CHAPTER VI

Complexity of Random Problems

Y. S. Abu-Mostafa

Electrical Engineering and Computer Science Departments
California Institute of Technology
Pasadena, CA 91125

The notion of random problems has been introduced to capture the difficulty of modeling natural pattern recognition problems. These problems do not have a concise effective definition. We discuss the heuristic and formal aspects of random problems, and quantify their difficulty through a number of complexity measures. The relations between these measures provide lower and upper bounds on the cost of the system that solves a given random problem. Finally, we address the role of learning by example in solving random problems from a theoretical and practical points of view.

I. Introduction

It is evidently very difficult to model natural environments such as visual scenes. In certain applications, such as commercial data compression, simplified models seem to be adequate for the job. However, in applications such as pattern recognition, simplified models invariably lead to unsatisfactory results when the method based on the model is applied to realistic inputs.

The difficulty in modeling natural environments is reflected in the failure to find satisfactory answers to some very basic questions about these enivronments. One such question is: what makes two pictures 'similar'? Answering this question amounts to defining a distance metric on the class of pictures, such that similar pictures will be a small distance apart. Failure to define such a metric is the reason why a theory as elegant as rate-distortion theory has not had a notable practical impact in speech communication for example.

Is it possible to solve the pattern recognition problem in a natural environment without having a good model for the environment in question? Let us consider for example the problem of recognizing the presence of a tree in a picture (a 512×512 array of bits). It may seem possible to perform this task without knowing what a tree is, simply by writing an algorithm that takes the 512×512-bit input array and performs some preprocessing, edge detection, and parsing to decide whether there is tree or not. However, a closer look reveals that the algorithm must have the definition or the model of a tree embedded in it, for one can in principle use the algorithm to classify every possible 512×512 array into tree/no tree thus exactly defining what it means to be a tree in a picture.

The necessity for having, explicitly or implicitly, a model for the environment is particular to pattern recognition. In a different application such as computer graphics, one may be able to generate what looks like a very natural tree without having, explicitly or implicitly, a model for what a tree is. The reason is that the graphics algorithm will be considered successful if it can produce a number of pictures containing real-looking trees, not *all* pictures containing real-looking trees. One can therefore focus on a very small subset of trees which have an underlying regularity that makes them easier to generate but does not take away from their natural appearance. The problem with pattern recognition is that the algorithm should be ready for an *arbitrary* input.

Is the difficulty in finding good models for natural environments inherent? Unless and until a model is found, this remains a subjective question since we do not know, in formal terms, what these environments are (if we knew, that would be the model!). This difficulty extends to problems of pattern recognition in natural environments. We will study a formal class of problems called random problems, which are inherently difficult to model. The definition of random problems formalizes the plausible properties of natural pattern recognition problems. The relevance of this chapter to natural pattern recognition problems is thus a thesis, not a provable fact. The notion of random problems can also be studied as a mathematical notion, without regard to its possible connections with pattern recognition.

The organization of this chapter is as follows. We motivate and formalize what it means for a problem to be random in section II. This includes a dichotomy between random problems and structured problems. In section III, we introduce different measures that quantify the complexity of a random problem, derive basic relations between these complexity measures, and discuss other measures that can be defined. The real complexity, however, lies in the automated learning of a random problem, which is the subject of section IV. This section discusses the basic issues involved in a learning process.

II. Random Problems

We will consider the decision problem as defined by a function $f:\{0,1\}^N \rightarrow \{0,1\}$ where N is arbitrary but fixed. For example, in the above discussion of recognizing the presence of a tree in a picture, the problem would be formalized as a function $f:\{0,1\}^{512\times512} \rightarrow \{0,1\}$, where $f(\mathbf{x})=1$ iff $\mathbf{x}$ (interpreted as a 512×512 visual scene) represents 'a picture that contains a tree'. Whereas natural patterns are usually represented by analog values on a continuous domain, we shall deal with patterns represented by a finite number of bits. This is adequate for conveying all the main ideas, and generalization to analog representations can be made using rate-distortion methods.

As a computational problem, the tree-in-a-picture problem as well as many other natural pattern recognition problems is best posed as a finite problem (fixed N). This contrasts the canonical way of posing computational problems that allows the input to be any one of an infinite number of possible instances (variable N). For example, the Hamilton-circuit problem (does a given graph have a 'path' of edges that contains every vertex once and only once) can be posed for any graph of any size. The canonical way allowing variable N is necessary to invoke the tools of algorithms and time complexity which require uniformity with respect to N. Posing the tree-in-a-picture problem as a uniform problem would require the extension of the finite problem (say with $N=(512)^2$) to arbitrarily large values of N. However, there is no natural extension for the tree-in-a-picture problem that parallels conventional problems such as the Hamilton-circuit problem. For example, having finer sampling of the scene (say $N=(1024)^2$) does not really provide a 'bigger' instance of the problem, just a better (and perhaps more informative) representation of the same instance.

Fortunately, our characterization of problems as random versus structured will not require uniformity with respect to N. Therefore, we will not be forced into artificial extensions of finite problems. Let us develop the notion of random versus structured for problems of the form $f:\{0,1\}^N \rightarrow \{0,1\}$. Consider the following two problems where $N=30$ and the input string $\mathbf{x}\in\{0,1\}^{30}$ is interpreted as the binary representation of a number in the range from 0 to $2^{30}-1 \approx 10^9$. The first problem addresses the question 'is this number a prime?', i.e., $f(\mathbf{x})=1$ iff $\mathbf{x}$ represents a prime number. To solve this familiar problem, one can apply any of the known tests for primality to the input $\mathbf{x}$. Alternatively, one can take advantage of the finite nature of the problem and consult an exhaustive list of all primes in the range of values $\mathbf{x}$ can assume.

The second problem is less familiar and offers fewer choices for solution. Given $\mathbf{x}$, it addresses the question 'is this the social security number of a convict?', i.e., $f(\mathbf{x})=1$ iff $\mathbf{x}$ represents the social security number of a convict.

There is clearly a basic difference between the 'prime' problem and the 'convict' problem. Let us first rid the 'convict' problem from secondary attributes that may confuse the issue. The notion of a convict may be a bit ambiguous, but one can clearly make it well defined by spelling out some details such as 'convicted by a US court of law'. Even with the notion of a convict unambiguously defined, the basic difference between the two computational problems persists. This difference is what distinguishes structured problems from random problems.

In order to solve the 'convict' problem, we need a list of the social security numbers of all convicts. Although we can also solve the 'prime' problem by consulting the list of primes, we have the option there to use a primality test. The primality test can be viewed as a short effective definition of the notion 'prime'. In the case of the notion 'convict', it is highly unlikely that there is a simple relation between the social security number and the legal status of a person. Therefore, the list of convicts which constitutes an effective definition of the notion 'convict' cannot be compressed into a significantly shorter definition. Problems which do not have a concise effective definition are called *random problems* [1,3]. The characterization 'random' is based on the Kolmogorov complexity approach to randomness [9], which is the lack of a concise effective definition. It is unrelated to the probabilistic connotation of the word 'random'. This will be made precise shortly.

Having no concise effective definition does not mean that the list of social security numbers of convicts is the shortest effective definition of the notion 'convict'. Indeed, there may well be simple relations that allow the list to be somewhat compressed. This property is typical in many natural pattern recognition problems that can be modeled as random problems. Invariably, there will be some regularities that can be taken advantage of. These regularities often obscure the random nature of these problems. An effective definition of a problem has to cover the 'irregular component' as well, and may therefore have to be long even if there are many regularities in the problem.

To formalize the notion of a random problem, we use the Universal Turing Machine [14,17] as our computational model. Let U be a fixed Universal Turing Machine. U can be thought of as a standard general-purpose computer that runs programs on data in the usual way. Let $f:\{0,1\}^N \rightarrow \{0,1\}$ be the problem in question. Computing f on the machine U requires a program $\mathbf{p} \in \{0,1\}^*$ (a finite binary string which can be thought of as the compiled program in machine language). The length of the program $\mathbf{p}$, denoted by $|\mathbf{p}|$ in bits, is the main criterion here. If f can be computed with a short program, f will be considered a structured problem. If the shortest program that can compute f is longer than a certain threshold, f will be considered a random problem.

What does it mean for a program $\mathbf{p}$ to compute f on the machine U? There are two plausible definitions which turn out to be essentially equivalent, as far as the length of the program is concerned. We may require that U, starting with $\mathbf{p}$ and any input $\mathbf{x} \in \{0,1\}^N$, terminates leaving the value of $f(\mathbf{x})$ (this operation is denoted by $U(\mathbf{p},\mathbf{x}) = f(\mathbf{x})$). This

definition corresponds to the situation where we have successfully written a pattern recognition program that is ready to take any input pattern and classify it correctly. The other definition requires $\mathbf{p}$ to define f ; rather than to compute it for a given input. Let $\tau(f)$ be a listing of the truth table of f , i.e., $\tau(f) = \tau_0\tau_1 \cdots \tau_{2^N-1}$ where τ_k is the value of $f(\mathbf{x})$ when $\mathbf{x}$ is the N-bit binary representation of the number k (thus $\tau(f)$ is the 'last column' in the truth table of f that holds the 2^N binary values $f(\mathbf{x})$ assumes as $\mathbf{x}$ varies over $\{0,1\}^N$). Defining f amounts to specifying $\tau(f)$, and we thus require that U, starting with $\mathbf{p}$, terminates leaving $\tau(f)$ as output. In symbols, $U(\mathbf{p}) = \tau(f)$. In this definition, there is no particular input we want to classify. However, both definitions imply that $\mathbf{p}$ has complete information about f . In the first definition, this is shown by the ability of $\mathbf{p}$ to compute $f(\mathbf{x})$ for an arbitrary $\mathbf{x}$, while in the second definition this is shown by the fact that $\mathbf{p}$ provides the entire truth table of f .

The reason why these two definitions are essentially equivalent in terms of the length $|\mathbf{p}|$ is that each program can be simply modified into the other without significant increase in length. If we have the first program that takes $\mathbf{x}$ and computes $f(\mathbf{x})$, we can modify it by adding a routine that generates all $\mathbf{x} \in \{0,1\}^N$, lexicographically, runs the program on each $\mathbf{x}$, and saves the computed value of $f(\mathbf{x})$. The result is that the modified program ends up producing the entire truth table $\tau(f)$ which is what the second program is supposed to do. On the other hand, if we have the second program that produces $\tau(f)$ without reference to any particular input $\mathbf{x}$, we can modify it by adding a routine that takes a particular $\mathbf{x}$, runs $\mathbf{p}$ to generate $\tau(f)$, and then searches through $\tau(f)$ for τ_k that corresponds to $f(\mathbf{x})$. When it finds this τ_k, it erases everything else and terminates. The result is that the modified program ends up producing the value of $f(\mathbf{x})$ for a particular $\mathbf{x}$ which is what the first program is supposed to do. It is easy to verify that these modifications lengthen each program by only $o(N)$ bits. We choose the second program $\mathbf{p}$ that generates $\tau(f)$ and base our formal definitions on it.

There are many different programs $\mathbf{p}$ that can generate the truth table $\tau(f)$ for a given problem $f : \{0,1\}^N \rightarrow \{0,1\}$, and some of these programs will be shorter than others. The shortest possible length of a program $\mathbf{p}$ that generates $\tau(f)$ is called the Kolmogorov complexity [9] of the string $\tau(f)$. In general, the Kolmogorov complexity of a string $\mathbf{s} \in \{0,1\}^*$ is defined by

$$K(\mathbf{s}) = \min \{ \, |\mathbf{p}| \;\; | \;\; U(\mathbf{p}) = \mathbf{s} \, \}$$

(again, $U(\mathbf{p}) = \mathbf{s}$ denotes that U, starting with $\mathbf{p}$, terminates leaving $\mathbf{s}$). The Kolmogorov complexity of $\mathbf{s}$ measures the degree of randomness or lack of structure in $\mathbf{s}$. The more random strings $\mathbf{s}$ will have larger values of $K(\mathbf{s})$. However, $K(\mathbf{s})$ will never have to be larger than $\approx |\mathbf{s}|$, for one can always generate $\mathbf{s}$ by a program $\mathbf{p}$ whose function is 'print $\mathbf{s}$', and this program is only marginally longer than $\mathbf{s}$. In fact, the most random strings $\mathbf{s}$ do have $K(\mathbf{s}) \approx |\mathbf{s}|$, which can be shown by counting the number of short programs and realizing that there are not enough of them to generate all strings of length $|\mathbf{s}|$. When applied to $\mathbf{s} = \tau(f)$, the Kolmogorov complexity leads to the following definition [2]:

Definition. The *randomness* R of a problem $f:\{0,1\}^N \rightarrow \{0,1\}$ is defined by

$$R(f) = \log_2 K(\tau(f)) \quad \text{bits}$$

The logarithm is taken to normalize $R(f)$ to the range from ≈ 0 (where $\mathbf{p}$ is very short, hence $\tau(f)$ is highly structured) to $\approx N$ (where $\mathbf{p}$ is as long as $\tau(f)$ itself, hence $\tau(f)$ has no pattern whatsoever).

The value of $R(f)$ is what determines whether f is a structured problem or a random problem. The idea of the definition of random problems is that they have larger values of $R(f)$ than do structured problems. There is a number of technical variations on how one can approach this definition. We will fix a threshold R_0 (the particular choice of which is not crucial to most of the theory) and introduce the following definition [3].

Definition. A problem $f:\{0,1\}^N \rightarrow \{0,1\}$ is said to be *random* if $R(f) \geq R_0$.

For example, if our threshold $R_0 = 20$, we will call a problem random if it has no effective definition (i.e., if there is no program that generates $\tau(f)$) which is shorter than 2^{20} (about a million) bits. One can hardly expect a human programmer to be able to write a program for such a problem. It is not the program length, per se, that makes us expect that human programmers will not be able to write the program. After all, programming teams have written extremely elaborate programs in a systematic way. It is rather the fact that the *shortest* program for the problem is very long that implies that there is no pattern in the problem that can be understood and translated systematically into computer code by a human programmer.

With $R_0 = 20$, the vast majority of classification problems that can be defined on a 512×512 image are (very) random, but most of them have no practical significance. Indeed, most of the conventional computational problems are structured, not random. One example is the Hamilton-circuit problem. Let us take $N = (512)^2$ and interpret the input $\mathbf{x}$ as a 512×512 adjacency matrix of a graph on 512 vertices, where 1 in the (i,j) position means that there is an edge between vertex i and vertex j, and 0 means that there is no edge. To compute $f:\{0,1\}^{512\times 512} \rightarrow \{0,1\}$ is to determine whether there is a path of edges in the graph that contains every vertex once and only once. One algorithm to do that is to exhaust all possible (cyclic) permutations of the vertices and check, for each permutation, if there is an edge between every consecutive pair of vertices. Coding this algorithm as a program does not take very many bits (although running the program will take a very long time). Whenever there is a short algorithm for a problem, no matter how inefficient, the problem is structured. In spite of being a structured problem, the Hamilton-circuit problem is believed to be an intractable problem, requiring time exponential in N to be solved. It is the length of the program, not the running time, that makes a problem random or structured.

The use of the Universal Turing Machine U as the computational model on which the definition of random problems is based has certain ramifications. On the one hand, it makes the definition quite general since any other 'reasonable' model of computation can be simulated by U (the Church-Turing thesis) without increasing the length of the program involved by

more than a constant. On the other hand, U may be too powerful a model for practical purposes. For instance, U is allowed to run the program $\mathbf{p}$ to generate $\tau(f)$ for any length of time as long as it eventually terminates. It is a standard result in the theory of computation that some programs will run for an exceedingly long time before they terminate. It seems of no practical use to have a short program that would generate $\tau(f)$ if it were allowed to run for millions of years! One remedy for this problem is to introduce a less general but more practical definition of randomness that includes a restriction on the running time. One such definition requires a program $\mathbf{p}$ that computes $f(\mathbf{x})$ for any $\mathbf{x} \in \{0,1\}^N$ within T steps, where T is some practical time function of N. Let $U_T(\mathbf{p}) = \mathbf{s}$ denote that U, starting with $\mathbf{p}$, terminates within T steps leaving $\mathbf{s}$ as output. The time-bounded randomness can be defined by

$$R_T(f) = \log_2 \min \{ |\mathbf{p}| \mid U_T(\mathbf{p},\mathbf{x}) = f(\mathbf{x}) \}$$

and a definition of time-bounded-random problems can be based on that. Another dilemma is resolved when such restricted definitions of randomness are adopted. This logical dilemma arises when we want to *prove* that a given problem is random. Chaitin's version of Gödel's incompletenesss theorem [6] implies that it is impossible to prove that a given problem is random (using the usual axiomatic systems, and assuming R_0 is not very small). However, it becomes possible (though perhaps difficult) to prove that a given problem is time-bounded random when an effective time bound T is used. It is readily possible, however, to prove that *most* of the problems in some simple classes are random problems.

The difficulty in proving randomness for specific problems creates an awkward situation for the application of the theory. When we consider a specific problem such as the tree-in-a-picture problem or the Hamilton-circuit problem, we may be able to tell when it is a structured problem by finding a simple algorithm for it, but we will be unable to tell with certainty when it is a random problem. The uncertainty is apparent when we are dealing with a natural pattern recognition problem for which no simple algorithm has been found. Should we assert that such a problem is a random problem? There are compeling, yet inconclusive, informal arguments for and against the assertion. However, the mere fact that the problem may be structured is of no use if the 'short program' cannot be found. In practice, problems which appear to be random, but may have an unknown wry structure, will be treated as random problems. Fortunately, the approach for solving random problems which is discussed in section IV, will still work if the problem turns out to be structured. In section III, we will prove that certain problems are random, with high probability.

Another practical modification in the definition of randomness aims at tolerating small errors. If it is considered too rigid to require $\mathbf{p}$ to produce the truth table of f perfectly (or equivalently classify each input $\mathbf{x}$ correctly), one can relax this requirement. Assumung a probability distribution on the input space $\{0,1\}^N$ (measuring relative frequency or relative importance of different $\mathbf{x}$'s), and allowing error δ of the time, the following definition emerges.

$$R_\delta(f) = \min \{ R(g) \mid \Pr(f(\mathbf{x}) \neq g(\mathbf{x})) \leq \delta \}$$

where, instead of insisting on computing $f(\mathbf{x})$ or $\tau(f)$, we settle for a function g which does not differ much from f, but is simpler to describe. If $R_\delta(f)$ is used instead of $R(f)$ to define random problems, some previously random problems will not be considered random any more because they have a structured approximation. Using $R_\delta(f)$ may be more plausible practically, not only because in practice some error can usually be tolerated, but also because natural pattern recognition problems are typically fuzzy and it would be an overspecification to require perfect classification of every input. For instance, some images equally qualify as containing a tree or not containing one, depending on who is looking. More importantly, some 'images' do not look like pictures of anything in the first place and we may not care how our tree-in-a-picture algorithm would classify them. A score of other variations of $R(f)$, and a corresponding number of notions of a random problem, can be produced by merging the error tolerance, the bound on computation time, and other practical considerations.

III. Complexity Measures

In this section, we address the difficulty of random problems through a number of complexity measures. These measures are based on different models of computation, ranging from simple lists to Turing machines. The values of these measures reflect the difficulty of the problem from the point of view of the respective models. Our first complexity measure is the randomness $R(f)$ itself, which is the basis for categorizing a problem $f:\{0,1\}^N \rightarrow \{0,1\}$ as random ($R(f) \geq R_o$) or structured. Although this categorization is binary (random versus structured), the actual value of $R(f)$ describes in more detail how random or how structured a problem is, and how big the system that implements f has to be. We start by investigating the distribution of values $R(f)$ assumes.

Let us take N sufficiently larger than R_o, say $N > R_o + 10$. Let f be any of the 2^{2^N} problems that can be defined with N Boolean inputs. The range of values of $R(f)$ is determined by $R(f) = \log_2 K(\tau(f))$. The Kolmogorov complexity $K(\mathbf{s})$ of any string $\mathbf{s}$ ranges from the minimum length of any syntactically correct program, to the length of the program 'print $\mathbf{s}$'. Therefore, depending on the technical details of the Universal Turing Machine model U, there are constants C_1 and C_2 such that

$$C_1 \leq K(\mathbf{s}) \leq |\mathbf{s}| + C_2$$

Since $|\tau(f)| = 2^N$, $R(f)$ ranges from ≈ 0 to $\approx N$ bits. In fact, the vast majority of the 2^{2^N} problems $f:\{0,1\}^N \rightarrow \{0,1\}$ have $R(f) \approx N$. To see this, we estimate the number of problems having $R(f) < N-1$. Each f in this class must have a distinct program $\mathbf{p}$ of length $|\mathbf{p}| < 2^{N-1}$ that generates $\tau(f)$. However there are at most $2^0+2^1+2^2+\cdots+2^{2^{N-1}-1} < 2^{2^{N-1}}$ programs in this length range, which is much less than 2^{2^N}. Hence, almost all problems are crowded around $R(f) \approx N$. This also implies that almost all problems are random problems.

We can use this fact to construct a random problem in a probabilistic manner. We construct $\tau(f)=\tau_0\tau_1\cdots\tau_{2^N-1}$ by flipping a fair coin independently to determine each τ_k. Since this experiment generates any of the 2^{2^N} problems with equal probability, and since almost all of these problems are random problems, the experiment will, with overwhelming probability, generate a random problem. This experiment highlights the relation between the notion of randomness in the Kolmogorov sense as used in defining random problems and the notion of randomness as used in probability theory. If we generate a problem in a random (probability) way, we will probably get a random problem (Kolmogorov)! It is important not to confuse the two notions. However, it is useful to use the fact that probability is involved in constructing a problem to draw conclusions about its likely status as a random problem.

The second complexity measure is based on the observation that, in many natural pattern recognition problems, the truth table $\tau(f)$ has many more 0's than 1's, or vice versa. For example, in the tree-in-a-picture problem, the set of pictures containing trees is a tiny subset of the set of all 512×512 binary arrays. $\tau(f)$ of this problem has far fewer 1's than 0's. The reason for this common situation in natural pattern recognition problems is that the input is usually represented in a highly redundant fashion. This means that many allowable input patterns never occur in practice (correspond to *don't care* [8]). When the representation is redundant, the dimensionality N of the problem is more than the 'essential' dimensionality. Our second complexity measure quantifies this essential dimensionality, or entropy [2].

Definition. Let $h(f)=\min\left\{\,|f^{-1}(1)|\,,\,|f^{-1}(0)|\,\right\}$. The (deterministic) *entropy* of f is defined by

$$H(f)=\log_2(1+h(f)) \quad \text{bits}$$

where $f^{-1}(1)=\{\mathbf{x}\mid f(\mathbf{x})=1\}$ and $f^{-1}(0)=\{\mathbf{x}\mid f(\mathbf{x})=0\}$, which means that $h(f)$ is the number of 1's or the number of 0's in $\tau(f)$, whichever is smaller. Clearly $0\leq h(f)\leq 2^{N-1}$, hence the entropy $H(f)$ ranges from 0 to $\approx N$ bits, essentially the same range as $R(f)$.

In some sense, $H(f)$ is a weak complexity measure. Knowing $H(f)$ alone does not provide much information about the complexity of f. For instance, a function with few 1's (hence low entropy) may be fairly complex if the 1's are scattered at random in the truth table, whereas a function as simple as $f(\mathbf{x})=x_1$ (the first bit of $\mathbf{x}$) has the maximum possible entropy. However, in the Appendix, we show $R(f)$ can be at most $\approx H(f)$, hence the entropy serves as an upper bound for the randomness.

We have used the symbol '$\approx$' without formal definition. What we mean by $\approx A$ is $A\pm o(N)$, where $o(N)$ is a function of N which is asymptotically negligible with respect to N. The error terms represented by $o(N)$ are a consequence of formalization technicalities and simulation overhead, and they will vary depending on specific details of the models.

The strength or weakness of a complexity measure is a consequence of the strength or weakness of the model on which the measure is based. The criterion in each measure is the size of the specification of $\tau(f)$ using the model. For example, $R(f)$ is based on the Universal

Turing Machine and the criterion is the length of the program **p** that runs on U to generate $\tau(f)$. Similarly, we can think of $H(f)$ as being based on a model that genrates $\tau(f)$ when we specify where the 1's are (or the 0's, whichever are fewer). The size of the specification is taken as the number of points we need to specify. Because the model is simplistic, the size of the specification is sometimes more than necessary because it does not take advantage of any relation between where the 1's (or 0's) are in $\tau(f)$. The Universal Turing Machine, on the other hand, is a powerful model that can take advantage of very subtle relations and this allows **p** to be shorter and shorter.

What makes a model more powerful than another? If model A can simulate model B, but not vice versa, A is more powerful than B. Simulating B means taking a description of B together with the input to B and simulating what B would do on the input. For example, the Universal Turing Machine can be programmed in a straightforward manner to generate $\tau(f)$ from the specification of where the 1's (or the 0's) are, while the simplistic model that does only that cannot simulate an arbitrary program running on the Universal Turing Machine. This is why the randomness $R(f)$ is less than or equal to $\approx H(f)$. In fact, these two measures are extreme in that their models are very weak or very strong. Other measures based on intermediate models will lie between $R(f)$ and $H(f)$.

The third complexity measure is based on a model which is a tradeoff between model power and practical realization. The model is combinational circuits [14], which are loop-free interconnections of logic gates. Combinational circuits cover a wide spectrum of practical electronic circuits as well as certain classes of neural networks. The measure is called $C(f)$, the *circuit complexity*. It measures, on a logarithmic scale, the minimum size of a circuit implementation of the problem f. The range of $C(f)$ is also $0 \leq C(f) \leq N$ bits. The particular choice of the logic gates in the circuit that defines $C(f)$ affects the value of the measure only marginally. In the Appendix, we give a concrete definition for $C(f)$, and show that $C(f)$ lies between $R(f)$ and $H(f)$, i.e., $R(f)$ is at most $\approx C(f)$ and $C(f)$ is at most $\approx H(f)$. If we neglect terms of the order $o(N)$, we can write

$$R(f) \leq C(f) \leq H(f)$$

This relation will also hold for other variations of $C(f)$ which are not based on a purely combinational model. In practice, when we try to build systems to solve pattern recognition problems, we do not have all the details of the system at the outset. The above relation provides bounds on one important detail, how big the system has to be.

While $R(f)$ is as small as a measure can be that is based on a 'realizable' model, $H(f)$ is as large as a measure can be that is based on a 'useful' model. Measures based on other practical models with varying degree of sophistication will lie somewhere between $R(f)$ and $H(f)$. $C(f)$ is such a measure, and it is practically more relevant to the cost of implementing f than $R(f)$ is, although $R(f)$ is more profound.

The differences between these complexity measures are also meaningful quantities. For example, $H(f)$–$R(f)$ is called the *false entropy* of f. Since $H(f)$ does not take advantage of any regularity in $\tau(f)$ while $R(f)$ takes advantage of all effective regularities, false entropy quantifies the regular part or the hidden structure of the problem. For example, the tree-in-a-picture problem has a significant false entropy due to such regularities as shift and scale invariance in visual scenes and a number of other, more subtle, properties. The identification of these regularities dramatically simplifies the problem, since in effect it reduces the entropy by the corresponding number of bits. Since the actual size is exponential in the complexity measure, one bit difference cuts the system size by half. We will come back to false entropy when we discuss learning in section IV.

Finally, we define a version of $C(f)$ that accommodates some error tolerance, in the same way we defined $R_\delta(f)$ in section II. We assume a probability distribution on the set of inputs $\{0,1\}^N$ and define

$$C_\delta(f) = \min \{ C(g) \mid \Pr(f(\mathbf{x}) \neq g(\mathbf{x})) \leq \delta \}$$

where δ is the allowed probability of error. Clearly, $C_\delta(f) \leq C(f)$. In words, $C_\delta(f)$ measures the smallest size of a circuit that can approximately solve the problem f.

IV. Learning

When a problem $f : \{0,1\}^N \rightarrow \{0,1\}$ is declared random, the implication is that no small system, be it an algorithm or a circuit, will be able to solve it. This constraint, by itself, does not pose a serious difficulty for solving the problem in practice (for example, if $C(f) = 30$, this implies a circuit with about a billion gates). Indeed, recent and projected technological advancements make it possible to build huge and intricate systems in a routine fashion. The difficulty lies in *finding* the system that solves a given a problem. The shortest program to solve a random problem will necessarily have no logical pattern to it; if it did one could take advantage of this pattern and write a shorter program. The practical impossibility of finding a near-optimal system purely by 'thinking about it' is the premise for introducing the notion of a random problem in the first place. If the premise proves false, the distinction between random problems and structured problems will be that of degree, not of nature.

However, if we do not insist on a near-optimal system, there is a canonical solution to any problem, be it random or structured. We store $\tau(f)$ in memeory and when we want to classify an input $\mathbf{x}$, we search through $\tau(f)$ for the corresponding τ_k. This may be a viable solution for the 'convict' problem for example, where $\tau(f)$ is not too long. We may even take advantage of the low-entropy nature of the problem and store only the $\mathbf{x}$'s where $f(\mathbf{x}) = 1$ (the list of convicts), and search through this list when we want to classify an input. However, if we want to solve the tree-in-a-picture problem, this approach is totally out of the question. Neither N (the relevant parameter for the storage of $\tau(f)$) nor $H(f)$ (the relevant parameter for the

storage of the 1's of f) is small enough to make these exhaustive schemes possible.

Example. Consider a simplified tree-in-a-picture problem $f:\{0,1\}^{512\times 512}\rightarrow\{0,1\}$ constructed as follows. We take one particular picture containing a tree (say a picture you took in your back yard and digitized into 512×512 binary pixels), call it $\hat{\mathbf{x}}$. We then pick 100 fixed pixels scattered around (uniformly if you will) and designate these as noise pixels that are allowed to be 1 or 0. Having at most 100 scattered 'spots' out of more than 250,000 pixels will not obscure the appearance of the picture. The problem is thus defined by $f(\mathbf{x})=1$ if, and only if, $\mathbf{x}$ agrees with $\hat{\mathbf{x}}$ in all pixels except (possibly) the designated noise pixels. Solving this problem by the above exhaustive schemes requires prohibitive storage. Indeed, storing the 1's of this extremely simplified pattern recognition problem requires more than 10^{30} bits, which will remain prohibitive even when this book becomes old!

The difference between an exhaustive solution and an optimal solution is quantified precisely by the false entropy. We pay an exponential price for every bit of false entropy we fail to 'remove'. This reflects the essential difficulty in solving pattern recognition problems. Exhaustive schemes cannot be implemented, and non-exhaustive schemes cannot be found. While the former difficulty is unsurmountable, the latter may be overcome if we can find an *automated* method for constructing the system, rather than try to construct the system ourselves. The method of learning is a good candidate. Learning is the mechanical process that takes as input pieces of information about f and produces as output an implementation of f. For example, in real life, children see and hear about trees, then become able to recognize trees. The following questions address some key issues pertaining to a learning process.

1. Is the input information about f sufficient?
2. How costly, and how good, is the output implementation of f ?
3. Is the learning process feasible?

Example. One of the simplest models of a learning process is an algorithm L whose input is the complete information about f, namely $\tau(f)$, and whose desired output is a program $\mathbf{p}$ of the shortest possible length that can generate $\tau(f)$. However, such an algorithm does not exist. This fact is a consequence of the uncomputability of the Kolmogorov complexity [9].

This example implies that the automatic removal of the entire false entropy is impossible. When the desired output is a combinational circuit rather than a program, the uncomputability question goes away since one can in principle find the smallest circuit by exhaustive search. The feasibility of learning in these cases becomes a question of how much time is needed to produce the output. Valiant [18] and others formalize this question and give partial answers in a number of cases. Unfortunately, most of the results about interesting learning tasks are negative, predicting that the learning problem is intractable. However, this research area is still at its infancy, and a lot of work needs to be done before solid conclusions can be drawn about the feasibility of automated learning for practical problems. In what follows, we will discuss some basic considerations.

The basic learning scheme is *learning by example.* In this scheme, the learning process is provided with a number of examples $\mathbf{x} \in \{0,1\}^N$ together with the corresponding values $f(\mathbf{x})$. The process is supposed to use this information to produce a system that implements f for all inputs. Learning by example in this pure form reflects the ideal situation where we do not have to do any thinking, just pump examples into the process and wait for the system to pop out! As one would expect, experimental results of this scheme are not very promising. A closer look reveals inherent limitations. There must be restrictions on the class of problems from which f is chosen if a (non-exhaustive) set of examples will suffice to define f to the learning process. Moreover, how the process extrapolates the set of examples must match the restrictions of the class of problems in question. Given all that, the question remains about how much time is needed for the process to produce the system.

How the process extrapolates a set of examples to the entire input space is a major consideration in learning which is called *generalization.* Any viable learning process must have some capability of generalization. We cannot hope to provide an exhaustive set of examples in a practical situation, not only because of their astronomical number, but also because of the difficulty of producing them (e.g., the set of all pictures containing trees). Generalization forces the assumption of underlying regularity, and if the correct generalization is made, this amounts to removing false entropy. Any number of questions can be asked about tradeoffs between the parameters of a generalization process.

It is generally a waste of resources (sometimes fatally so) to depend entirely on a set of examples to tell the process about f when we can use our partial understanding of the problem to give 'hints' to the process to point it in the right direction. For example, the learning process of the tree-in-a-picture problem will treat scaled versions of the same picture as unrelated pictures if we do not tell it somehow about the scale invariance of visual scenes. Some hints are given to the process in a subtle way. When we provide a given process with a non-exhaustive set of examples, we implicitly hint (perhaps falsely!) that the way the process naturally generalizes is the right way. Some major hints are included in the choice of representation of the input. A visual scene represented by $(512)^2$ pixels with no adjacency information is a hopeless starting point for learning. Each bit of regularity missed by the process doubles the number of apparently independent cases it has to worry about. It is a challenge for theory and practice alike to combine our partial understanding of the problem with learning by example to produce a successful pattern recognition system.

Appendix

In this appendix, we provide a technical definition for the circuit complexity $C(f)$ and prove the relations between $R(f)$, $H(f)$, and $C(f)$. Our building block of circuits is the universal gate. A universal gate with n inputs and 1 output simulates an arbitrary Boolean function of n variables. A combinational circuit is made up of any number of universal gates (usually with different n's) interconnected in a loop-free manner.

Definition. The (denormalized) *cost* of a universal gate of n inputs ($n \geq 0$) is defined to be 2^n 'cells'. The cost of a collection of gates is the sum of the costs of the gates.

This definition is motivated by the actual number of cells in an electronic memory, and by the fact that implementing an n-input universal gate requires an exponential number of standard gates. A collection of Q universal gates with $n_1, \cdots, n_Q$ inputs costs $\sum_{i=1}^{Q} 2^{n_i}$ cells. We use these gates together with the input lines $x_1, \cdots, x_N$ to build a combinational circuit Γ to simulate a given function. Γ simulates f if f is one of the gate outputs $y_1, \cdots, y_Q$.

Definition. The (normalized) *circuit complexity* of a Boolean function f, denoted by $C(f)$, is defined by:

$$C(f) = \log_2 \min \{ \text{ cost of } \Gamma : \Gamma \text{ simulates } f \} \quad \text{bits}$$

A constant function f is the output of a universal gate with zero inputs. Such a gate costs $2^0 = 1$ cell. Hence, $C(f) = 0$ bits for the two constant functions, and $C(f) > 0$ for all other functions. Also, $C(f) \leq N$ for any function f which depends on N variables, since a universal gate with N inputs (2^N cells) can simulate any such function. Notice that $C(f)$ differs by at most a constant from the normalized circuit complexity based on any other complete basis of switching devices such as 2-input NAND gates. To see this, one can simulate universal gates using the complete basis and vice versa. For example, a 2-input NAND gate can be simulated by a 2-input universal gate which costs 4 cells. Hence, the simulation costs 4 times the number of NAND gates, in cells. When the logarithm is taken, the two measures differ by at most a constant. Similarly, an n-input universal gate that costs 2^n cells can be simulated by $A \times 2^n / n \leq A \times 2^n$ NAND gates [16].

We now prove [2] that $C(f) \leq H(f) + o(N)$. Functions of entropy H have $h = 2^H - 1$ 1's or 0's in their truth tables. We are interested in estimating the circuit complexity of the functions of entropy H. Without loss of generality, we shall consider only the functions with h 1's.

Definition. Given a function f of entropy H, the state of the variables in a subset S of $\{x_1, \cdots, x_N\}$ is said to be *positive* if there is an assignment of 0's and 1's to the *rest* of the variables that makes $f(\mathbf{x}) = 1$.

f can have at most $h = 2^H - 1$ positive states for any subset S, since there are only h 1's in the truth table of the function. Therefore, as far as f is concerned, the state of the variables in S can be encoded using $\lceil \log_2 (1 + h) \rceil = \lceil H \rceil$ binary variables (the extra 1 represents 'the state is not positive'). Taking $|S| > \lceil H \rceil$, this encoding constitutes information compression, since we represent a number of variables by a smaller number of variables. Furthermore, in terms of the new variables (the compressed variables from S together with the rest of the variables outside S), the entropy of the function remains the same, and hence we can repeat this compression.

Consider an arbitrary function f of N variables whose entropy is H bits. Since we can compress any number of variables into $\lceil H \rceil$ variables, we repeatedly compress $\lceil H \rceil+1$ variables into $\lceil H \rceil$ variables, each time using $\lceil H \rceil$ universal gates of $\lceil H \rceil+1$ inputs. We thus reduce the N variables to $N-1$, $N-2$, ... , down to any number of variables, say $\lceil H \rceil+1$ variables. We can then implement the function f in terms of these $\lceil H \rceil+1$ variables using one universal gate of $\lceil H \rceil+1$ inputs. The compression from N to $\lceil H \rceil+1$ variables takes $(N-\lceil H \rceil-1)\times \lceil H \rceil$ universal gates of $\lceil H \rceil+1$ inputs each, in addition to the final gate with $\lceil H \rceil+1$ inputs. Therefore, the cost of this circuit is $(1+(N-\lceil H \rceil-1)\times \lceil H \rceil)\, 2^{\lceil H \rceil+1}$ cells. Since $0 \leq \lceil H \rceil \leq N$, this cost is $\leq 2^{H+o(N)}$ cells. This proves that $C(f) \leq H(f)+o(N)$.

In order to prove that $R(f) \leq C(f)+o(N)$ (hence $R(f) \leq H(f)+o(N)$), we need to estimate the distribution of $C(f)$, which is done in the following proposition.

Proposition. Let $\hat{N}_K = |\{ f : \{0,1\}^N \rightarrow \{0,1\} \mid C(f) \leq K \}|$. For $0 \leq K \leq N$, the following holds:

$$\log_2 \log_2 \hat{N}_K \leq K + \log_2 (6+N)$$

Proof. We can take $K \geq 1$ since the statement is clear for $K < 1$ (constant functions only). We will use $\exp(x)$ to denote 2^x when convenient. Let $M = \lceil K \rceil$. We overestimate $\hat{N}_K$ by $\hat{N}_M$. To do this, we shall estimate the number of different ways we can choose a collection of gates given the total cost of 2^M cells, the number of circuits that can be formed using a given collection of gates, and the number of functions that can be simulated on a given circuit. We restrict the gates to have a positive number of inputs, since zero-input gates contribute only the constant functions which can be simulated otherwise. Restricting the cost to be *exactly* 2^M cells is justified by adding 1-input gates (all costs involved are even) without using their outputs.

1. A collection of gates is isomorphic to a multiset of numbers (a multiset is a set where repetition of elements is allowed), where each number corresponds to the number of inputs in a gate. The number of multisets $<n_1, \cdots, n_Q>$ with positive n_i's satisfying $\sum_{i=1}^{Q} 2^{n_i} = 2^M$ (i.e., whose cost is 2^M cells) is at most 2^{2^M}. To see this, let ρ_M be the number of different multisets $X_M = <n_1, \cdots, n_Q>$ where the n_i's are positive integers satisfying $\sum_{i=1}^{Q} 2^{n_i} = 2^M$ for the positive integer M. We claim that any X_{M+1} can be written as the union of two X_M's, except $X_{M+1} = <M+1>$. This can be proved by taking the X_{M+1} and replacing each two 1's in it by a single 2 (conceivably leaving a single 1 at the end). We next replace each two 2's by a single 3 and continue in this fashion until we replace the $M-1$'s by M's. This procedure must yield exactly $<M, M>$ because there is at most a single 1, a single 2 , ... , a single $M-1$ left and these cannot contribute to $\sum_{i=1}^{Q} 2^{n_i}$ more than $2^M - 2$. Therefore, we go back and decompose the two M's getting two X_M's which proves the claim. Hence ρ_{M+1} is at most $1 + \frac{\rho_M(\rho_M+1)}{2}$. For

$\rho_M \geq 2$, which holds for $M \geq 2$, this is at most ρ_M^2. The proof now follows by induction after overestimating ρ_1 and ρ_2 by 2^{2^1} and 2^{2^2} respectively.

2. Given N variables $x_1, \cdots, x_N$ and calling the outputs of the Q gates $y_1, \cdots, y_Q$, we have at most $N+Q$ different variables that can be input to each gate. Therefore, for each collection of gates with $n_1, \cdots, n_Q$ inputs, we have at most $\prod_{i=1}^{Q} (N+Q)^{n_i}$ possible interconnection schemes or circuits. This can be rewritten as $\exp\left(\sum_{i=1}^{Q} n_i \log_2 (N+Q) \right)$. Subject to $\sum_{i=1}^{Q} 2^{n_i} = 2^M$, the maximum value of $\sum_{i=1}^{Q} n_i$ occurs when all the n_i's are 1's and is 2^{M-1}. Since Q is also at most 2^{M-1}, we get the overestimate $\exp\left(2^{M-1} \log_2 (N + 2^{M-1}) \right)$.

3. For each of the above circuits, there are $\prod_{i=1}^{Q} 2^{2^{n_i}}$ ways to program the universal gates and, for each of these, we have at most Q implemented functions out of the Q gates. Hence, the number of functions that can be implemented on the circuit is at most $Q \prod_{i=1}^{Q} \exp 2^{n_i}$. This is at most $\exp\left((M-1) + 2^M \right) \leq \exp\left(2 \times 2^M \right)$.

Therefore, from 1,2, and 3, the number of functions f that can be implemented within cost 2^M cells is at most the product of the three estimates, namely $\exp\left((3 + \tfrac{1}{2} \log_2 (N + 2^{M-1}))2^M\right)$. Since $M < K+1$ and also $M \leq N$, this is at most $\exp\left((6 + \log_2 (N + 2^{N-1}))2^K\right)$. Since $N \leq 2^{N-1}$ (N is an integer), this is at most $\exp\left((6 + \log_2 (2^{N-1} + 2^{N-1}))2^K\right)$, which reduces to $\exp \exp\left(K + \log_2 (6+N) \right)$. |

Thus, the number of functions with $C(f) \leq K$ is at most $2^{2^{K+o(N)}}$. A corollary of the proposition is that almost all functions of N variables have $C(f) \geq N - o(N)$, which is a known result [16].

We can now prove that $R(f) \leq C(f) + o(N)$ by constructing a program to generate $\tau(f)$ based on the smallest circuit that simulates f. We fix a lexicographic ordering of *all* circuits, with the less costly circuits coming first. The program $|\mathbf{p}|$ includes the smallest index of a circuit that simulates f (by the proposition, this index will be at most $\log_2 2^{2^{C(f)+o(N)}} = 2^{C(f)+o(N)}$ bits long), a constant-length routine to 'decode' the circuit from its index, and a constant-length routine to generate $\tau(f)$ from the circuit. The details are straightforward.

References

[1] Y. Abu-Mostafa, "Complexity of random problems," *Abstracts of Papers, IEEE International Symposium on Information Theory*, Brighton, England, IEEE Catalog # 85 CH 2201-2, p. 84, June 1985.

[2] Y. Abu-Mostafa, "The complexity of information extraction," *IEEE Trans. on Information Theory*, vol. IT-32, pp. 513-525, 1986.

[3] Y. Abu-Mostafa, "Random Problems," *Journal of Complexity*, Academic Press, vol. 4, 1988.

[4] A. Aho et al., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[5] A. Borodin, "On relating time and space to size and depth," *SIAM J. on Computing*, vol. 6, pp. 733-744, 1977.

[6] G. Chaitin, "Gödel's theorem and information," *International Journal of Theoretical Physics*, vol. 22, pp. 941-954, 1982.

[7] R. Duda and P. Hart, *Pattern Classification and Scene Analysis*, Wiley-Interscience, 1973.

[8] Z. Kohavi, *Switching and Finite Automata Theory*, McGraw-Hill, 1978.

[9] A. Kolmogorov, "Three approaches for defining the concept of information quantity," *Information Transmission*, vol. 1, pp. 3-11, 1965.

[10] O. Lupanov, "Complexity of formula realization of logical algebra," *Problems of Cybernetics*, vol. 3, pp. 782-811, 1960.

[11] P. Martin-Löf, "The definition of random sequences," *Information and Control*, vol. 9, pp. 602-619, 1966.

[12] C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.

[13] N. Pippenger, "Information theory and the complexity of Boolean functions," *Mathematical Systems Theory*, vol. 10, pp. 129-167, 1977.

[14] J. Savage, *The Complexity of Computing*, Wiley-Interscience, 1976.

[15] C. Shannon, "A mathematical theory of communication," *Bell Sys. Tech. J.*, vol. 27, pp. 379-423, 1948.

[16] C. Shannon, "The synthesis of two-terminal switching circuits," *Bell Sys. Tech. J.*, vol. 28, pp. 59-98, 1949.

[17] A. Turing, "On computable numbers with an application to the Entscheidungsproblem," *Proc. London Math. Society*, vol. 42, pp. 230-265, 1936.

[18] L. Valiant, "A theory of the learnable," *Communications of the ACM*, vol. 27, pp. 1134-1142, 1984.