x, s) = 2s$ then the learner's program for f will be allowed to use at most twice as many steps as the teacher's program.

Let g' denote the teacher's function box, which is universal by assumption. By the assumptions of our model, the examples tape contains a presentation of the target function f and the answer tape contains an index i such that g'_i extends f . For any x in the domain of f , the teacher can make oracle calls to determine $g'(i, x, s)$ for $s = 0, 1, 2, \dots$ until it finds the minimum s for which $g'(i, x, s) \neq ?$, i.e., until it finds the value of the measure $G'_i(x)$.

The teacher $T_b^*(g')$ responds to a request from the learner about an entry (x, y) on the examples tape by determining the value of $G'_i(x)$ and placing the value of $b(x, G'_i(x))$ on the message tape. Note that if (x, y) is on the examples tape, x must be in the domain of f . Thus, for the pair (x, y) , T_b^* supplies the measure of its program for f on input x , increased by the bounding function b .

Assume the learner's function box is g . The learner $L^*(g)$ uses the pairs on the examples tape and the complexity information supplied by the teacher to eliminate possible hypotheses j such that g_j runs too long or gives the wrong value.


```

L*(g)
{
  For (n = 0; ; n++)                                /* Main Loop */
  {
    j =  $\pi_1(n)$ ;                                       /* View n as pair (j, m) */
    m =  $\pi_2(n)$ ;
    Write j + 1 and then j on guess tape;

    For (k = 0; ; k++)                                /* Testing Loop */
    {
      Get k-th pair (x, y) from examples tape;
      Request corresponding value z from teacher;

      For (s = 0; s ≤ max(m, z); s++)                  /* Bounded Loop */
      {
        If ((v = g(j, x, s)) ≠ ?)
        {
          Break;                                     /* Leave Bounded Loop */
        }

        If ((v ≠ y) or (s > max(m, z)))
        {
          Break;                                     /* Leave Testing Loop */
        }
      }
    }
  }
}

```

Figure 1. Algorithm for L^* .

The algorithm for L^* is shown in figure 1. In particular, at each iteration through the Main Loop, $L^*(g)$ chooses a hypothesis j and a dovetail value m , and writes the numbers $j + 1$ and then j on the guess tape. (The purpose of $j + 1$ is to guarantee a mind change.) In the Main Loop, the pairing function $n = \langle j, m \rangle$ gives all possible pairs of a conjecture j and a dovetail value m as n ranges over the natural numbers. $L^*(g)$ then enters the Testing Loop and begins to check the conjecture j and the dovetail value m against all the pairs (x, y) on the examples tape.

For each pair (x, y) , $L^*(g)$ makes a request to the teacher, who returns a number z on the message tape. The learner treats $z' = \max(z, m)$ as a running time bound for $g_j(x)$. That is, $L^*(g)$ makes oracle calls to determine $g(j, x, s)$ for $s = 0, 1, \dots, z'$, in the Bounded Loop. If $G_j(x) \leq z'$ and $g_j(x) = f(x)$, then $L^*(g)$ remains in the Testing Loop, retaining the hypothesis j and dovetail value m and searching for the next pair to check on the examples tape. Otherwise, $G_j(x) > z'$ or $G_j(x) \leq z'$ and $g_j(x) \neq f(x)$, and $L^*(g)$ breaks out of the Testing Loop and starts the next iteration of the Main Loop, which moves on to the next conjecture and dovetail value to be considered.

We now assume that the learner's function box g is b -related to the teacher's function box g' , and prove that $L^*(g)$ learns f from $T_b^*(g')$. Recall that if $n = \langle j, m \rangle$, then $\pi_1(n) = j$ and $\pi_2(n) = m$. There exists a natural number n that satisfies the following two conditions.

1. $g_{\pi_1(n)} \supseteq f$.
2. $G_{\pi_1(n)}(x) \leq \max(\pi_2(n), b(x, G'_i(x)))$, for all points $x \in \text{Dom}(f)$.

By hypothesis, the index i on the answer tape is such that $g'_i \supseteq f$. Since g is b -related to g' , there exists a j such that $g_j \supseteq g'_i \supseteq f$ and $G_j(x) \leq b(x, G'_i(x))$ for all but finitely many $x \in \text{Dom}(g'_i)$.

Since $g'_i \supseteq f$ we have that $\text{Dom}(f) \subseteq \text{Dom}(g'_i)$. Thus $G_j(x) \leq b(x, G'_i(x))$, for all but finitely many $x \in \text{Dom}(f)$. Let X be the set of points in $\text{Dom}(f)$ such that $G_j(x) > b(x, G'_i(x))$. Let $m = \max_{x \in X} \{G_j(x)\}$. Then it must be that $G_j(x) \leq \max(m, b(x, G'_i(x)))$ for all $x \in \text{Dom}(f)$. The choice of $n = \langle j, m \rangle$ satisfies the two conditions given above.

Let n_0 be the least n that satisfies these two conditions. Once variable n in the Main Loop of the algorithm reaches n_0 , the learner will output a correct hypothesis j and will never change it because it will never exit the Testing Loop. For all those $n < n_0$, however, the Testing Loop will eventually be exited and the Main Loop will be iterated with the next value of n . Thus, the outputs on the guess tape finitely converge to a value j such that g_j extends f , that is, $L^*(g)$ converges correctly.

This holds for every presentation of every partial recursive function f on the examples tape and every index i such that g'_i extends f on the answer tape. Therefore $L^*(g)$ learns all the partial recursive functions from $T_b^*(g')$. This in turn holds for every recursive bounding function $b(x, s)$ and every pair of function boxes g and g' such that g' is universal and g is b -related to g' , which proves the first assertion of the theorem.

What remains to be proved is that L^* is box-and-teacher-proof. Suppose $L^*(g)$ converges to an index j . We must prove that g_j extends the target function f . Because every iteration of the Main Loop guarantees a mind change (this was the purpose of writing $j + 1$ on the guess tape before j), $L^*(g)$ converges to j only if it stays in the Testing Loop forever. This happens only if the function g_j agrees with the target function f for all points x that appear as the first component of a pair (x, y) on the examples tape. But all points $x \in \text{Dom}(f)$ appear eventually as the first component of a pair on the examples tape, thus g_j extends f . $\square$

This positive result seems in some respects fairly modest. The learner L^* is box-and-teacher proof and is able to learn all the partial recursive functions from T_b^* , but only when its function box is “not too much slower” (as measured by b) than the teacher’s. This is certainly an improvement over the “outright coding” protocol, in which the learner is neither box-proof nor teacher-proof and the learner’s and teacher’s function boxes are assumed to be identical.

However, it might be that a much stronger positive result could be proved in this model, possibly by using a more elaborate version of outright coding. We therefore address the following question.

Is there a learner L , a teacher T , and a function box g' such that $L(g)$ learns all the partial recursive functions from $T(g')$ for any acceptable function box g ?

An affirmative answer to this question would cast serious doubt on the model we have defined. We now present two corollaries of Theorem 1; the second one gives a negative answer to this question.

Corollary 1. *There is no learner L , no teacher T and no function box g' for the teacher such that $L(g)$ learns the constant zero function Z from $T(g')$ for all acceptable function boxes g .*

Proof: If for some teacher T and some teacher's function box g' there exists such a learner L , then there exists a naive learner $L'(g)$ which simulates $T(g')$ and $L(g)$ and thus learns Z for all acceptable function boxes g , contradicting Theorem 1. $\square$

Corollary 2. *There is no learner L , no teacher T and no function box g' for the teacher such that $L(g)$ learns all the partial recursive functions from $T(g')$ for all acceptable function boxes g .*

Proof: If there were a learner L , a teacher T and a function box g' for the teacher such that for all acceptable function boxes g , $L(g)$ learns all the partial recursive functions from $T(g')$, then, in particular, $L(g)$ would learn the constant zero function from $T(g')$, contradicting Corollary 1. $\square$

This answers the question raised above and shows that the issue of outright coding does not trivialize the model. We note that these negative results do not depend on the teacher having only oracle access to its programming system. The teacher and its function box can be chosen arbitrarily (e.g., the programming system can be the standard Turing machine system), and still there is no learner that can cope with all acceptable function boxes.

6. Primitive recursive function boxes

Our general definition permits a function box to be any total recursive function of three arguments. This parallels the generality of Blum's definition of an abstract complexity measure for a programming system, and allows any programming system with a complexity measure to be represented as a function box. However, one cost of this generality is that the class of function boxes is not itself recursively enumerable.

Because our motivation is ultimately to gain insight into practical situations involving teaching and learning, we wish to consider restricted classes of function boxes. For example, a particular kind of robot might have an action space drawn from a rather limited set of possibilities.

In this section and the next we consider primitive recursive function boxes. The class of all such function boxes is recursively enumerable. Let $g^0, g^1, g^2, \dots$ be a fixed recursive enumeration of all 3-argument primitive recursive functions from $\mathbf{N}$ to $\mathbf{N} \cup \{?\}$. Thus g_i^k denotes the i -th function in the k -th primitive recursive function box, and G_i^k its measure.

Function boxes derived from many natural programming systems are primitive recursive, for example, the standard Turing machine function box is primitive recursive. We could take a somewhat more abstract approach and consider recursively enumerable classes of function boxes with certain additional properties, but we believe comprehensibility is aided by considering the concrete case of the primitive recursive functions.

One interesting consequence of this restriction is that the learner is able to find, in the limit, a primitive recursive program that is equivalent to its function box. This follows from the well-known result that any recursively enumerable class of total recursive functions can be identified in the limit by an inductive inference machine. This ability to gain a certain

kind of “self-knowledge” in the limit suggests that the restriction to primitive recursive function boxes might make the learner’s job considerably easier in general. However, in the next section we see that this optimism is not borne out.

The restriction to primitive recursive function boxes allows us to prove an approximate converse of Theorem 2. First we introduce some notation. Given a teacher T , learner L , and function box g' , let $G(L, T, g')$ denote the class of primitive recursive function boxes g such that $L(g)$ learns all the partial recursive functions from $T(g')$.

The following theorem says that if we have a box-proof pair L and $T(g')$, then there is some total recursive function b such that every primitive recursive function box g for which $L(g)$ can learn all the partial recursive functions from $T(g')$ is b -related to g' . That is, if $L(g)$ succeeds in learning all the partial recursive functions from $T(g')$, g cannot be “too much slower” than g' .

Theorem 3. *Let a teacher T , a learner L , and a function box g' be given. Assume that L and $T(g')$ are box-proof. Then there exists a total recursive function b , such that every function box g in $G(L, T, g')$ is b -related to g' .*

Proof: We show how to compute a total recursive function $b(x, s)$ such that every g in $G(L, T, g')$ is b -related to g' . As an intermediate step, we show how to compute a total recursive function $b(i, k, x, s)$ such that if $g^k \in G(L, T, g')$ then there exists a j such that g_j^k extends g'_i and $G_j^k(x) \leq b(i, k, x, G'_i(x))$ for all but finitely many x .

To see that this suffices to prove the theorem, define $b(x, s)$ to be the maximum of $b(i, k, x, s)$ over all $i \leq x$ and $k \leq x$. Then, for any $g \in G(L, T, g')$ there exists a k such that $g = g^k$, because all of the elements of $G(L, T, g')$ are primitive recursive. By the properties of $b(i, k, x, s)$, for every i there exists a j such that g_j^k extends g'_i and therefore g_j extends g'_i . Also, for all but finitely many x , $G_j^k(x) \leq b(i, k, x, G'_i(x))$. Because $b(i, k, x, s) \leq b(x, s)$ for all but finitely many x and $G_j(x) = G_j^k(x)$ for all x , $G_j(x) \leq b(x, G'_i(x))$ for all but finitely many x . Thus, every $g \in G(L, T, g')$ is b -related to g' .

The algorithm to compute $b(i, k, x, s)$ is shown in figure 2. The key idea of the algorithm is that given i and k , if $g^k \in G(L, T, g')$ and we run $L(g^k)$ and $T(g')$ with a presentation of g'_i on the examples tape and i on the answer tape, then the guesses of L must finitely converge to some j such that g_j^k extends g'_i . If we knew this j , we could simply define $b(i, k, x, s)$ to be the value of $G_j^k(x)$ when $s = G'_i(x)$, and 0 otherwise. There are two problems we must overcome in this approach: (1) we cannot be certain of the value of j at any finite time, and (2) for any particular k , g^k may not be in $G(L, T, g')$. These are handled using the box-proofness of L and $T(g')$ and the fact that the bound may be exceeded for finitely many x .

The subroutine **RUNTHEM** performs a simulation of the learner and teacher. When called with parameters T, g', i, L, k , and t , **RUNTHEM** simulates t steps of the computation of the learner $L(g^k)$ and the teacher $T(g')$ with a presentation of the function g'_i on the examples tape and i on the answer tape. After t simulated steps, **RUNTHEM** returns the last index output by the learner on the guess tape, or 0 if the learner has not output anything to the guess tape in the first t steps. The presentation of g'_i on the examples tape is enumerated according to a fixed computable dovetailing strategy. **RUNTHEM** clearly computes a total

```

B(i, k, x, s)
{
  b = 0;

  If (G'_i(x) == s)
  {
    j = RUNTHEM(T, g', i, L, k, x);

    For (z = 0; g^k(j, x, z) == ?; z++)          /* Loop */
    {
      j' = RUNTHEM(T, g', i, L, k, x + z);

      If (j' ≠ j)
        Break;                                   /* Leave Loop */
    }
    b = max(b, z);
  }
  Return b;
}

```

Figure 2. Algorithm for computing the bound $b(i, k, x, s)$.

recursive function, given programs for T , L , and g' and natural numbers i , k , and t as inputs.

Next we show that $b(i, k, x, s)$ is total recursive. It suffices to show that the Loop in the program $B(i, k, x, s)$ always terminates, either because $g^k(j, x, z) \neq ?$ or because $j' \neq j$. Assume to the contrary that for some values of i , k , x , and s , the Loop is entered but never exited.

Because the Loop is entered, the value of s must be $G'_i(x)$, which means that $g'_i(x)$ is defined. Each time through the Loop, z is incremented, so the fact that the condition $g^k(j, x, z) = ?$ is never violated means that $g_j^k(x)$ is undefined. Thus, g_j^k does not extend g'_i .

The call to `RUNTHEM` within the Loop allows one additional step of simulation at each iteration, so the fact that the returned value j' never differs from j means that the learner never changes its guess from j , that is, its guesses finitely converge to j . Hence the guesses finitely converge to a value j such that g_j^k does not extend g'_i , which contradicts the hypothesis that L and $T(g')$ are box-proof. Thus $b(i, k, x, s)$ is total recursive.

Now we prove that $b(i, k, x, s)$ gives the required bound. Assume that $g^k \in G(L, T, g')$. Then for any i , $L(g^k)$ and $T(g')$ run with any presentation of g'_i on the examples tape and i on the guess tape must result in the learner's guesses finitely converging to some j such that g_j^k extends g'_i . Thus, there are values j and t_0 such that `RUNTHEM` called with T , g' , i , L , k , and any $t \geq t_0$ returns the value j , and g_j^k extends g'_i .

Consider any $x \geq t_0$ such that $g'_i(x)$ is defined. Then for $s = G'_i(x)$, $B(i, k, x, s)$ will enter the Loop with the above value of j , and no j' returned by `RUNTHEM` in the Loop will differ from j . Because g_j^k extends g'_i and $g'_i(x)$ is defined, there must eventually be a first value of z such that $g^k(j, x, z) \neq ?$, namely, $z = G_j^k(x)$. This will be the value of z when the Loop is exited and also the value returned for $b(i, k, x, s)$. Therefore, $G_j^k(x) \leq b(i, k, x, G'_i(x))$ for all but finitely many x , as was claimed. $\square$

Combining this with Theorem 2, we have the result that learner and teacher pairs of the form L^* and $T_b^*(g')$ are as powerful in this setting as any box-proof pair L and $T(g')$.

Corollary 3. *Let a teacher T , a learner L and a function box g' be given. Assume that L and $T(g')$ are box-proof. Then there exists a total recursive function $b(x, s)$ such that $G(L, T, g') \subseteq G(L^*, T_b^*, g')$.*

Proof: Theorem 3 allows us to construct a bound b , which is exactly what is needed to apply Theorem 2. $\square$

We can derive a type of closure result using Corollary 3. (Minicozzi (1976) gives a variety of interesting closure results for the case of target-proof learning.)

Corollary 4. *Let L_1 and L_2 be learners and let T_1 and T_2 be teachers. Let g' be an arbitrary function box. Assume that L_1 and $T_1(g')$ are box-proof and L_2 and $T_2(g')$ are box-proof. Then there exists a total recursive function b such that*

$$G(L_1, T_1, g') \cup G(L_2, T_2, g') \subseteq G(L^*, T_b^*, g').$$

Proof: Corollary 3 implies that there is a total recursive function $b_i(x, s)$, such that $G(L_i, T_i, g') \subseteq G(L^*, T_{b_i}^*, g')$, for $i = 1, 2$. Let us define b to be the maximum of b_1 and b_2 , i.e., $b(x, s) = \max(b_1(x, s), b_2(x, s))$. Then we have that all function boxes g that are b_1 -related or b_2 -related to g' , are also b -related to it. The result follows. $\square$

7. Is everything easy now?

Recall that the restriction to primitive recursive function boxes allows the learner to find in the limit a primitive recursive program for its own function box. Also, the counterexample function box constructed in the proof of Theorem 1 is not necessarily primitive recursive. These considerations suggest the following question.

Does there exist a learner L , a teacher T , and a function box g' for the teacher such that $L(g)$ learns all the partial recursive functions from $T(g')$ for all primitive recursive acceptable function boxes g ?

Corollary 2 answered this question in the negative without the restriction to primitive recursive function boxes, by using Theorem 1. However, even with the restriction to primitive recursive function boxes, the answer is still “no.” This is shown by Corollary 6 below. Thus, even with the restriction to primitive recursive function boxes, our model is not trivialized by the issue of “outright coding.”

First we give a theorem that strengthens Theorem 1 for the case of primitive recursive function boxes. The proof is given in the Appendix.

Theorem 4. *There is no naive learner L such that for any primitive recursive acceptable function box g , $L(g)$ learns the constant zero function Z .*

We prove several corollaries of this theorem. The first two strengthen Corollary 1 and Corollary 2, respectively.

Corollary 5. *There is no learner L , no teacher T and no function box g' for the teacher such that the learner $L(g)$ learns the constant zero function Z from the teacher $T(g')$ for all primitive recursive acceptable function boxes g .*

Proof: Assume to the contrary that such a learner L , teacher T and function box g' exist. Then we construct a naive learner L' which when run with function box g , simulates $L(g)$ and $T(g')$ with a presentation of the constant zero function Z on the examples tape, and an index i such that $g'_i = Z$ on the answer tape, and copies $L(g)$'s guesses to its own guess tape. The guesses of $L(g)$ must finitely converge to an index j such that $g_j = Z$. Thus $L'(g)$ learns Z for all primitive recursive acceptable function boxes g , contradicting Theorem 4. $\square$

Corollary 6. *There is no learner L , teacher T and function box g' such that the learner $L(g)$ learns all the partial recursive functions from the teacher $T(g')$ for all primitive recursive acceptable function boxes g .*

Proof: Learning all the partial recursive functions would entail learning the constant zero function Z , which would contradict Corollary 5. $\square$

Corollary 7. *There is no total recursive function b and no function box g' such that $L^*(g)$ learns all the partial recursive functions from $T_b^*(g')$, for all primitive recursive acceptable function boxes g .*

Proof: This is a special case of Corollary 6. $\square$

We note that each of the three preceding corollaries holds in the case that the teacher uses a fixed, known programming system like the standard Turing machine system.

Corollary 8. *For every universal function box g' and for every total recursive function $b(x, s)$ there exists a primitive recursive acceptable function box g which is not b -related to g' .*

Proof: Suppose to the contrary that there exists a universal function box g' and a total recursive function $b(x, s)$ such that every primitive recursive acceptable function box is b -related to g' . Then, applying Theorem 2, for every function box g that is b -related to g' , $L^*(g)$ learns all the partial recursive functions from $T_b^*(g')$. In particular, for this b , $L^*(g)$ learns all the partial recursive functions from $T_b^*(g')$ for every primitive recursive acceptable function box g , because of our assumption that they are all b -related to g' . This contradicts Corollary 7. $\square$

We observe that for every acceptable function box g' and every total recursive function $b_1(x, s)$, there is some “bad” but primitive recursive and acceptable function box g'' such

that $L^*(g'')$ cannot learn all the partial recursive functions from $T_{b_1}^*(g')$. However, because g'' is acceptable, it is b_2 -related to g' for some (larger) total recursive function $b_2(x, s)$, which means that $L^*(g'')$ can learn all the partial recursive functions from $T_{b_2}^*(g')$. However, for b_2 there is another “bad” but primitive recursive and acceptable function box g''' , for which $L^*(g''')$ cannot learn all the partial recursive functions from $T_{b_2}^*(g')$. Continuing in this way, there exists an infinite sequence of “worse and worse” but still primitive recursive and acceptable function boxes for the learner L^* .

8. Discussion

We have introduced a new model of a *naive learner* with a *function box* that provides only black-box access to the programming system in which the learner must frame its hypotheses. Theorem 1 shows that such a learner cannot succeed in finding an index for the constant zero function in every acceptable programming system.

We then introduce a teacher into the model, and ask which pairs of learners and teachers are successful in teaching every partial recursive function to the learner. Theorem 2 gives a very general class of such pairs, in which the teacher uses its own performance on the target function to calculate a running time bound for the learner. The learner uses the bound to prune unsuccessful searches for a program in its function box. Provided the learner’s function box is “not too much slower” than the teacher’s, the learner will succeed in finding an index for an extension of the target function. Estimated solution times can be powerful heuristics in human problem solving. For example, the information that solving a certain problem should take no more than thirty seconds allows me to ignore strategies that would take very much longer. The resulting pairs of learner and teacher embody a kind of “constructivist” theory of learning, in the sense that the learner uses very general guidance from the teacher to construct its own hypothesis, which may be quite different from the teacher’s. The strategy also has a built-in robustness against problems with the teacher or the learner’s function box.

By restricting attention to primitive recursive function boxes and requiring robustness against problems with the learner’s function box, we show in Theorem 3 that this bounding strategy is in a certain sense as powerful as any other strategy for pairs of learners and teachers. Finally, in Theorem 4 and its corollaries we show that the restriction to primitive recursive function boxes does not trivialize the learning problem.

Several directions and questions are suggested by this research. One is how to weaken the rather strong assumptions made by the learner and teacher about their protocols for communication. Another question is how the teacher can provide more specific guidance than just a bound on running time, while still avoiding “outright coding.” An interesting, important, and probably difficult direction is to attempt to translate some of the issues and insights from this work into the domain of polynomial-time learning. Another direction is to allow function boxes to contain arbitrary, not necessarily computable, three-argument total functions. The contents of the examples tape would be a presentation of one of the functions in the teacher’s function box, and the other basic definitions would remain essentially the same. In this setting it appears that an analog of Theorem 2 holds, but it is not clear what else might be true. The gap between the (computable) cognitive space and the (potentially

uncomputable) action space might provide interesting analogies to the opacity of some of our own “black box” capabilities.

Many questions remain open even in the present framework. Most of our results concern learning the class of all partial recursive functions in the limit; smaller target classes and more restricted versions of b -relatedness are natural directions to consider. Questions about the relationships between target-proofness, box-proofness, and teacher-proofness, and combinations of them (e.g., box-and-teacher-proofness) are another possible direction. For example, there might be classes of function boxes for the learner and teacher such that some class of target functions can be learned in a box-proof and teacher-proof manner but not in a box-and-teacher-proof manner.

Appendix

In this section we give a proof of Theorem 4. Let L be any naive learner, and fix any presentation of the constant zero function on the input tape of L . We construct a primitive recursive acceptable function box g such that $L(g)$ does not correctly identify the constant zero function with this input.

The construction of g is intertwined with a simulation of $L(g)$. We describe the construction of g as an infinite process of defining the entries in the initially completely undefined table $g(i, x, s)$, and divide the process into stages $n = 1, 2, 3, \dots$. The standard Turing machine system is presented by the function box $t(i, x, s)$. Let $k(n) = n(n + 1)/2$, for which $k(n) - k(n - 1) = n$. The variable h holds the most recent guess of L , initialized to -1 .

The procedure *Define* takes i, x, s , and h as input and ensures that the entry for $g(i, x, s)$ is defined, as follows. If $g(i, x, s)$ is already defined, then *Define* does nothing. If $g(i, x, s)$ is not already defined, then *Define* defines it to be $?$ if $i = h$ or $t(i, x, s)$ if $i \neq h$.

In stage n of the process, call *Define* for all tuples i, x, s , and h , where h is the current guess of L , $0 \leq i \leq n$, $0 \leq x \leq n$, and $0 \leq s \leq k(n)$. Then simulate L on the fixed input tape for one more step. If L makes an oracle call to $g(i, x, s)$, then call *Define* for i, x, s , and the current value of h and return the value of $g(i, x, s)$. If L makes a guess j , then set the value of h to j . After simulating one step of L , go to stage $n + 1$.

Then g is total recursive because the entry $g(i, x, s)$ is defined by the end of stage m , where $m = \max\{i, x, s, 1\}$. All the loops involved are bounded, so the computation is primitive recursive, and g is a primitive recursive function box. For all triples of natural numbers i, x, s either $g(i, x, s) = t(i, x, s)$ or $g(i, x, s) = ?$. Now consider the behavior of $L(g)$.

If $L(g)$ converges to an index j_0 then the function g_{j_0} is undefined except possibly at finitely many values of x . To see this, note that if $L(g)$ converges to j_0 then there exists n_0 such that $h = j_0$ for all stages $n \geq n_0$. Up to stage n_0 , at most finitely many entries $g(j_0, x, s)$ are defined, and all subsequent ones will be defined to be $?$.

If i is any index to which $L(g)$ does not converge then $g_i = t_i$. To see this, assume that $L(g)$ does not converge to i and let x be any natural number. If t_i is undefined on x , then $g(i, x, s) = ?$ for every natural number s , and g_i is undefined on x . If $t_i(x) = y$, then for all sufficiently large s , $t(i, x, s) = y$. Because $L(g)$ does not converge to i , there are arbitrarily

large n for which $i \neq h$ at stage n . Thus there exists an n_0 such that $n_0 \geq i$, $n_0 \geq x$, $t(i, x, s) = y$ for all $s \geq n_0$, and $i \neq h$ at stage n_0 . At stage n_0 , *Define* is called with (among others) inputs i , x , and $s = k(n_0 - 1) + 1, \dots, k(n_0)$. The only way these entries can be defined already is if $L(g)$ has made oracle calls to them in previous stages. The total number of oracle calls in stages up to n_0 is at most $n_0 - 1$ so for at least one of the n_0 values of s , the entry $g(i, x, s)$ is defined to be $t(i, x, s)$, which is equal to y . Thus at least one entry for i and x equals y rather than $?$, so $g_i(x) = y = t_i(x)$. Hence, $g_i = t_i$, as claimed.

To see that g is acceptable, we argue as follows. If $L(g)$ does not converge to any index, $g_i = t_i$ for all i , and the function box g contains the standard Turing machine system, which is acceptable. If $L(g)$ converges to j_0 , then $g_i = t_i$ for all $i \neq j_0$, and g_{j_0} is undefined except at finitely many points. It is not difficult to see that in this case also $g_0, g_1, g_2, \dots$ is an acceptable system.

Finally, to see that $L(g)$ fails to identify the constant zero function, note that either $L(g)$ fails to converge to any index or it converges to an index j_0 for an almost everywhere undefined function, which is not an index of the constant zero function. This concludes the proof of Theorem 4.

Acknowledgments

An earlier version of this paper appeared in the 1997 COLT proceedings (Angluin & Krikis, 1997). This research was supported in part by the National Science Foundation, grants CCR-9213881 and CCR-9610295. We thank the anonymous referees for several helpful suggestions, including the simplified proof of Theorem 4 that appears in the Appendix.

Notes

1. This would permit the exploration of subrecursive action spaces, a direction we have not pursued in this paper.
2. This is the case considered by Blum and Blum (1975) and Minicozzi (1976).

References

- Angluin, D., & Krikis, M. (1997). Teachers, learners and black boxes. In *Proc. 10th Annu. Conf. on Comput. Learning Theory* (pp. 285–297) New York, NY: ACM Press.
- Blum, L., & Blum, M. (1975). Toward a mathematical theory of inductive inference. *Inform. Control*, 28:2, 125–155.
- Blum, M. (1967). A machine-independent theory of the complexity of the recursive functions. *Journal of the ACM*, 14, 322–336.
- Gold, E. M. (1967). Language identification in the limit. *Inform. Control*, 10, 447–474.
- Goldman, S. A., & Kearns, M. J. (1991). On the complexity of teaching. *J. of Comput. Syst. Sci.*, 50:1, 20–31.
- Goldman, S. A., & Mathias, H. D. (1996). Teaching a smarter learner. *J. of Comput. Syst. Sci.*, 52:2, 255–267.
- Jackson, J., & Tomkins, A. (1992). A computational model of teaching. In *Proc. 5th Annu. Workshop on Comput. Learning Theory* (pp. 319–326) New York, NY: ACM Press.
- Krikis (1998). Learning from different teachers and imperfect queries. Ph.D. thesis, Yale University.
- Machtey, M., & Young, P. (1978). *An introduction to the general theory of algorithms*. Theory of Computation Series. Elsevier North-Holland, Inc.

- Mathias, D. H. (1995). DNF—if you can't learn 'em, teach 'em: An interactive model of teaching. In *Proc. 8th Annu. Conf. on Comput. Learning Theory* (pp. 222–229). New York, NY: ACM Press.
- Minicozzi, E. (1976). Some natural properties of strong identification in inductive inference. *Theoret. Comput. Sci.*, 2, 345–360.
- Shinohara, A., & Miyano, S. (1991). Teachability in computational learning. *New Generation Computing*, 8, 337–347.
- Smith, C. H. (1994). *A recursive introduction to the theory of computation*. New York: Springer-Verlag.

Received June 9, 1999

Revised July 13, 2001

Accepted July 13, 2001

Final manuscript July 13, 2001