The Functions of Finite Support: a Canonical Learning Problem [1]

Rūsiņš Freivalds[2]
Institute of Mathematics and Computer Science
University of Latvia, Raiņa bulvāris 29, LV-1459, Riga, Latvia and
Efim Kinber
Department of Computer and Information Science
Sacred Heart University, 5151 Park Avenue Fairfield, CT 06432, USA and
Carl H. Smith[3]
Department of Computer Science
University of Maryland, College Park, MD 20912, USA

Abstract

The functions of finite support have played a ubiquitous role in the study of inductive inference since its inception. In addition to providing a clear and simple example of a learnable class, the functions of finite support are employed in many proofs that distinguish various types and features of learning. Recent results show that this ostensibly simple class requires as much space to learn as any other learnable set and, furthermore, is as intrinsically difficult as any other learnable set. This makes the functions of finite support a candidate for being a canonical learning problem. We argue for this point in the paper and discuss the ramifications.

Introduction

The starting point for studies in inductive inference is the model of learning by example introduced by Gold (1967). This is a simple model of learning algorithms that input examples of some function and produce programs that are intended to compute the function generating the examples. Learning takes place as the "correct" program must be produced after the learning algorithm has seen only finitely many examples. The functions used as input are typically (partial) recursive functions. Using traditional encoding techniques, this class of functions is rich enough to model a wide range of phenomena (Angluin & Smith, 1983). Similar models were used by philosophers of science who were interested in understanding the scientific method (Burks, 1958; Carnap, 1952; Carnap & Jeffrey, 1971; Popper, 1968; Putnam, 1975).

The mathematically rich study of inductive inference proceeds by defining variations of Gold's model and showing that they give rise to a different class of learnable sets of functions. For example, the study of team inference (Smith, 1994b) was started by the early result that there are two sets of functions, each of which is learnable, but their union is not (Blum & Blum, 1975). The functions of finite support were one of the two sets used. Subsequently, this example set was used many times in the study of inductive inference to distinguish between two types of learnability.

[1]This work was facilitated by an international agreement under NSF Grants 9119540 and 9421640

[2]Supported by Latvian Science Council Grant No.93.599.

[3]Supported in part by NSF Grants 9020079 and 9301339 and the Fulbright Commision. Currently on leave at the University of Kaiserslautern.

One of the criticisms of work in inductive inference is that the examples used are artificial, or in the case of the functions of finite support, overly simplistic. The ubiquitous use of, and the simplicity of the functions of finite support make them a candidate for being a "canonical" learning problem. Recent results (Freivalds et al., 1995, Freivalds, Kinber & Smith, 1995) concerning various notions of complexity for learning strongly suggest that the reason for the numerous appearance of the functions of finite support in the literature is because they play a role for learning analogous to the role played by the CNF satisfiability problem for general complexity theory (Garey & Johnson, 1979).

In what follows, we present the Gold model in greater detail and then review the recent results alluded to above. In an effort to minimize notation, formal definitions have been placed in the appendix. We conclude with a deeper analysis of the functions of finite support with respect to learning.

The Gold Model

Virtually all learning by example models are enhancements or restrictions of Gold's notion of learning in the limit (Gold, 1967). In this model, an algorithmic device takes input examples representing the graph of a function (or strings in a language) and, from time to time, outputs programs. If the input examples will exhaust the set of all possible examples and there is a single program that is almost always the program produced by the device and this program computes the function represented by the input, then we say that the device has learned the function. Clearly, some type of learning must have taken place as the correct function is produced after the device has seen only finitely many of the infinitely many examples. In this way, each algorithmic device will learn a certain (undecidable) set of recursive functions. The collection of all such sets, across all algorithms, gives rise to the collection of all learnable sets of recursive functions.

This model is called "learning in the limit" as it is never known when the correct program has first appeared as an output of the learning algorithm. In real life situations, one frequently never knows when they have completely learned something. For example, technology frequently introduces new words into the language. Is anyone willing to say they know *all* of any natural language? While many of us claim to be expert drivers, relatively few drivers know about such fine points as "heal-toe maneuvers" or "4 wheel drifts." In some

cases, such as learning single digit multiplication tables, it is easy to determine when the learning is complete. Such cases have been studied in the inductive inference community, and they represent another example of a restriction of the basic Gold model of learning in the limit.

If the inputs for the Gold model are drawn from some distribution, the learning device is constrained to operate within some complexity bounds, and the program produced by the device need not be correct on every input, but just often enough with respect to the distribution of inputs and a function of the number of examples seen, then the PAC model of learning (Valiant, 1984) results. Similarly, any model of learning by example can be induced by restricting or augmenting the basic Gold model.

We proceed with an example. Below, we describe an algorithm that will learn all the functions of finite support. These are the recursive functions that take the value zero on all but finitely many inputs. While the name of this class of functions comes from the branch of Mathematics known as Analysis, the first use in computer science of a recursive version of the functions of finite support appears to be in (Meyer, 1972).

The algorithm A we have in mind starts by initializing a table T to be null and outputs a program for the everywhere zero function. A then proceeds to perpetually execute the following three steps:

1. Read another input pair (x, y),
2. Add (x, y) to T iff $y \neq 0$ (indicating that another "support" point has been located), and
3. Output a program p described below. If the table T has not changed, the program produced as output is identical to the most recent previously produced program.

   p has a copy of T and, on input x will output y such that the pair (x, y) is in T and output 0 if there is no such pair in the current table T.

It remains to be verified that A works correctly on any function of finite support. Since A outputs a new program only when the table changes, and that happens only when a nonzero range point of the function is found, convergence to a single program will be achieved. Furthermore, this final program will have all the support points in the table, as otherwise it wouldn't be the final program. Finally, this last program is correct, as all the non zero points are in the table and this is consulted first by every program that is produced by A. If the value is not found in the table, then the input cannot map to a support point and a 0 is the value returned by the final program.

Gold was primarily interested in language learning. Via a standard encoding of strings of symbols as natural numbers, a *language* can be viewed as a function that maps encodings of strings in the language to 1 and all other inputs to 0. Gold (1967) proved that any class containing all the finite languages and one infinite language could not be learned. Of course, when the finite languages are viewed as functions, they become the functions of finite support. As mentioned earlier, the functions of finite support were used to prove the non union theorem of (Blum & Blum, 1975). The functions of finite support have infinite VC dimension, so they cannot be PAC learned. However, using dynamic sampling, the concept class of all finite subsets of the natural numbers (essentially the functions of finite support) can be PAC learned (Linial, Mansour & Rivest, 1991).

Learning with a Limited Memory

The observation that humans learn without the benefit of a perfect memory has stimulated researchers in many fields to consider some form of memory restricted learning. For example, in the field of neural modeling, it has been suggested that one of the functions of rapid eye movement (REM) sleep is to discard some memories to keep from overloading our neural networks (Crick & Mitchison, 1983). Independent simulations have verified that occasional "unlearning" aids in learning (Hopfield, Feinstein & Palmer, 1983). In a similar vein, neural networks with a limitation on the type of the weight in each node were considered in (Siegelmann and Sontag, 1992). The types considered are integer, rational and real. Each successive type can, potentially, place higher demands on memory utilization within each node. Each type also expands the inherent capabilities of the neural networks using that type of node weights.

Linguists have also considered the effect of memory for learning. Braine (1971) suggested that human memory is organized as a cascading sequence of memories. The idea is that items to be remembered are initially entered in the first level of the memory and then later moved to successive levels, finally reaching long term memory. In Braine's model, each of the transitionary memory components are subject to degradations. Consequently, items to be remembered that are not reinforced by subsequent inputs may be eliminated from some level of the memory before they become permanently fixed in memory. Wexler and Culicover (1980) formalized many notions of language learning, including one where the learning algorithm was to have access to the most recently received data and the machines' own most recent conjecture. Their model was generalized in Osherson, Stob & Weinstein, 1986) to allow the learning mechanism access to the last n conjectures as well as the most recently received data item. This generalization was shown not to increase the potential of such mechanisms to learn languages.

Within the study of inductive inference, there have been similar investigations of learning algorithms that do not remember all the data they have seen, nor remember all the programs that they have produced as outputs (Jantke & Beick, 1981; Miyahara, 1987; Wiehagen, 1976). Similar results were obtained. A different approach to memory limited learning was investigated in (Heath et al., 1991). The issue addressed in their work is to calculate how many *passes* through the data are needed in order to learn.

All off the above mentioned formal approaches restrict the number of items to be retained in memory without addressing

the *size* of those items. Since standard encoding techniques can be used to represent many items as a single integer, space utilization is not accurately addressed by this work. Lin and Vitter consider memory requirements for learning sufficiently smooth distributions (Lin & Vitter, 1994). Since they assume that the inputs are in some readable form, the issue of how much space it takes to store a number never arises. An attempt to rectify this difficult was made by researchers working with the PAC (probably approximately correct) model of learning (Valiant 1984). However, their results were not a true measure of space complexity because they measured space utilization as a function of the size of the smallest *answer*, not the size of the input (Boucheron & Sallantin, 1988; Haussler, 1985). PAC learning while remembering only a fixed number of examples, each of a bounded size is considered in (Ameur et al., 1993; Floyd, 1989; Helmbold, Sloan & Warmuth 1989). The most general investigation on this line was the observation in (Schapire, 1990) that the boosting algorithm can be made reasonably space efficient as well.

Sample complexity gives only a very crude accounting of space utilization. Learning procedures may want to remember other information than just prior examples. For example, all algorithms are based on some underlying finite state device. The states of the underlying finite state machine can also be used as a form of long term memory. To see this point, consider the standard example from automata theory of the finite state automaton that accepts all strings of length 0 mod 3. This machine has only three states, the start state s_0 and s_1 and s_2. Every symbol read forces a change of state from either s_0 to s_1, or from s_1 to s_2 or from s_2 to s_0. The only accepting state is s_0. In effect, state s_i $(i \leq 2)$ "remembers" that the input string seen so far is of length i mod 3. This technique can be applied to learning algorithms. To encode a single bit in the state space, just double the number of states. The second copy of the original state set behaves the same way as the original. If the machine is in the original state space, then a "0" is being remembered. If the machine is in the new state space, then a "1" is being remembered. In this fashion, an arbitrary amount of information may be remembered in the state space of an algorithm. The size of the algorithm may become enormous, but the amount of storage used by the algorithm may still be miniscule as measured by the techniques mentioned above. Hence, even the sample complexity metric neglects to count some of the long term storage employed by learning algorithms.

All of the above mentioned shortcomings of memory limit learned were address in the model introduced in (Freivalds, Kinber & Smith, 1995). There, memory utilization was measured as a function of the number of bits of input seen so far. Both long and short term memory were considered. The short term memory is erased every time the learning algorithm produces an output or reads another data item. All the state information, retained data, remembered outputs, and whatever is kept in the long term memory. It is assumed that each learning algorithm receives its input in such a way that it is impossible to back up and reread some input after another has been read. This model eliminates the possibility of coding lots of data into a single location as it is the number of bits of memory that is counted. Furthermore, if the state space is used to remember something, then this space is also counted as the state information is also kept in the long term memory. Finally, this model is a true complexity theoretic accounting of space utilization as the reckoning is done as a function of the size of the input.

One of the results that was shown in (Freivalds, Kinber & Smith, 1995a) is that if some set S is learnable, then it is learnable by an algorithm that uses linear long storage. In fact, the linear storage function that comes out of the proof is $c + (1 + \epsilon) \cdot n$, where n is the number of bits of input, c is a constant large enough to accommodate the state information of a universal simulator and ϵ is arbitrarily small. Hence, the space complexity world for learning is very compact. Everything lies between constant and linear space. The idea of the proof is that the factor n storage is used to remember *all* the data seen as input and the $\epsilon \cdot n$ factor is used by the universal simulator to simulate that operation of the learning algorithm that is purported to exist by the fact that S was learnable. As n grows large enough, the $\epsilon \cdot n$ factor is sufficient to run the simulation.

Several lower bound results were proven in (Freivalds, Kinber & Smith, 1995a). In particular, a linear lower bound on long term memory was shown for learning the functions of finite support. The idea of this proof is that if the input comes as a bit string representing the range of an initial segment of some function of finite support, then the entire bit string must be remembered. If not, then there would be two different bit strings, representing initial segments of two different functions of finite support, that would drive the learning algorithm into the same memory state. Hence, any algorithm using less than linear long term memory would converge to the same answer for two different functions of finite support. Hence, this alleged learning procedure is erroneous.

Intrinsic Complexity

The complexity of learning has been addressed in formally with the PAC model (Valiant 1984) and Angluin's teacher/learner model (Angluin, 1988). These studies carefully counted the resources used by various learning algorithms. However, the complexity of learning algorithm is different from the complexity of the learning task.

To see this, consider the following example. We will give an algorithm to learn all the polynomial time computable functions. The algorithm uses the enumeration technique of (Gold 1967). As a preliminary step, define an enumeration of all and only the linear time computable functions. The i^{th} program in this enumeration interprets i as an ordered pair (j, k), and runs the j^{th} Turing machine (from some standard list) on any input x for $k \cdot x + k$ steps. The learning algorithm initially guesses the first program in its enumeration and then starts reading data. When some data is input that disagrees with output of the current guess, the learning algorithm con-

siders the next program in its list that does not contradict the input it has seen so far. A moments reflection is all that is necessary to realize that, given input from some linear time computable function, this procedure will, in the limit, converge to a correct program.

In contrast with the simple learning procedure described above, consider learning the exponential time computable functions. In fact, the same basic algorithm works. The only modification necessary is to change the simulation time bound to something like $k^x + k$. This trivial modification leaves the essence of the algorithm unchanged. It is hard to argue that the modified algorithm is more difficult to understand or create. The learning process represented by the original algorithm and the modified one are essentially the same. However, due to the cost of exponential simulations compared with linear simulations, the complexity of the algorithm above for learning the exponential time computable functions is much greater than its complexity when learning the linear time functions.

In (Freivalds, Kinber & Smith 1995) a natural notion of reductions between learning problems was presented. The basic idea of the reduction is to map the examples from one concept into suitable examples to another concept that you know how to learn. Then, apply the known learning algorithm to the transformed examples and map the conjectures produced by the algorithm back to suitable conjectures for the original concept domain. The formal definition is a bit more intricate as "suitable" needs careful elaboration. Please refer to the appendix for these details.

It turns out that the learning of the polynomial time computable functions and the learning of the exponential time computable functions are reducible to each other (Freivalds, Kinber & Smith, 1995b). So, by the notion of complexity of learning problems induced by the natural notion of reduction, these two learning problems are equivalent. It is also the case that the functions of finite support are *complete* with respect to the notion of reduction (Freivalds, Kinber & Smith, 1995b). This means that a set of recursive functions S is learnable iff there are two mappings, one that transforms every function in S into a function of finite support, and one that transforms the programs that result from giving the transformed function as input to an algorithm that learns the functions of finite support into suitable programs for functions in S. Another view is that anything that makes learning difficult, but not impossible, is somehow reflected in the functions of finite support. This point is elaborated on in the next section.

The Functions of Finite Support Revisited

We have seen in the previous sections that the functions of finite support are a surprisingly complex learning problem with respect to space utilization and reductions from other learning problems. Since learning the functions of finite support appears so simple, this must be a canonical learning problem. In this section we attempt to analyze the learning of the functions of finite support to see what all the components of learning are.

Recalling the algorithm to learn the functions of finite support, we see that the algorithm has one large loop, the first step of which is to request more data. After some new data arrives, a decision must be made as to whether or not it is important and/or relevant. In the case of learning the functions of finite support, this decision is trivial, a simple test for 0. In general, a relevancy decision may be more complicated. However, for any learning problem, if the problem is solvable then there must be a computable algorithm to decide relevance of new data.

The next component of the algorithm to learn the functions of finite support is the production of another program conjectures to compute the function represented by the input. Here again, what happens is very simple. A program is produced that is correct *assuming* that all the support points have already occurred in the input. Briefly, if all the relevant data is present, then the program produced is correct. In general, there must be an effective way to generate a correct program from *all* the relevant data. The algorithm to learn the functions of finite support is correct because, in part, all the correct data will arrive by some point. Since all learning by example takes place by generalizing from a finite set of "relevant" examples, for learning to happen at all there must be some sub algorithm that transforms the set of "all relevant data" into a correct program.

The last observation to make is that while learning the functions of finite support, one is never quite sure if all the relevant data (points of support) have been seen as input. No matter what finite set of data has been observed, there are infinitely many different functions of finite support that are consistent with that data. If it were not for this uncertainty, all learning by example that was possible would be rote learning, or simple memorization.

In summary, there are three main points: **1.** The relevance of each data item must be effectively determined, **2.** The correct answer must obtainable uniformly from the complete set of relevant examples, and **3.** There is no way to effectively determine when the complete set of relevant examples has been observed.

Without the first two points, there can be no learning by computers. The third point about the indeterminacy of completion separates rote learning from the more interesting learning scenarios. Learning of the *self describing functions* provides an example matching the only the first two points. This class of functions contains all those functions that, on argument 0, evaluate to a complete description of a program for the function. At first, this class may sound very contrived. However, since any learning requires the existence of a finite set of data that enables the learning, the self describing functions are merely those functions where this data set is concisely encoded in a single point. Furthermore, this example represents the minimal size set of relevant examples and the most trivial uniform procedure to turn the set of relevant examples into a correct program. In the Introduction, the existence of two learnable sets whose union was not learnable

was mentioned. The functions of finite support was one of the two sets, the self describing functions are the other. Note that since the relevant data set is a single point, it is very easy to know when all the relevant data has been received. The set of self describing functions is *not* complete (Freivalds, Kinber & Smith, 1995b).

Conclusions

A very simple learning problem, that of learning the functions of finite support was examined in detail. This class can be learned, but only with requirements on memory utilization that match the theoretical maximum. Furthermore, the set is complete with respect to a very natural notion of reduction between learning problems. The functions of finite support are an ideal canonical learning problem. The straight forward procedure to learn the functions of finite support is the clearest schema possible. All of the fundamental components of learning are present, in the simplest form in this example. These components are: assessing relevance of examples, turning the complete set of relevant examples onto a solution, and lack of certainty about when the complete set of relevant examples has been seen.

Acknowledgements

This paper was written while the third author was on sabbatical leave at the University of Amsterdam. The support provided by the University and the NWO in The Netherlands is gratefully acknowledged. Rob Schapire pointed out the finite set result of (Linial, Mansour & Rivest, 1991) to us.

References

Ameur, F., Fischer, P., Höffgen, K., and auf der Heide, F. (1993). Trial and error: a new approach to space-bounded learning. In *Proceedings of the 1st European Conference on Computational Learning Theory*, pages 133–144. Oxford University Press.

Angluin, D. (1988). Queries and concept learning. *Machine Learning*, 2:319–342.

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

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

Boucheron, S. and Sallantin, J. (1988). Some remarks about space-complexity of learning, and circuit complexity of recognizing. In Haussler, D. and Pitt, L., editors, *Proceedings of the 1988 Workshop on Computational Learning Theory*, pages 125–138, San Mateo, CA. Morgan Kaufmann.

Braine, M. D. S. (1971). On two types of models of the internalization of grammars. In Slobin, D. I., editor, *The Ontogenesis of Grammar*, pages 153–186. Academic Press.

Burks, A. W. (1958). *Collected Papers of Charles Sanders Peirce*, volume 7. Harvard University Press, Cambridge, Mass.

Carnap, R. (1952). *The Continuum of Inductive Methods*. The University of Chicago Press, Chicago, Illinois.

Carnap, R. and Jeffrey, R. (1971). *Studies in Inductive Logic and Probability*. University of California Press, Berkeley, California.

Crick, F. and Mitchison, G. (1983). The function of dream sleep. *Nature*, 304(14):111–114.

Floyd, S. (1989). Space-bounded learning and the Vapnik-Chervonenkis dimension. In Rivest, R., Haussler, D., and Warmuth, M., editors, *Proceedings of the 1989 Workshop on Computational Learning Theory*, pages 349–364. Morgan Kaufmann.

Freivalds, R., Kinber, E., and Smith, C. H. (1995a). On the impact of forgetting on learning machines. *Journal of the ACM*, 42(6):1146–1168.

Freivalds, R., Kinber, E. B., and Smith, C. H. (1995b). On the intrinsic complexity of learning. *Information and Computation*, 123(1):64–71.

Garey, M. and Johnson, D. (1979). *Computers and Intractability: a Guide to NP- Completeness*. W. H. Freeman & Co., San Francisco, Calif.

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

Haussler, D. (1985). Space efficient learning algorithms. Technical report, University of California at Santa Cruz. UCSC-CLR-88-2.

Heath, D., Kasif, S., Kosaraju, R., Salzberg, S., and Sullivan, G. (1991). Learning nested concept classes with limited storage. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 777–782.

Helmbold, D., Sloan, R., and Warmuth, M. (1989). Learning nested differences of intersection-closed concept classes. In Rivest, R., Haussler, D., and Warmuth, M., editors, *Proceedings of the 1989 Workshop on Computational Learning Theory*, pages 41–56. Morgan Kaufmann.

Hopfield, J. J., Feinstein, D. I., and Palmer, R. G. (1983). 'unlearning' has a stabilizing effect in collective memories. *Nature*, 304(14):158–159.

Jantke, K. P. and Beick, H. R. (1981). Combining postulates of naturalness in inductive inference. *Electronische Informationsverabeitung und Kybernetik*, 17:465–484.

Lin, J. H. and Vitter, J. S. (1994). A theory for memory-based learning. *Machine Learning*, 17(2):143–168.

Linial, N., Mansour, Y., and Rivest, R. L. (1991). Results on learnability and the Vapnik-Chervonenkis dimension. *Information and Computation*, 90(1):33–49.

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

Meyer, A. (1972). Program size in restricted programming languages. *Information and Control*, 21:382–394.

Miyahara, T. (1987). Inductive inference by iteratively working and consistent strategies with anomalies. *Bulletin of Informatics and Cybernetics*, 22:171–177.

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

Popper, K. (1968). *The Logic of Scientific Discovery*. Harper Torch Books, N.Y.

Putnam, H. (1975). Probability and confirmation. In *Mathematics, Matter and Method*, volume 1. Cambridge University Press.

Rescher, N. (1970). *Scientific Explanation*. The Free Press, New York.

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

Schapire, R. (1990). The strength of weak learnability. *Machine Learning*, 5(2):197–227.

Schilpp, P. (1963). *Library of Living Philosophers: the Philosopy of Rudolph Carnap*, volume 11. Open Court Publishing Co., LaSalle, IL.

Siegelmann, H. T. and Sontag, E. D. (1992). On the computational power of neural nets. In *Proceedigns of the Fifth Annual ACM Workshop on Computational Learning Theory*, pages 440–449. AMC.

Smith, C. (1994a). *A Recursive Introduction to the Theory of Computation*. Springer-Verlag.

Smith, C. (1994b). Three decades of team learning. In *Proceedings of AII/ALT'94, Lecture Notes in Artificial Intelligence*, volume 872, pages 211–228. Springer Verlag.

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

Wexler, K. and Culicover, P. W. (1980). *Formal Principles of Language Acquisition*. The MIT Press.

Wiehagen, R. (1976). Limes-erkennung rekursiver funktionen durch spezielle strategien. *Elektronische Informationsverarbeitung und Kybernetik*, 12:93–99.

Appendix

The completely formal definitions for the concepts discussed in the paper appear in this appendix. Some familiarity with the fundamental notions of recursion theory is assumed. For an discussion of all the material assumed below, see one of (Machtey & Young 1978; Smith 1994).

Definition 1 The *functions of finite support* are the set of all recursive functions f such that $f(x) = 0$ for all but finitely many x's.

Definition 2

1. An *inductive inference machine* is an algorithmic device the inputs examples interpreted as ordered pairs from the graph of some function and outputs programs.
2. Suppose inductive inference machine M is given the entire graph of some function f and outputs a sequence of programs: $p_0, p_1, p_2, \cdots$. The M *converges on* f iff *either* the sequence of outputs is finite *or* there exists an i such that for all $j \geq i$, $p_j = p_i$.
3. An inductive inference machine M *learns a function* f iff M converges on f to a program that computes f.
4. An inductive inference machine M *learns a set of functions* S iff M learns each f in S.
5. A set of functions S is *learnable* iff there is an inductive inference machine M that learns S.

Notice that no mention is made of the order in which an inductive inference machine receives its input is made mention of in the above definition. This is because any inductive inference machine can be made to operate in a manor which is independent of the order the inputs arrive in without any loss of generality (Blum & Blum 1975).

Now we give the definitions of learning problem reductions. Let the natural numbers serve as names for programs and φ_i denote the function computed by program i. Any natural way of associating names for programs with actual programs results in what is called a *Gödel numbering* (Rogers 1958) or an *acceptable programming system*.

Definition 3 An *admissible sequence* for a recursive function f is a sequence of programs $p_0, p_1, \cdots$ such that for some n, $\varphi_{p_n} = f$, and for all $n' \geq n$, $p_{n'} = p_n$.

Definition 4 An *operator* is a mapping from functions to functions. Θ is a *recursive operator* iff there is a recursive function f such that, for any program i, $\Theta(\varphi_i) = \varphi_{f(i)}$.

Definition 5 Suppose U and V are learnable sets of recursive functions. Then, U is *reducible* to V iff there are recursive operators Θ and Ξ such that

1. Θ is injective on U, and
2. $\Theta(U) \subseteq V$, and
3. for any $f \in U$ and any admissible sequence τ for $\Theta(f)$, $\Xi(\tau)$ is an admissible sequence for f.