DISCOVERING PROBLEM SOLUTIONS WITH LOW KOLMOGOROV COMPLEXITY AND HIGH GENERALIZATION CAPABILITY
Technical Report FKI-194-94

Jürgen Schmidhuber
Fakultät für Informatik
Technische Universität München
80290 München, Germany

`schmidhu@informatik.tu-muenchen.de`
`http://papa.informatik.tu-muenchen.de/mitarbeiter/schmidhu.html`

August 1994

Abstract

Many machine learning algorithms aim at finding "simple" rules to explain training data. The expectation is: the "simpler" the rules, the better the generalization on test data ($\rightarrow$ Occam's razor). Most practical implementations, however, use measures for "simplicity" that lack the power, universality and elegance of those based on Kolmogorov complexity and Solomonoff's algorithmic probability. Likewise, most previous approaches (especially those of the "Bayesian" kind) suffer from the problem of choosing appropriate priors. This paper addresses both issues. It first reviews some basic concepts of algorithmic complexity theory relevant to machine learning, and how the Solomonoff-Levin distribution (or universal prior) deals with the prior problem. The universal prior leads to a probabilistic method for finding "algorithmically simple" problem solutions with high generalization capability. The method is based on Levin complexity (a time-bounded generalization of Kolmogorov complexity) and inspired by Levin's optimal universal search algorithm. With a given problem, solution candidates are computed by efficient "self-sizing" programs that influence their own runtime and storage size. The probabilistic search algorithm finds the "good" programs (the ones quickly computing algorithmically probable solutions fitting the training data). Simulations focus on the task of discovering "algorithmically simple" neural networks with low Kolmogorov complexity and high generalization capability. It is demonstrated that the method, at least with certain toy problems where it is computationally feasible, can lead to generalization results unmatchable by previous neural net algorithms. Much remains do be done, however, to make large scale applications and "incremental learning" feasible.

Keywords: *Generalized Kolmogorov complexity, Solomonoff-Levin distribution, generalization, universal search, self-sizing programs, neural networks.*

1 INTRODUCTION

The first number is 2. The second number is 4. The third number is 6. The fourth number is 8. What is the fifth number? The answer is 34. The reason is the following law. The nth number is

$$n^4 - 10n^3 + 35n^2 - 48n + 24.$$

But an IQ test requires you to answer "10" instead of "34". Why not "34"? The reasons are: (1) "simple" solutions are preferred over "complex" ones. This idea is often referred to as "Occam's razor". (2) It is assumed that the "simpler" the rules, the better the generalization on test data. (3) The makers of the IQ test assume that everybody agrees on what "simple" means.

Similarly, many researchers agree[1] that learning algorithms ought to extract "simple" rules to explain training data. But what exactly does "simple" mean? The only theory providing a convincing objective criterion for "simplicity" is the theory of Kolmogorov complexity (or algorithmic complexity). Contrary to a popular myth, the incomputability of Kolmogorov complexity (due to the halting problem) does not prevent machine learning applications, because there are tractable yet very general extensions of Kolmogorov complexity. Few machine learning researchers, however, make use of the powerful tools provided by the theory.

Purpose of paper. This work and the experiments to be presented herein are intended (1) to demonstrate that basic concepts from the theory of Kolmogorov complexity are indeed of interest for machine learning purposes, (2) to encourage machine learning researchers to study this theory, and (3) to point to some limitations of the current state of the art and to important open problems.

Outline. Section 2 briefly reviews the following basic concepts of algorithmic complexity theory relevant to machine learning: (1) Kolmogorov complexity, (2) The universal prior (or Solomonoff-Levin distribution), under which the probability of a computable object (like the solution to a problem) is essentially equal to the probability of guessing its shortest program on a universal computer, (3) Solomonoff's theory of inductive inference, and how it justifies Occam's razor, (4) The principle of minimal description length (MDL) in its general form, (5) Levin complexity (a generalization of Kolmogorov complexity) and Levin's universal optimal search algorithm. For a given computable solution to a problem, consider the negative log of the probability of guessing a program that computes it, plus the log of its runtime. The Levin complexity of the solution is the minimal possible value of this. Levin's universal search algorithm essentially generates and tests solution candidates (from a set of possible computable candidates) in order of their Levin complexity, until a solution is found. For a broad class of problems, universal search can be shown to be optimal with respect to total expected search time, leaving aside a constant factor which does not depend on the problem. To my knowledge, section 3 presents the first general implementation of (a probabilistic variant of) universal search on a conventional digital machine. It is based on efficient "self-sizing" programs which influence their own runtime and storage size. Simulations in section 4 focus on the task of finding "simple" neural nets with high generalization capability. Experiments with toy problems demonstrate that the method, at least in certain cases where it is computationally feasible, can lead to generalization results that seem to be impossible to obtain by more traditional neural net algorithms (also briefly reviewed in section 4). To end this paper with a promising outlook, section 5 presents preliminary experiments with extensions designed for incremental learning. There, the "best" solution candidate found so far serves as a basis for additional improvements. Although their theoretical foundations are not yet well-developed, incremental extensions appear to be promising for improving neural net algorithms, evolutionary and genetic algorithms, and other learning paradigms. Section 6 concludes with general remarks on problem solving and Occam's razor.

2 BASIC CONCEPTS RELEVANT TO LEARNING

This section briefly reviews a few basic concepts from the theories of algorithmic probability and Kolmogorov complexity (a.k.a. "algorithmic complexity"). Selected references and a very brief and incom-

[1] The final section provides some remarks for those who don't agree.

plete history of the subject can be found at the end of the section.

Algorithmic complexity theory provides various different but closely related measures for the complexity (or simplicity) of objects. We will focus on the one that appears to be the most useful for machine learning. Informally speaking, the complexity of a computable object is the length of the shortest program that computes it and halts, where the set of possible programs forms a prefix code.

To make this more precise, consider a Turing machine (TM) with 3 tapes: the program tape, the work tape, and the output tape. All three are finite but may grow indefinitely. For simplicity, but without loss of generality, let us focus on binary tape alphabets $\{0,1\}$. Initially, the work tape and the output tape consist of a single square filled with a zero. The program tape consists of finitely many squares, each filled with a zero or a one. Each tape has a scanning head. Initially, the scanning head of each tape points to its first square. The program tape is "read only", the output tape is "write only", their scanning heads may be shifted only to the right (one square at a time). The work tape is "read/write", its scanning head may move in both directions. Whenever the scanning heads of work tape or output tape shift beyond the current tape boundary, an additional square is appended and filled with a zero. The case of the program tape's scanning head shifting beyond the program tape boundary will be considered later. For the moment we assume that this does never happen. The TM has a finite number of internal states (one of them being the initial state). Its behavior is specified by a function F (implemented as a look-up table). F maps the current state and the contents of the square above the scanning head of the work tape to a new state and an action. There are 8 actions: shift worktape left/right, write 1/0 on worktape, write 1/0 on output tape and shift its scanning head right, copy contents above scanning head of program tape onto square above scanning head of work tape and shift program tape's scanning head right, and halt.

Self-delimiting programs. Let $\mid s \mid$ denote the number of bits in the bitstring s. Consider a nonempty bitstring p written onto the program tape such that the scanning head points to the first bit of p. p is a *program* for some TM T, iff T reads all $\mid p \mid$ bits and halts. In other words, during the (eventually terminating) computation process the head of T's program tape incrementally moves from its start position to the end of p, but not any further. One may say that p carries in itself the information about when to stop, and about its own length. Obviously, no program can be the prefix of another one.

Compiler theorem. Each TM C, mapping bitstrings (written onto the program tape) to outputs (written onto the output tape) computes a partial function $f_C : \{0,1\}^* \rightarrow \{0,1\}^*$ (f_C is undefined where C does not halt). It is well known that there is a universal TM U with the following property: for every TM C there exists a constant prefix μ_C such that $f_C(p) = f_U(\mu_C p)$ for all bitstrings p. μ_C is the compiler that compiles programs for C into equivalent programs for U.

In what follows, let p denote a (self-delimiting) program.

Kolmogorov complexity. Given U, the Kolmogorov complexity (a.k.a. "algorithmic complexity," "algorithmic information," or occasionally "Kolmogorov-Chaitin complexity") of an arbitrary bitstring s is denoted as $K_U(s)$ and is defined as the length of a shortest program producing s on U:

$$K_U(s) = \min_p \{\mid p \mid : \; f_U(p) = s\}.$$

$K_U(s)$ is noncomputable, otherwise the halting problem could be solved. However, by comparing the number of possible programs with less than n bits ($< 2^n$) and the number of possible bitstrings with greater than n bits ($>> 2^n$), one observes: most strings s are complex (or "random", or "incompressible") in the sense that they cannot be computed by a program much shorter than s.

Invariance theorem. Due to the compiler theorem, $K_{U_1}(s) = K_{U_2}(s) + O(1)$ for two universal machines U_1 and U_2. Therefore we may choose one particular universal machine U and henceforth write $K(s) = K_U(s)$.

Machine learning, MDL, and the prior problem. In machine learning applications, we are often concerned with the following problem: given training data D, we would like to select the most probable hypothesis H generating the data. Bayes formula yields

$$P(H \mid D) = \frac{P(D \mid H)P(H)}{P(D)}. \tag{1}$$

We would like to select H such that $P(H \mid D)$ is maximal. This is equivalent to minimizing

$$-\,logP(H \mid D) = -logP(D \mid H) - logP(H) + logP(D). \qquad (2)$$

Let us interprete these equations. Since D is given, $P(D)$ may be viewed as a normalizing constant. $P(D|H)$ can usually be measured or at least approximated. According to classical information theory, $-log\ P(D|H)$ is the "optimal" (minimal, most efficient) code length or description length for D, given H. $-log\ P(H)$ is the minimal code length for H. This leads to the minimum description length (MDL) principle: *The best hypothesis for explaining the data is the one that minimizes the sum of the description length of the hypothesis and the description length of the data when encoded by the hypothesis.* But where does the prior $P(H)$ come from? How does one define an *a priori* probability distribution on the set of possible hypotheses without introducing arbitrariness? This is often perceived as the prior problem of Bayesian approaches. The theory of algorithmic probability, however, provides a solution.

Universal prior. Define $P_U(s)$, the *a priori probability* of a bitstring s, as the probability of guessing a (halting) program that computes s on U. Here, the way of guessing is defined by the following procedure: initially, the program tape consists of a single square. Whenever the scanning head of the program tape shifts to the right, do: (1) Append a new square. (2) With probability $\frac{1}{2}$ fill it with a 0; with probability $\frac{1}{2}$ fill it with a 1. We obtain

$$P_U(s) = \sum_{p: f_U(p)=s} (\frac{1}{2})^{|p|}.$$

Clearly, the sum of all probabilities of all halting programs cannot exceed 1 (no halting program can be the prefix of another one). But certain program tape contents may lead to non-halting computations.

Under different universal priors (based on different universal machines), probabilities of a given string differ by not more than a constant factor independent of the string size, due to the invariance theorem (the constant factor corresponds to the probability of guessing a compiler). Therefore we may drop the index U and write P instead of P_U. This justifies the name "*universal* prior", also known as the Solomonoff-Levin distribution. Universal priors appear to be the only convincing method for assigning probabilities to hypotheses (or other computable objects) in advance.

Algorithmic entropy. $-logP(s)$ is denoted by $H(s)$, the algorithmic entropy of s.

Dominance of shortest programs. It can be shown (the proof is non-trivial) that

$$K(s) = H(s) + O(1). \qquad (3)$$

Since $H(s) = -logP(s)$, this implies

$$P(s) = (\frac{1}{2})^{K(s)-O(1)} = 2^{O(1)}2^{-K(s)} = O(2^{-K(s)}).$$

The probability of guessing any of the programs computing some string and the probability of guessing its shortest program are essentially equal. The probability of a string is dominated by the probabilities of its shortest programs.

Inductive inference and Occam's razor. Occam's razor prefers solutions whose minimal descriptions are short over solutions whose minimal descriptions are longer. The "modern" prefix-based version of Solomonoff's theory of inductive inference justifies Occam's razor in the following way. Suppose the problem is to extrapolate a sequence of symbols (bits, without loss of generality). We have already observed a bitstring s and would like to predict the next bit. Let si denote the event "s is followed by symbol i" for $i \in \{0,1\}$. Bayes tells us

$$P(s0 \mid s) = \frac{P(s \mid s0)P(s0)}{P(s)} = \frac{P(s0)}{P(s)}, \quad P(s1 \mid s) = \frac{P(s1)}{P(s)}.$$

We are going to predict "the next bit will be 0" if $P(s0) > P(s1)$, and vice versa. Since $P(si) = O((\frac{1}{2})^{K(si)})$ for $i \in \{0,1\}$, the continuation with lower Kolmogorov complexity will (in general) be the more likely one.

Since Kolmogorov complexity is incomputable in general, the universal prior is so, too. A popular myth states that this fact renders useless the concepts of Kolmogorov complexity, as far as practical machine learning is concerned. But this is not so, as will be seen next. There we focus on a natural, computable, yet very general extension of Kolmogorov complexity.

Levin complexity. Let us slightly extend the notion of a program. In what follows, a program is a string on the program tape which can be scanned completely by U. The same program may lead to different results, depending on the runtime. For a given computable string, consider the log of the probability of guessing a program that computes it, plus the log of the corresponding runtime. The Levin complexity of the string is the minimal possible value of this. More formally, let U scan a program q (a program in the extended sense) written onto the program tape before it finishes printing s onto the work tape. Let $t(q, s)$ be the number of steps taken before s is printed. Then

$$Kt(s) = \min_{q}\{\mid q \mid + log\ t(q, s)\}.$$

An invariance theorem similar to the one for K holds for Kt as well.

Universal search. Suppose we have got a problem whose solution can be represented as a (bit)string. Levin's universal optimal search algorithm essentially generates and evaluates all strings (solution candidates) in order of their Kt complexity, until a solution is found. This is essentially equivalent to enumerating all programs in order of decreasing probabilities, divided by their runtimes. Each program computes a string that is tested to see whether it is a solution to the given problem. If so, the search is stopped. To get some intuition for universal search, let $P(s \mid p)$ denote the probability of computing a solution s from a halting program p on the program tape. For each p there is a given runtime $t(p)$. Now suppose the following gambling scenario. You bet on the success of some p. The bid is $t(p)$. To minimize your expected losses, you are going to bet on the p with maximal $\frac{P(s|p)}{t(p)}$. If you fail to hit a solution, you may bet again, but not on a p you already bet on. Continuing this procedure, you are going to list halting programs p in order of decreasing $\frac{P(s|p)}{t(p)}$. Of course, neither $P(s \mid p)$ nor $t(p)$ are usually known in advance. Still, for a broad class of problems, including P and NP problems and time-limited optimization problems, universal search can be shown to be optimal with respect to total expected search time, leaving aside a constant factor independent of the problem size: if string s can be computed within t time steps by a program p, and the probability of guessing p (as above) is $\bar{P}$, then within $O(\frac{t}{\bar{P}})$ time steps, systematic enumeration according to Levin will generate p, run it for t time steps, and output s. In the experiments below, a probabilistic algorithm strongly inspired by universal search will be used.

Conditional algorithmic complexity. The complexity measures above are actually simplifications of slightly more general measures. Suppose the universal machine U starts the computation process with a nonempty string x already written on its work tape. Conditional Kolmogorov complexity $K(s \mid x)$ is defined as

$$K(s \mid x) = min_p\{\mid p \mid: U\ computes\ s\ from\ p,\ given\ x\ on\ the\ worktape\}.$$

$Kt(s \mid x)$ is defined analoguously. Variants of conditional complexity are used to determine how much information some string conveys about another one. In the context of machine learning, conditional complexity measures how much additional information has to be acquired, given what is already known.

History spotlights / Selected references

In 1965, A. N. Kolmogorov (1903-1987), founder of modern axiomatic probability theory (Kolmogorov, 1933), was the first to introduce a variant of the complexity measure K for its own sake (Kolmogorov, 1965). Levin (1984) cites announcements of Kolmogorov's lectures on this subject dating back to 1961. In independent and even earlier work, R. J. Solomonoff (1964) had already come up with the same measure as a by-product of his work on algorithmic probability and inductive inference (a preliminary version of his paper is dated 1960). Both Solomonoff and Kolmogorov observed K's machine independence. Today, even Solomonoff himself refers to K as "Kolmogorov complexity", e.g. (Solomonoff, 1986). In 1969, G. J. Chaitin independently also published the essential concepts (Chaitin, 1969) (some hints were already provided at the end of his 1966 paper). Important related early work is described in (Martin-Löf, 1966; Gács, 1974; Schnorr, 1971). Apparently, L. A. Levin was the first to introduce and analyze today's "standard form" of Kolmogorov complexity based on halting programs and prefix codes (Levin, 1974), see also (Gács, 1974; Levin, 1973a; Levin, 1976; Zvonkin and Levin, 1970). Levin proved equation (3): $K(s) = H(s) + O(1)$. The importance of prefix codes was independently seen by Chaitin (1975), who also proved (3) and attributes part of the argument to N. Pippenger. Levin introduced Kt complexity and the universal optimal search algorithm (see e.g. (Levin, 1973b) and (Levin, 1984), where related ideas are attributed to Adleman – see also (Adleman, 1979)). Other generalizations of Kolmogorov complexity have been proposed, e.g. (Hartmanis, 1983), but see the contributions in (Watanabe, 1992) for more. Easily computable approximations of the MDL principle were formulated by Wallace and Boulton (1968) and Rissanen (1978, 1983, 1986). Such approximations build the basis of most if not all current machine learning applications, e.g. (Quinlan and Rivest, 1989; Gao and Li, 1989; Milosavljevič and Jurka, 1993; Pednault, 1989). Barzdin, referred to in (Zvonkin and Levin, 1970), related Kolmogorov complexity to a variant of Gödel's incompleteness theorem, a subject which became a central theme of Chaitin's research (Chaitin, 1987). Meanwhile, the theory of Kolmogorov complexity has split into many subfields. An excellent overview and many additional details on the history are given in Li and Vitanyi's book (1993). See also (Cover et al., 1989). See (Schmidhuber, 1994) for an application to fine arts. The presentation above is partly inspired by presentations found in (Chaitin, 1987), (Li and Vitányi, 1993), and (Solomonoff, 1986).

3 PROBABILISTIC SEARCH FOR USEFUL SELF-SIZING PROGRAMS WITH LOW LEVIN COMPLEXITY

Levin's universal search algorithm was considered of interest for theoretical purposes (see e.g. (Allender, 1992) and (Li and Vitányi, 1993)). However, it seems that nobody implemented it for experimental applications, perhaps in fear of the ominous "constant factor" which may be large. To my knowledge, general universal search was implemented for the first time during the project that led to this paper (Solomonoff (1986) himself apparently implemented restricted versions). In what follows, however, I will focus on the implementation of a slightly different probabilistic algorithm (also based on Levin complexity and strongly inspired by universal search). The experimental results obtained with the probabilistic algorithm (see section 4) are very similar to those obtained by the original universal search procedure (Schmidhuber and Jankowski, 1994).

Overview. The method described in this section searches and finds algorithms that compute solutions to a given problem specified by possibly very limited "training data". The goal is to discover solutions with high generalization performance on "test data" unavailable during the search phase. Towards this purpose, the probabilistic search algorithm randomly generates programs written in a general assembler-like programming language based on sequences of integers. Programs may influence their own storage size and runtime. Each program computes a solution candidate which is tested on the training data. The probability of generating a program p and an upper bound t_{max} for its runtime essentially equals the quotient of the probability of guessing p, and t_{max}. This implies that candidates with low Levin complexity are preferred over candidates with high Levin complexity. To measure generalization performance, candidates fitting the training data are evaluated on test data. In the experiments (section 4), solution candidates will be weight matrices for a neural net supposed to solve certain generalization tasks that are difficult or impossible to solve by conventional neural net algorithms.

"Universal" programming language. First we need a "universal" set of primitive instructions (called "primitives") that may be composed to form arbitrary algorithms for computing arbitrary (partial recursive) functions. The only limitation we are willing to accept is the storage size of our machine.

It is easy to devise "universal" sets of primitives. Which ones do we prefer? Informally, there is one general constraint to obey: whatever is computable on the used hardware, should be computable *just as efficiently* (up to a small constant factor) by a program composed from primitives. For instance, on a typical serial digital machine we would like to use primitives exploiting the fast storage addressing mechanisms. We would not want to limit ourselves to the simulation of, say, a slow one tape Turing machine. Likewise, on a machine with many parallel processors we would like to use a set of primitives allowing for programs with maximal parallelism. In what follows, the focus will be on conventional digital machines. Here is a description of the set-up used in the experiments to be described later:

Storage. Programs are sequences of integers. They are stored in the *storage*, consisting of a single array of cells. Each cell has an integer address in the interval $[-s_w, s_p]$. Both s_w and s_p are positive integers. The *program tape* is the set of cells with addresses in $[0, s_p]$. The *work tape* is the set of cells with addresses in $[-s_w, -1]$. Cells with non-negative addresses belong to the program tape. Cells with negative addresses belong to the work tape. The contents of the cell with address i is denoted by $c_i \in [-maxint, maxint]$, and is of type integer as well (in the implemented version, $maxint$ equals 10000). During execution of a program, the used portion of the program tape may increase. The used portion of the work tape may increase or decrease. At any time step, the variable Max ($-1 \leq Max \leq s_p$) denotes the topmost address of the used storage. The variable Min ($-s_w \leq Min \leq 0$) denotes its smallest address. At any given time, *legal addresses* are in the dynamic range $[Min, OracleAddress]$, where $OracleAddress = Max + 1$, by definition. At any given time, the integer sequence on the program tape (up to address Max) is called the current program. $Max = -1$ implies the "empty" program.

Instructions. At any given time, the variable *InstructionPointer* may equal the address of one of the cells, whose contents may be interpretable as an instruction. There are n_{ops} different possible instructions (in the implemented version, $n_{ops} = 13$). Each instruction is uniquely represented by an *instruction number* from the set $\{0, ..., n_{ops}-1\}$. An instruction may have up to three arguments (of type integer), or none. Arguments are stored in the addresses following the address of the instruction. For each argument of each instruction, there is a *legal argument range* (a set of integer values the argument is allowed to take on). Within certain limits, legal argument ranges can be dynamically modified by programs, as will be seen shortly.

Initialization, time limits, time probability. In the beginning of the execution of a "program" or "run", the variables *OracleAddress*, *InstructionPointer*, *Min*, and *CurrentRuntime* are all set to zero. The variable *CurrentTimeLimit* is used to define an upper bound for the runtime of the current program. To obtain a probabilistic variant of universal search, *CurrentTimeLimit* is chosen randomly as follows: elements from the set $\{0, 1\}$ are drawn with equal probability until the first "1" is drawn. Let n_t denote the number of trials. *CurrentTimeLimit* is set to $UnitTime \times 2^{n_t}$, where $UnitTime$ equals 16 time steps (each program will be allowed to execute at least 16 instructions – but it may choose to halt earlier). If *CurrentTimeLimit* exceeds $MaxTimeLimit$, then it is replaced by $MaxTimeLimit = 2^{24} = 16,777,216$. The *time probability* of the current program is defined by $max((\frac{1}{2})^{n_t}, \frac{UnitTime}{MaxTimeLimit})$. Short runtimes are more likely than long runtimes.

Instruction cycle and oracles. A single step of the *program interpreter* works as follows: if the *InstructionPointer* equals $OracleAddress$ ($= Max + 1$), then this is interpreted as the request for an oracle. A primitive and the corresponding arguments are chosen randomly from the set of legal options (to be described below). They are sequentially written onto the program tape, starting from $OracleAddress$. Max and $OracleAddress$ are increased accordingly, to reflect the growth of used program tape. Then the new primitive gets executed (except when growth beyond s_p halts the program). If there is no oracle request: if the *InstructionPointer* equals i, then if the content $c_i \in [0, n_{ops} - 1]$, the corresponding number of arguments n_i and the corresponding legal argument ranges are looked up and checked against the contents of the n_i addresses following the current address. If the instruction is "syntactically correct", it gets executed. Otherwise the current program is halted. If the executed primitive did not change the value of the *InstructionPointer* (e.g. by causing a jump), the *InstructionPointer* is set to point to the address following the address of (the last argument of) the current instruction. If an instruction was executed, *CurrentRuntime* is incremented. If the *CurrentTimeLimit* is reached, the program is halted.

Runs, programs, and space probability. After initialization, the instruction cycle is repeated until a halt situation is encountered. The *space probability* of a program is defined as the product of the probabilities of all parameters and primitives requested and executed during its runtime. Essentially, the space probability is the probability of guessing the executed content of the program tape.

Probabilistic search. Programs are generated randomly and executed as described above, and their results are evaluated until some problem-specific performance criterion is met. Obviously, results with low Levin complexity are preferred over results with high Levin complexity. (In an alternative implementation, the original universal search algorithm was used to systematically generate all solution candidates in order of their Levin complexities).

Used primitives. The instruction numbers and the semantics of the primitives used in the experiments are listed below. An expression of the form "address*i*" denotes the value (interpreted as an address) found in the ith cell following the one containing the current instruction (indirect addressing is used throughout). The following list assumes syntactical correctness of the instructions. Rules for legal argument ranges and syntactical correctness will be given shortly.

0 *Jumpleq(address1, address2, address3)*. If the contents of *address1* is less than or equal to the contents of *address2*, the *InstructionPointer* is set equal to *address3*.

1 *Output(...)*. A primitive for interaction with an external environment. It corresponds to the TM action of "writing the output tape" (see section 2). In the experiments, *"output"* will be called *"WriteWeight"*. It will be used to generate weights for a neural network. Variants of it will be specified where needed.

2 *Jump(address1)*. The *InstructionPointer* is set equal to *address1*.

3 *Stop()*. Halt the current program.

4 *Add(address1, address2, address3)*. The contents of *address1* is added to the contents of *address2*, the result is written into *address3*.

5 *GetInput(address1, address2)*. Another primitive for interaction with an external environment. It requires n_I separate "input fields" that may be modified by the environment (in the experiments, n_I will equal 20). *GetInput* reads the current value of the ith input field into *address2*, where i is the value found in *address1*. In conjunction with primitives changing the environmental state, *GetInput* provides an opportunity for exploiting the computing resources of the "outside world". In the applications below, however, *GetInput* will be pretty useless – all the input fields will remain zero all the time.

6 *Move(address1, address2)*. The contents of *address1* is copied to *address2*.

7 *Allocate(address1)*. The size of the work tape is increased by the value found in *address1*, the new cells are initialized with zeros. Min is updated accordingly (growth beyond $-s_w$ halts the program). No variable can be written before enough space has been *Allocated* on the work tape. As will be explained below, *Allocate* is essential for self-sizing programs.

8 *Increment(address1)*. The contents of *address1* is incremented.

9 *Decrement(address1)*. The contents of *address1* is decremented.

10 *Subtract(address1, address2, address3)*. The contents of *address1* is subtracted from the contents of *address2*, the result is written into *address3*.

11 *Multiply(address1, address2, address3)*. The contents of *address1* is multiplied by the contents of *address2*, the result is written into *address3*.

12 *Free(address1)*. The size of the work tape is decreased by the value found in *address1*. Min is updated accordingly. This primitive complements *Allocate*.

Rules for legal argument ranges and syntactical correctness. Jumps may lead to any address in the dynamic range $[Min, Max+1]$ (recall that $Max+1$ always equals the current value of *OracleAddress*). Operations that read the contents of certain cells (like *add, move, jumpleq* etc.) may read only from addresses in $[Min, Max]$. Operations that change the contents of certain cells may write only into work tape addresses in $[Min, -1]$. Thus, the program tape is "read/execute" only, except for random writes requested by moves of the *InstructionPointer* to *OracleAddress*. This makes reruns easy, as will be seen below. The work tape is "read/write/execute". Results of arithmetic operations leading to underflow or overflow are replaced by $-maxint$ or $maxint$, respectively. No more than 5 work tape cells may be *Allocated* or *Freed* at a time.

COMMENTS

1. Universality. It is not difficult to show that the above primitives form a universal set in the following sense: they can be composed to form programs writing any computable integer sequence onto the work tape (within the given size and range limitations). Note that the primitives make it easy to create programs for handling stacks, recursion, etc. A program may create executable code (or sub-programs) on the work tape. Since executable code is represented as a sequence of numbers, the code may modify itself. The scheme allows for very general sequential interaction with the environment (given appropriate problem-specific actions translating storage contents into output actions and environmental changes).

2. Self-sizing programs. The whole set-up is an adaptation of the Turing machine from section 2. It is designed to be more efficient in the sense that programs may exploit what conventional digital machines are good at: fast storage addressing, jumping, etc.[2] Using a single array of cells with negative addresses for the work tape and positive addresses for the program tape considerably simplifies the primitives. At first glance, there seems to be a price to pay for the general addressing capabilities: random jumps are unlikely to contribute to successful programs if there are many possible legal addresses to jump to. However, since programs may influence their own size and thus the number of available legal addresses, they have the potential to remain small and "likely", as will be seen next.

How do programs influence their own size? They can keep small by (1) avoiding requests for new oracles (e.g. by avoiding jumps to the current $OracleAddress$), and (2) by using *Allocate* and *Free* in a balanced way. Oracle requests, *Allocate* and *Free* provide the only ways of influencing the number of "visible" legal addresses available in used storage. The oracle requests are the only source of randomness, however. If the current program does not request many oracles, its space probability will tend to remain low, although the program may perform extensive computations. The bigger the used storage, however, the smaller the probability of guessing a particular "visible" address, and the less likely the arguments of instructions (like *"Jumleq"*) generated by future oracle requests. Small is beautiful.

3. Reruns. Since the program tape is "read/execute" only, we can rerun a randomly chosen halting program written onto the program tape by erasing the work tape, making the $InstructionPointer$ equal to 0, and starting the execution loop. The contents of the program tape completely determine the contents of the work tape at any given time (unless the environment reacts in a non-deterministic way). The desire for being able to rerun programs also is the reason for the way oracle requests are handled. One might have introduced an extra primitive *"RequestOracle"* calling for a random instruction to appear in some location, say, at the end of the used program tape. But then different reruns would have led to different oracles, in general. The way it is done, oracle requests are generated by any operation moving the $InstructionPointer$ to the top of the used program tape (the $OracleAddress$), and any oracle is executed immediately. (Thus, any legal instruction appearing on the program tape out of the blue is executed at least once. One might say that "randomness is not wasted" but handled efficiently.) During reruns, the same moves of the $InstructionPointer$ will not provoke oracle requests. This is because what used to be the $OracleAddress$ at some point of guessing the program, now isn't any more. During reruns, there is absolute determinism (unless the environment reacts in a non-deterministic way, of course).

4. Probabilistic setting. Why use a probabilistic search algorithm instead of the original universal search procedure? Why create time limits and programs randomly instead of systematically enumerating them? One reason is to avoid unintended bias. For instance, unintended bias may be introduced by imposing a systematic (say, alphabetic) order among programs with equal quotients of probability and runtime. A drawback of the probabilistic version above, however, is that programs with low Levin complexity (in general) will be tested more than once.

When speed is an issue, then we will prefer systematic enumeration, or a slightly more complicated probabilistic variant whose expected search time equals the one of systematic enumeration. Variants of systematic universal search based on the primitives above were implemented in collaboration with Norbert Jankowski (Schmidhuber and Jankowski, 1994). With the examples below, however, total

[2] In principle, it is possible to run a variant of universal search on a neural net architecture instead of a conventional digital machine. In earlier work, it was shown (in a different context) how neural nets may "talk about their own weights in terms of activations" and modify their own weight matrix (Schmidhuber, 1993b; Schmidhuber, 1993a). Such self-modifying capabilities can be used to form the basis of a universal set of primitives.

search time is not the main issue: the simulations in the next section (based on probabilistic search) are intended to highlight generalization performance, not speed. Very similar results were obtained by systematic search, however.

4 APPLICATION: FINDING "SIMPLE" NEURAL NETS

Neural networks are particularly well-studied instances of "generalizers", see e.g. (Maass, 1994; Baum and Haussler, 1989; Amari and Murata, 1993; Wolpert, 1993; Moody, 1992; Pearlmutter and Rosenfeld, 1991; Barron, 1988; Mozer and Smolensky, 1989) and the numerous references given below. For this reason, the simulations presented in this section focus on the task of finding algorithmically simple neural networks with high generalization capability. Let us first briefly look at a few rather recent definitions of "simplicity" used in supervised neural net training algorithms. In what follows, "solutions" are weight vectors of neural nets.

4.1 PREVIOUS ALGORITHMS FOR MAKING NETS "SIMPLE"

- *Weight decay.* A special error term (in addition to the standard term enforcing matches between desired outputs and actual outputs) encourages weights close to zero. The idea is that a zero weight does not cost many bits to be specified, thus being "simple" (Hinton and van Camp, 1993). Pearlmutter and Hinton were probably the first to propose weight decay, while Rumelhart was perhaps the first to suggest its use for reducing overfitting. Variants of weight decay were successfully applied by Weigend et al. (1990), Krogh and Hertz (1992), and others.

- *Soft weight sharing.* Nowlan and Hinton (1992) introduce an additional objective function encouraging groups of weights with nearly equal values. The weights are taken to be generated by mixtures of Gaussians. The fewer the number of Gaussians and the closer some weight is to the center of some Gaussian, the higher its probability, and the fewer bits are needed to encode it (according to classical information theory (Shannon, 1948)).

- *Bayesian strategies for backprop nets.* MacKay evaluates hyper-parameters (such as weight-decay rates) with respect to their probabilities of generating the observed data (MacKay, 1992). Estimates of the probabilities are computed on the basis of Gaussian assumptions.

- *Optimal Brain Surgeon.* Hassibi and Storck (1993) use second order information to obtain "simple" nets by pruning weights whose influence on the error is minimal, and changing other weights to compensate. See (LeCun et al., 1991) for a related approach. See (Vapnik, 1992) and (Guyon et al., 1992) for some theoretical analysis.

- *"Non-algorithmic" MDL methods based on Gaussian priors.* To minimize the sum of the description lengths of a neural net and its errors, Hinton and van Camp (1993) assume Gaussian weight priors and Gaussian error distributions, and minimize the asymmetric divergence (or Kullback-Leiber distance) between prior and posterior after training.

- *Methods for finding "flat" minima.* Hochreiter and Schmidhuber (1994) use efficient second order methods to search for large connected regions of "acceptable" error minima. This corresponds to "simple" networks with low description length and low expected overfitting.

- *"Mutual information networks".* Deco et al. (1993) measure network complexity with respect to given data by measuring the mutual information between inputs and internal representations extracted by the hidden units.

- *Methods for removing redundant information from input data.* If the input data can be compressed, the networks processing the data can be made smaller (and simpler), in general. From the standpoint of classical information theory, an optimal compression algorithm is one that builds a factorial code of the input data (a code with statistically independent components, e.g. (Barlow, 1989)). Various "neural" methods for compressing input data are known. See (Schmidhuber, 1992b) for a "neural" method designed to generate factorial codes. See (Atick et al., 1992) for a focus on visual inputs. See (Schmidhuber, 1992a) for loss-free sequence compression. See (Becker, 1991) for numerous additional references.

There are also numerous heuristic *constructive methods*, where network size grows in case of underfitting the training data. MDL approaches in other areas of machine learning include (Quinlan and Rivest, 1989; Gao and Li, 1989; Milosavljevič and Jurka, 1993; Pednault, 1989). Among the implemented methods, neither the neural net approaches nor the other ones are general in the sense of Solomonoff, Kolmogorov, and Levin. All the previous implementations use measures for "simplicity" that lack the universality and elegance of those based on Kolmogorov complexity and algorithmic information theory. Many previous approaches are based on ad-hoc (usually Gaussian) priors.

The remainder of this paper is mostly devoted to simulations of the more general method based on the universal prior, self-sizing programs, and the probabilistic search algorithm preferring candidates with low Levin complexity over candidates with high Levin complexity. With certain seemingly trivial but actually non-trivial toy problems it will be demonstrated that the approach can lead to generalization results unmatchable by more traditional neural net algorithms. It should be mentioned, however, that this does not say much about the applicability of the method to real world tasks.

4.2 GENERALIZATION TASKS: SIMULATIONS

In the experiments, the following values were used for maximal program tape size and work tape size: $s_p = 100$, $s_w = 1000$. The current implementation (which is not optimized for speed) tests about 3,000 programs per second on a SUN SPARC ELC. On average, a program runs for not many more than 10 time steps before halting or being halted. But there are programs running for millions of time steps, of course.

4.3 A PERCEPTRON FOR COUNTING INPUTS

The following pattern association task may seem trivial but will be made difficult (for traditional approaches) by providing only very few training examples.

The task. A linear (perceptron-like) network with 100 input units, one output unit, and 100 weights, is fed with 100-dimensional binary input vectors. x^p denotes the p-th input vector. x_i^p denotes the ith component of x^p, where i ranges from 0 to 99. Each input vector has exactly three bits set to one, all the other bits are set to zero. Obviously, there are $\binom{100}{3} = 161,700$ possible inputs. The network's output in response to x^p is

$$y^p = \sum w_i x_i^p,$$

where w_i is the i-th weight. Each weight may take on integer values between -10000 and 10000. The task is to find weights such that y^p equals the number of *on*-bits in x^p, for all 161,700 possible x^p. The number of solution candidates in the search space of possible weight vectors is huge: 20001^{100}. This is too much for exhaustive search.

The solution. The only solution to the problem is: make all w_i equal to 1. The Kolmogorov complexity of this solution is small, since there is a short program that computes it. Its Levin complexity is small, too, since its "logical depth" (the runtime of its shortest program (Bennett, 1988)) is less than 400 time steps.

The difficulty. If the training set is very small (e.g. if there are just four or five training examples), then conventional perceptron algorithms will not solve this apparently simple problem. They will not achieve good generalization on unseen test data. One reason is that connections from units that are always off won't be changed at all by conventional gradient descent algorithms, e.g. (Werbos, 1974; LeCun, 1985; Parker, 1985; Rumelhart et al., 1986). Note, however, that scaling the inputs differently

Addresses:	0	1	2	3	4	5
Contents:	1	1	0	1	1	0
Interpretation:	WriteWeight	1	jumpleq	1	1	0

Table 1: *A program for the counting perceptron.*

is not going to improve matters. Nor is weight decay. Weight decay encourages weight matrices with many zero entries. For the current task, this is a bad strategy.

The training data. To illustrate the generalization capability of search for solution candidates with low Levin complexity, **only 3 training examples are used**. They were randomly chosen from the $161,700$ possible inputs. The first training example is the binary vector x^1 with *on*-bits at the positions 5, 17, and 86 (and *off*-bits everywhere else). The second one, x^2, has *on*-bits at the positions 13, 55, and 58. The third one, x^3, has *on*-bits at the positions 40, 87, and 94. In all three cases, the desired output (target) is 3. Generalization results (to be described below) obtained with this particular training set are very similar to those obtained with different sets of 3 training examples (created by randomly permuting the input units that are never on).

The search procedure is as follows: the probabilistic search algorithm (as described in section 3) lists and executes programs computing solution candidates (weight vectors). The primitive *"WriteWeight"* (replacing *"output"*, see section 3) is used for writing network weights. It has one argument and uses the variable *WeightPointer* taking on values from the set $\{0, 1, ..., 99\}$. In the beginning of a run, *WeightPointer* and all weights are initialized to 0. The instruction number and the semantics of *"WriteWeight"* are as follows (compare the list of primitives given in section 3):

1 *WriteWeight(address)*. $w_{WeightPointer}$ is set equal to the contents of *address*[3]. The variable *WeightPointer* is incremented. Halt if *WeightPointer* out of range.

Only if the solution candidate fits the training data exactly is the solution tested on the test data. Note that this is like a "reward-only-at-goal" task: The measure of success is binary – either the network fits all the training data, or it doesn't. There is no teacher providing a more informative error signal (such as the distance to the desired outputs).

RESULTS. Programs fitting the 3 training exemplars were found in 20 out of 100000 runs. **Only 2 of them did not lead to perfect generalization on the $\binom{100}{3} - 3 = 161,697$ unseen test examples.**

The first weight vector fitting the training data was found after 904 runs. The corresponding program was a "wild" one, allocating a lot of space and executing many useless instructions, but still leading to perfect generalization on all the unseen test data. Before halting, the program used 702 out of 1024 allocated time steps. Its time probability was 2^{-6} (recall that the unit time is only 16 time steps). Its space probability was $2.8 * 10^{-18}$.

Another weight vector fitting the training data was computed during the 6038th run. The corresponding program is given in table 1. Here is a more readable interpretation (each program instruction is preceded by its address):

```
(0) Write the contents of address 1 (which is 1) onto the weight pointed to by
WeightPointer and increment weight pointer.  Halt if WeightPointer out of range.
(2) If the contents of address 1 is less or equal to the contents of address 1,
goto address 0.
```

Since the condition tested in the second instruction is always true, this little program will write down a correct solution, given enough time. It requires 201 time steps. In the case above it got more than enough time: the randomly chosen time limit was $16 * 2^{12} = 65536$.

[3]To allow for real-valued weights, set $w_{WeightPointer}$ equal to the contents of *address*, divided by 1000, say.

Addresses:	0	1	2	3	4	5	6	7
Contents:	1	0	1	0	0	5	5	0
Interpretation:	WriteWeight	0	WriteWeight	0	jumpleq	5	5	0

Table 2: *A faster program for the counting perceptron.*

After 351,168 runs, the system came up with a faster program. See table 2. The program does this:

```
(0) Write the contents of address 0 (which happens to be 1, due to the code of
``write'' being 1) onto the weight pointed to by WeightPointer and increment
WeightPointer.  Halt if WeightPointer out of range.
(2) Write the contents of address 0 onto the weight pointed to by WeightPointer
and increment WeightPointer.  Halt if WeightPointer out of range.
(4) If the contents of address 5 (which happens to be 5) is less than or equal
to the contents of address 5 (this is always true), goto address 0.
```

This program writes two times before jumping, thus reducing runtime from 200 to 150 time steps (recall that the execution of each instruction, including jumps, takes one time step). Its space probability is $9.88 * 10^{-9}$. **Other successful programs with exactly the same runtime were found in 7 out of 10^6 runs**. No faster programs were found.

Comment. With the example above, probabilistic search among self-sizing programs leads to excellent generalization performance. At least in theory, however, it might be possible that an appropriate variant of Nowlan's and Hinton's approach (1992) might achieve good generalization performance on this task, too. Recall from section 3 that Nowlan and Hinton encourage groups of weights with equal values, which is a good strategy in the case above. For this reason, the following task requires *that no two weights have equal values.* The Kolmogorov complexity of the solution, however, will again be low.

4.4 A PERCEPTRON FOR ADDING INPUT POSITIONS

The task. We use the same perceptron-like network and the same input data as above. The goal is different, however. The task is to find weights such that y^p equals the *sum* of the positions of *on*-bits in x^p, for all $\binom{100}{3} = 161,700$ possible x^p. Again, the task will be made difficult by providing only very limited training data.

The solution. The only solution to the problem is: make all w_i equal to i. Like with the example above, there are short and fast programs for computing the solution.

The training data. The 3 training inputs x^1, x^2, and x^3 from the previous task are used. The target values are different, however. Obviously, the target for input vector x^1 is 108. The target for input vector x^2 is 126. The target for input vector x^3 is 221. Again, success is binary: only if the solution candidate fits the 3 training examples exactly, the solution is evaluated on the test data. Note that conventional perceptron algorithms cannot solve this generalization problem.

RESULTS. Programs fitting the training data were found in 10 out of $5.5 * 10^7$ runs, using up a total search time of $8.14 * 10^8$ time steps. **Only 2 of the 10 successful runs did not lead to perfect generalization on the 161,697 unseen test examples.**

The first weight vector fitting the training data was found after 6,902,963 runs. Again, the corresponding program was a pretty wild one. *But it led to perfect generalization on all the test data.* Before halting, the program used 502 out of 8192 allocated time steps. Its time probability was 2^{-8}. Its space probability was $3.92 * 10^{-16}$. Table 3 shows the used part of the storage after execution. What the program does is this:

Addresses:	-3	-2	-1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Contents:	100	0	0	7	3	5	11	-2	1	-3	0	-3	9	11	8	-3	2	-1

Table 3: *Used storage after execution of a program for the adding perceptron.*

Addresses:	-1	0	1	2	3	4	5	6	7	8	9
Contents:	100	7	1	1	-1	8	-1	0	1	2	2

Table 4: *Used storage after execution of a more elegant program for the adding perceptron.*

```
(0) Allocate 3 cells on the work tape.  Initialize with zero.  Set Min = Min - 3.
(2) Get the contents of the input field (see list of instructions in section 3)
at position 11 (which is 0), and write it into address -2.
(5) Write the contents of address -3 onto the weight pointed to by WeightPointer
and increment WeightPointer.  Halt if WeightPointer is out of range.
(7) If the contents of address -3 is less or equal to the contents of address 9,
goto address 11.  Otherwise goto address 11.
(11) Increment the contents of address -3.
(13) Goto address -1.
(-1) If the contents of address 7 is less or equal to the contents of address 3
(always true), goto address 5.
```

The instructions beginning at the addresses (2), (7), and (-1) are useless. But at least they are not catastrophic. Essentially, the program first allocates space for a variable (initially zero) on the work tape (recall that the program tape is "read/execute" only, and cannot be used for variables). Then it executes a loop for incrementing and writing the variable contents onto the network's weight vector.

After $4.6 * 10^7$ runs, a faster and rather elegant (nearly minimal) program was found. Table 4 shows the used storage after execution. The program ran for 302 out of 512 allocated time steps. Its space probability is $9.65 * 10^{-10}$. Inspection will reveal the operation of the program.

Using different sets of 3 training examples (obtained by randomly permuting the input units that are never on) led to very similar generalization results.

4.5 INDEXING WRITE OPERATIONS

Clearly, the choice of primitives affects the probabilities of solutions. Algorithmic information theory tells us that this delays optimal search by no more than a constant factor. As long as the primitives form a universal set, primitives from another set can be composed from them. Still, constant factors may be large. This subsection repeats the experiment from section 4.4 with a slightly different set of primitives increasing the constant factor.

The primitive *"WriteWeight"* is redefined and gets an additional argument. The primitive *"GetInput"* is redefined and gets a new name: *"ReadWeight"*. There is no separate *WeightPointer* any more, and no automatic increment mechanism for *WeightPointer's* position. Instead, the new primitives may directly address, read and write the network's weights. The other primitives remain the same. Here are the two new ones, together with their instruction numbers (compare section 5):

1 *WriteWeight(address1, address2).* w_i is set equal to the contents of *address1*, where i is the value found in *address2*.

Addresses:	-100	-99	...	-3	-2	-1	0	1	2	3	4	5	6	7	8
Contents:	0	0	...	0	0	100	7	1	8	-1	1	-1	-1	2	0

Table 5: *Used storage after execution of another program for the adding perceptron, using a different "WriteWeight" primitive.*

5 *ReadWeight(address1, address2)*. w_i is written into the address found at location *address1*, where i is the value found in *address2*.

Appropriate syntax checks halt programs whenever they attempt to do something impossible, like writing a non-existent weight. Since the new *"WriteWeight"* primitive has an additional argument (to be guessed correctly), successful programs tend to be less likely.

RESULTS. Out of 10^8 runs using up a total search time of $1.436 * 10^9$ time steps, 3 runs generated weight vectors fitting the training data. **All of them allowed for perfect generalization on all the test data**. During execution, one of them filled the storage as seen in table 5. The program ran for 399 out of 1024 allocated time steps. Its space probability was $9.95 * 10^{-9}$. What it does is this:

(0) Allocate one cell on the work tape. Initialize with zero. Set Min = Min-1.

(2) Increment the contents of address -1.

(4) Make $w_{c_{-1}}$ equal to the contents of address -1.

(7) Jump to address 0.

Repeated execution of the instruction at address 0 unnecessarily allocates 100 cells of the work tape but does not do any damage other than slightly slowing down the program.

Using different sets of 3 training examples (obtained by randomly permuting the input units that are never on), led to very similar generalization results.

5 INCREMENTAL SEARCH

As seen above, the probabilistic search algorithm inspired by universal search can lead to excellent generalization performance. However, the tasks above are not very typical in the sense that the learning system does not receive any feedback about its progress. The success criterion is binary: either the system solves the task, or it doesn't. In "cruel" environments providing nothing but such limited evaluative feedback, not much can be done. For such cases, Levin's algorithm is indeed optimal.

With many typical learning situations in the real world, however, there is more informative feedback. For instance, "supervised" gradient-based neural net algorithms like back-prop (Werbos, 1974; LeCun, 1985; Parker, 1985; Rumelhart et al., 1986) make use of information provided by error signals (distances between actual network outputs and target values). Unlike universal search, these algorithms *incrementally* adjust network weights in an iterative manner: solution candidates found in previous trials serve as a basis for additional improvements. Reinforcement learning algorithms (Watkins, 1989; Dayan and Sejnowski, 1994; Barto et al., 1983; Williams, 1988; Schmidhuber, 1989) (see Barto (1989) for an overview) receive less informative environmental feedback than supervised learning algorithms, but they are designed to work in an incremental fashion as well. For instance, they tend to make use of the information provided by the *magnitude* of the rewards, and by the amount of time between rewarding events. Again, "good" solutions build the basis for "better" solutions. The same is true for simple hill-climbing and for "evolutionary" and "genetic" algorithms (GAs) (Rechenberg, 1971; Schwefel, 1974; Holland, 1975; Hoffmeister and Bäck, 1991)(see e.g. (Dickmanns et al., 1986; Schmidhuber, 1987; Koza, 1992) for applications of the GA paradigm to the evolution of computer programs).

The original universal search procedure as formulated by Levin is not designed for incremental learning situations. Indeed, the current theory of incremental learning is not well-developed. However, there appears to be more than one reasonable way of appropriately extending universal search. Some possibilities are given in Solomonoff's and Paul's more recent work, see (Solomonoff, 1986; Solomonoff, 1990; Paul and Solomonoff, 1991). Apparently, however, nobody has *implemented* incremental extensions of universal search so far, although both Solomonoff and Paul emphasize the importance of experiments.

An in-depth study of extensions designed for incremental learning is beyond the scope of this paper. To end it with a promising outlook, however, a few initial experimental results with certain probabilistic variants of the first implemented incremental extensions will be reported next.

5.1 EXPERIMENTS WITH INCREMENTAL EXTENSIONS

The basic set-up for the experiment with the network described in section 4.4 (the "adding" perceptron) is used again. However, to allow for "incremental learning", the following modifications are introduced.

Generation of "mutations" with low Levin complexity. To provide more informative feedback, weight vectors are evaluated in a non-binary fashion. The "fitness" of a weight vector is defined as its number of "correct" weights. Recall that the solution requires the ith weight to equal i. The weights (initially zero) are not re-initialized after each run. Instead, whenever there is an improvement (whenever some run leads to a weight vector with more correct weights than the best found so far), the weights of the modified weight vector are stored. Further runs try to generate further improvements of the modified weight vector. In other words, each run leads to a *"mutation"* of the best weight vector found so far. Essentially, weight vector mutations are listed in order of their Levin complexity, until an improvement is found. The improved weight vector goes into a new round of mutations.

Mutations of mutation algorithms. With the problem from section 4.5, the above modification by itself did not lead to a significant reduction of total search time. Typical improvements led to at most one additional correct weight per run (with most runs, there was no improvement at all). However, by introducing another modification, search time sometimes was reduced dramatically. Let us define the "fitness" of an improvement as the difference between the fitness of the newly generated weight vector and the best found so far. The additional modification is this: whenever the fitness of an improvement exceeds both 2 and the fitness of the best improvement so far, the corresponding mutation program is kept on the program tape. The work tape is erased. The following trials start with *InstructionPointer* being equal to the address following the end of the successful program ($Min \leftarrow 0$, *OracleAddress* $\leftarrow$ *InstructionPointer*). This means: **new mutation programs may build on earlier successful mutation programs**. This makes the approach similar in spirit to the approaches proposed in (Solomonoff, 1986; Solomonoff, 1990; Paul and Solomonoff, 1991). Successful programs often will represent short descriptions of mutations of many different solution components. The probability of a new successful mutation may be higher if we may mutate successful mutation algorithms (instead of just mutating mutation results). Thus, additional improvements may be more likely. Think of this: most programmers prefer rewriting programs in a high level language instead of rewriting the microcode.

RESULTS. With the first test, 25 of the first 1,356,777 runs led to 1-step improvements. This means that about one out of 54,271 runs led to a "better" weight vector (whose number of correct weights exceeded the one of the best found so far by exactly 1). In the end of this period, the network's weight vector had 25 correct weights. Then, at run number 1,356,777, there was a dramatic improvement leading to 57 correct weights. As described above, the corresponding mutation program was left on the program tape. Its space probability was $1.47 * 10^{-21}$. Very briefly after this event, at run number 1,357,193, the system generated an additional dramatic improvement. The additional code was just a jump to a useful position in the old code. Together with the old code, the new program led to 99 correct weights. The space probability of the additional code was high: $3.3 * 10^{-3}$ (this is the reason why it was found so quickly). The only missing correct weight was generated shortly after that, at run number 1,357,233. Thus, only 456 runs after the first dramatic improvement, the solution was completed. This corresponds to not more than a small fraction of a second of additional cpu time.

With the second test, again there was a series of at most 1-step improvements. This time it lasted until the network's weight vector had 46 correct weights. Then, at run number 6,308,386 (after about half an hour of cpu time), there was an apparently minor improvement leading to 49 correct weights.

But the minor improvement was actually a major breakthrough. The corresponding mutation program became the building block for a flurry of additional improvements. Nearly immediately afterwards, after run number 6,308,631, there were 53 correct weights. After run number 6,308,812, there were 68 correct weights. After run number 6,310,125, there were 99 correct weights. After run number 6,310,280, there were 100 correct weights. Thus, within only 1894 runs (less than a second of cpu time) following the apparently minor improvement, the solution was completed.

In additional experiments, the weight vector was reinitialized (with zeros) after each run. But programs leading to further improvements were not erased from the program tape, just as described above. On average, this led to equal or even better performance than the version keeping the best weight vector so far. Similar observations were made with other variants of incremental search.

COMMENTS / PROBLEMS / FUTURE WORK

1. Learning speed. The incremental extensions turned out to be faster than non-incremental search (compare section 4.5). Obviously, once a useful program is found, it may serve as a useful subprogram. This may dramatically increase the probability of further improvements, and thus reduce search time. In a way, the system may learn how to learn faster.

2. On improving "evolutionary" algorithms. In theory, the strategy of listing parameter *mutations* in order of their Levin complexity appears to be a smarter mutation strategy than the trivial mutation strategies employed by conventional hill-climbing, evolutionary and genetic algorithms, e.g. (Rechenberg, 1971; Schwefel, 1974; Holland, 1975; Hoffmeister and Bäck, 1991; Dickmanns et al., 1986; Koza, 1992). In general, the latter cannot be expected to come up (within reasonable time) with non-trivial changes that require many simultaneous "correlated mutations" in quite different positions (at every 5th position, say). Universal search, however, soon will find useful "correlated mutations" if their Levin complexities are low. Therefore, incremental extensions of universal search appear to be promising candidates for learning more complex tasks, and for replacing the less sophisticated strategies typically used for more traditional algorithms.

3. Code explosion. In theoretical investigations, more complex strategies for handling "subprograms" have been proposed. Solomonoff described methods for giving new names to successful programs, and using them as more complex primitives (Solomonoff, 1964; Solomonoff, 1986). This approach suffers from the same obvious problem as the methods tested above: as the code continues to expand, there is more material with which to form new programs (this will be referred to as the *"code explosion problem")*. Sometimes, code explosion may have a negative influence on the probability of additional successful code. For such reasons, Paul and Solomonoff (1991) address theoretical advantages of grouping related programs into "directories" of subprograms.

4. Compressing successful programs. To a degree, the severeness of the code explosion problem might be diminishable by searching for programs compressing old successful code. Solomonoff (1986) proposed to spend about one half of total search time on trying to compress previous useful programs, but this idea was not pursued (the focus of his paper is on combining concepts from algorithmic probability theory and more traditional approaches for assigning modified probabilities to subprograms appearing in successful programs). Program compression algorithms could help to deal with a related problem observed in the simulations: in general, the programs found by incremental search were not as short, efficient, and elegant as the ones found by non-incremental search. The reason is that incremental search tends to generate programs that incorporate non-optimal code from previous runs.

5. Future research. Much remains to be done to become clear about the mutual advantages and disadvantages of different "incremental" extensions of universal search. At the moment, nobody knows the best general algorithm for learning from previous experiences. Is there a strategy for incremental learning that is optimal in the same sense universal search is optimal for a broad class of non-incremental situations? This may be one of the most important questions in machine learning.

6 CONCLUDING REMARKS

It was shown that basic concepts from the theory of algorithmic complexity are of interest for machine learning purposes. At least with certain toy problems where it is computationally feasible, search with preference for solutions computable by short and fast programs may lead to excellent generalization performance unmatchable by more traditional algorithms. Although the focus of the experiments was on perceptron-like neural nets, the presented methods are general enough to be applied to a wide variety of problems. For instance, in work done in collaboration with Norbert Jankowski, variants of universal search were successfully applied to path finding problems in mazes (Schmidhuber and Jankowski, 1994). Much work on "incremental" learning in real world applications remains to be done, however.

The bias towards algorithmic simplicity is a very general one. It is weaker than most kinds of problem specific inductive bias, e.g. (Utgoff, 1986; Haussler, 1988). If a solution is indeed simple, the bias is justified (it does not require us to know "the way in which the solution is simple"). If the solution is *not* simple, the bias towards algorithmic simplicity won't do much damage: even in case of algorithmically complex solutions we cannot lose much if we focus on simple candidates first, before looking at more complex candidates. This is because in general the complex candidates greatly outnumber the simple ones. The few simple ones don't significantly affect total search time of an optimal search algorithm.

When will a general bias towards algorithmic simplicity not only cause no harm but also be *useful* for problem solving? How many solutions are indeed simple? The next paragraph appears to support the answer "hardly any". But the final part of this section argues that the expression "hardly any" actually refers to a worst case that is *atypical* for real world problems.

In general, generalization is impossible. To be more specific, let the task be to learn some relation between finite bitstrings and finite bitstrings. A training set is chosen randomly. In almost all cases, the shortest algorithm computing a (non-overlapping) test set essentially will have the size of the whole test set (recall from section 2 that most computable objects are incompressible). The shortest algorithm computing the test set, given the training set, won't be any shorter. In other words, the "mutual algorithmic information" (e.g. (Chaitin, 1987)) between test set and training set will be zero in almost all cases (ignoring an additive constant independent of the problem). Therefore, in the general case, (1) knowledge of the training set does not provide any clues about the test set, (2) there is no hope for generalization, and (3) obviously there is no reason why a "simple" (or any other kind of) solution should be preferred *a priori* over complex ones (related observations are discussed at length e.g. in (Dietterich, 1989; Schaffer, 1993; Wolpert, 1993)). This may be viewed as the reason why certain worst-case results of PAC-learning theory (initiated by Valiant, 1984) appear discouraging. Similarly for problem solving in general: a "problem" is usually defined by a search space of solution candidates, and a computable criterion for the solution. Most solutions to problems from the set of all possible well-defined problems are algorithmically complex (random, incompressible). Most such problems cannot be efficiently solved ("efficient" means faster than by exhaustive search), neither by Levin's universal search algorithm, nor by a hypothetical "optimal" incremental learning scheme, nor by any other method.

Apparently, however, many *typical* problems we are confronted with in the "real world" are simple! Simple in the sense that their solutions do not require as much information to be specified as most solution candidates. Problems that humans consider to be *typical* are *atypical* when compared to the general set of all well-defined problems (see also (Li and Vitányi, 1989)). Indeed, for all "interesting" problems, the bias towards algorithmic simplicity seems justified!

This may be a miracle. Or perhaps a consequence of the possibility that our universe is run by a short algorithm (every electron behaves the same way). Or (at least in some cases) just a consequence of the fact that we select only problems we can solve (we would not exist if we could not survive by doing so – but this is an anthropocentric argument). Anyway, our learning machines should try to make use of the enormous amount of algorithmic redundancy in our "friendly" universe. The most general way of doing so appears to be to use the tools provided by the theory of algorithmic probability and Kolmogorov complexity.

7 ACKNOWLEDGEMENTS

Thanks to Martin Eldracher, Sepp Hochreiter, Margit Kinder, Daniel Prelinger, Mark Ring, Jan Storck, Gerhard Weiß, and especially to Ray Solomonoff for useful comments on earlier drafts of this paper.

References

Adleman, L. (1979). Time, space, and randomness. Technical Report MIT/LCS/79/TM-131, Laboratory for Computer Science, MIT.

Allender, A. (1992). Application of time-bounded Kolmogorov complexity in complexity theory. In Watanabe, O., editor, *Kolmogorov complexity and computational complexity*, pages 6–22. EATCS Monographs on Theoretical Computer Science, Springer.

Amari, S. and Murata, N. (1993). Statistical theory of learning curves under entropic loss criterion. *Neural Computation*, 5(1):140–153.

Atick, J. J., Li, Z., and Redlich, A. N. (1992). Understanding retinal color coding from first prinicples. *Neural Computation*, 4:559–572.

Barlow, H. B. (1989). Unsupervised learning. *Neural Computation*, 1(3):295–311.

Barron, A. R. (1988). Complexity regularization with application to artificial neural networks. In *Nonparametric Functional Estimation and Related Topics*, pages 561–576. Kluwer Academic Publishers.

Barto, A. G. (1989). Connectionist approaches for control. Technical Report COINS Technical Report 89-89, University of Massachusetts, Amherst MA 01003.

Barto, A. G., Sutton, R. S., and Anderson, C. W. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13:834–846.

Barzdin, Y. M. (1988). Algorithmic information theory. In Reidel, D., editor, *Encyclopaedia of Mathematics*, volume 1, pages 140–142. Kluwer Academic Publishers.

Baum, E. B. and Haussler, D. (1989). What size net gives valid generalization? *Neural Computation*, 1(1):151–160.

Becker, S. (1991). Unsupervised learning procedures for neural networks. *International Journal of Neural Systems*, 2(1 & 2):17–33.

Bennett, C. H. (1988). Logical depth and physical complexity. In *The Universal Turing Machine: A Half Century Survey*, volume 1, pages 227–258. Oxford University Press, Oxford and Kammerer & Unverzagt, Hamburg.

Blumer, A., Ehrenfeucht, A., Haussler, D., and Warmuth, M. K. (1987). Occam's razor. *Information Processing Letters*, 24:377–380.

Chaitin, G. (1966). On the length of programs for computing finite binary sequences. *Journal of the ACM*, 13:547–569.

Chaitin, G. (1969). On the length of programs for computing finite binary sequences: statistical considerations. *Journal of the ACM*, 16:145–159.

Chaitin, G. (1975). A theory of program size formally identical to information theory. *Journal of the ACM*, 22:329–340.

Chaitin, G. (1987). *Algorithmic Information Theory.* Cambridge University Press, Cambridge.

Cover, T. M., Gács, P., and Gray, R. M. (1989). Kolmogorov's contributions to information theory and algorithmic complexity. *Annals of Probability Theory*, 17:840–865.

Dayan, P. and Sejnowski, T. (1994). TD(λ): Convergence with probability 1. *Machine Learning.* In press.

Deco, G., Finnoff, W., and Zimmermann, H. G. (1993). Elimination of overtraining by a mutual information network. In *Proceedings of the International Conference on Artificial Neural Networks, Amsterdam*, pages 744–749. Springer.

Dickmanns, D., Schmidhuber, J., and Winklhofer, A. (1986). Der genetische Algorithmus: Eine Implementierung in Prolog. Fortgeschrittenenpraktikum, Institut für Informatik, Lehrstuhl Prof. Radig, Technische Universität München.

Dietterich, T. G. (1989). Limitations of inductive learning. In *Proceedings of the Sixth International Workshop on Machine Learning, Ithaca, NY*, pages 124–128. San Francisco, CA: Morgan Kaufmann.

Gács, P. (1974). On the symmetry of algorithmic information. *Soviet Math. Dokl.*, 15:1477–1480.

Gao, Q. and Li, M. (1989). The minimum description length principle and its application to online learning of handprinted characters. In *Proc. 11th IEEE International Joint Conference on Artificial Intelligence, Detroit, Mi*, pages 843–848.

Guyon, I., Vapnik, V., Boser, B., Bottou, L., and Solla, S. A. (1992). Structural risk minimization for character recognition. In Lippman, D. S., Moody, J. E., and Touretzky, D. S., editors, *Advances in Neural Information Processing Systems 4*, pages 471–479. San Mateo, CA: Morgan Kaufmann.

Hartmanis, J. (1983). Generalized Kolmogorov complexity and the structure of feasible computations. In *Proc. 24th IEEE Symposium on Foundations of Computer Science*, pages 439–445.

Hassibi, B. and Stork, D. G. (1993). Second order derivatives for network pruning: Optimal brain surgeon. In Lippman, D. S., Moody, J. E., and Touretzky, D. S., editors, *Advances in Neural Information Processing Systems 5*, pages 164–171. San Mateo, CA: Morgan Kaufmann.

Haussler, D. (1988). Quantifying inductive bias: AI learning algorithms and Valiant's learning framework. *Artificial Intelligence*, 36:177–221.

Hinton, G. E. and van Camp, D. (1993). Keeping neural networks simple. In *Proceedings of the International Conference on Artificial Neural Networks, Amsterdam*, pages 11–18. Springer.

Hochreiter, S. and Schmidhuber, J. (1994). Simplifying networks by discovering "flat" minima. Technical Report FKI- -94, Fakultät für Informatik, Technische Universität München. To be presented at NIPS'94.

Hoffmeister, F. and Bäck, T. (1991). Genetic algorithms and evolution strategies: Similarities and differences. In Männer, R. and Schwefel, H. P., editors, *Proc. of 1st International Conference on Parallel Problem Solving from Nature, Berlin.* Springer.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor.

Kolmogorov, A. (1965). Three approaches to the quantitative definition of information. *Problems of Information Transmission*, 1:1–11.

Kolmogorov, A. N. (1933). *Grundbegriffe der Wahrscheinlichkeitsrechnung.* Springer, Berlin.

Koza, J. R. (1992). Genetic evolution and co-evolution of computer programs. In Langton, C., Taylor, C., Farmer, J. D., and Rasmussen, S., editors, *Artificial Life II*, pages 313–324. Addison Wesley Publishing Company.

Krogh, A. and Hertz, J. A. (1992). A simple weight decay can improve generalization. In Lippman, D. S., Moody, J. E., and Touretzky, D. S., editors, *Advances in Neural Information Processing Systems 4*, pages 950–957. San Mateo, CA: Morgan Kaufmann.

LeCun, Y. (1985). Une procédure d'apprentissage pour réseau à seuil asymétrique. *Proceedings of Cognitiva 85, Paris*, pages 599–604.

LeCun, Y., Kanter, I., and Solla, S. A. (1991). Second order properties of error surfaces: Learning time and generalization. In Lippman, D. S., Moody, J. E., and Touretzky, D. S., editors, *Advances in Neural Information Processing Systems 3*, pages 918–924. San Mateo, CA: Morgan Kaufmann.

Levin, L. A. (1973a). On the notion of a random sequence. *Soviet Math. Dokl.*, 14(5):1413–1416.

Levin, L. A. (1973b). Universal sequential search problems. *Problems of Information Transmission*, 9(3):265–266.

Levin, L. A. (1974). Laws of information (nongrowth) and aspects of the foundation of probability theory. *Problems of Information Transmission*, 10(3):206–210.

Levin, L. A. (1976). Various measures of complexity for finite objects (axiomatic description). *Soviet Math. Dokl.*, 17(2):522–526.

Levin, L. A. (1984). Randomness conservation inequalities: Information and independence in mathematical theories. *Information and Control*, 61:15–37.

Li, M. and Vitányi, P. M. B. (1989). A theory of learning simple concepts under simple distributions and average case complexity for the universal distribution. In *Proc. 30th American IEEE Symposium on Foundations of Computer Science*, pages 34–39.

Li, M. and Vitányi, P. M. B. (1993). *An Introduction to Kolmogorov Complexity and its Applications.* Springer.

Linsker, R. (1988). Self-organization in a perceptual network. *IEEE Computer*, 21:105–117.

Maass, W. (1994). Perspectives of current research about the complexity of learning on neural nets. In Roychowdhury, V. P., Siu, K. Y., and Orlitsky, A., editors, *Theoretical Advances in Neural Computation and Learning.* Kluwer Academic Publishers.

MacKay, D. J. C. (1992). A practical Bayesian framework for backprop networks. *Neural Computation*, 4:448–472.

Martin-Löf, P. (1966). The definition of random sequences. *Information and Control*, 9:602–619.

Milosavljevič, A. and Jurka, J. (1993). Discovery by minimal length encoding: A case study in molecular evolution. *Machine Learning*, 12:96–87.

Moody, J. E. (1992). The effective number of parameters: An analysis of generalization and regularization in nonlinear learning systems. In Lippman, D. S., Moody, J. E., and Touretzky, D. S., editors, *Advances in Neural Information Processing Systems 4*, pages 847–854. San Mateo, CA: Morgan Kaufmann.

Mozer, M. C. and Smolensky, P. (1989). Skeletonization: A technique for trimming the fat from a network via relevance assessment. In Touretzky, D. S., editor, *Advances in Neural Information Processing Systems 1*, pages 107–115. San Mateo, CA: Morgan Kaufmann.

Nowlan, S. J. and Hinton, G. E. (1992). Simplifying neural networks by soft weight sharing. *Neural Computation*, 4:173–193.

Parker, D. B. (1985). Learning-logic. Technical Report TR-47, Center for Comp. Research in Economics and Management Sci., MIT.

Paul, W. and Solomonoff, R. J. (1991). Autonomous theory building systems. Manuscript, revised 1994.

Pearlmutter, B. A. and Rosenfeld, R. (1991). Chaitin-Kolmogorov complexity and generalization in neural networks. In Lippman, D. S., Moody, J. E., and Touretzky, D. S., editors, *Advances in Neural Information Processing Systems 3*, pages 925–931. San Mateo, CA: Morgan Kaufmann.

Pednault, E. P. D. (1989). Some experiments in applying inductive inference principles to surface reconstruction. In *11th IJCAI*, pages 1603–1609. San Mateo, CA: Morgan Kaufmann.

Quinlan, J. R. and Rivest, R. L. (1989). Inferring decision trees using the minimum description length principle. *Information and Computation*, 80:227–248.

Rechenberg, I. (1971). Evolutionsstrategie - Optimierung technischer Systeme nach Prinzipien der biologischen Evolution. Dissertation. Published 1973 by Fromman-Holzboog.

Rissanen, J. (1978). Modeling by shortest data description. *Automatica*, 14:465–471.

Rissanen, J. (1983). A universal prior for integers and estimation by minimum description length. *The Annals of Statistics*, 11(2):416–431.

Rissanen, J. (1986). Stochastic complexity and modeling. *The Annals of Statistics*, 14(3):1080–1100.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning internal representations by error propagation. In *Parallel Distributed Processing*, volume 1, pages 318–362. MIT Press.

Schaffer, C. (1993). Overfitting avoidance as bias. *Machine Learning*, 10:153–178.

Schmidhuber, J. H. (1987). Evolutionary principles in self-referential learning, or on learning how to learn: The meta-meta-... hook. Report, Institut für Informatik, Technische Universität München.

Schmidhuber, J. H. (1989). A local learning algorithm for dynamic feedforward and recurrent networks. *Connection Science*, 1(4):403–412.

Schmidhuber, J. H. (1992a). Learning complex, extended sequences using the principle of history compression. *Neural Computation*, 4(2):234–242.

Schmidhuber, J. H. (1992b). Learning factorial codes by predictability minimization. *Neural Computation*, 4(6):863–879.

Schmidhuber, J. H. (1993a). On decreasing the ratio between learning complexity and number of time-varying variables in fully recurrent nets. In *Proceedings of the International Conference on Artificial Neural Networks, Amsterdam*, pages 460–463. Springer.

Schmidhuber, J. H. (1993b). A self-referential weight matrix. In *Proceedings of the International Conference on Artificial Neural Networks, Amsterdam*, pages 446–451. Springer.

Schmidhuber, J. H. (1994). Algorithmic art. Technical report, Fakultät für Informatik, Technische Universität München.

Schmidhuber, J. H. and Jankowski, N. (1994). Applications of Levin's optimal universal search algorithm. Technical report, Fakultät für Informatik, Technische Universität München. In preparation.

Schnorr, C. P. (1971). A unified approach to the definition of random sequences. *Mathematical Systems Theory*, 5:246–258.

Schwefel, H. P. (1974). Numerische Optimierung von Computer-Modellen. Dissertation. Published 1977 by Birkhäuser, Basel.

Shannon, C. E. (1948). A mathematical theory of communication (parts I and II). *Bell System Technical Journal*, XXVII:379–423.

Solomonoff, R. (1964). A formal theory of inductive inference. Part I. *Information and Control*, 7:1–22.

Solomonoff, R. (1986). An application of algorithmic probability to problems in artificial intelligence. In Kanal, L. N. and Lemmer, J. F., editors, *Uncertainty in Artificial Intelligence*, pages 473–491. Elsevier Science Publishers.

Solomonoff, R. (1990). A system for incremental learning based on algorithmic probability. In Pednault, E. P. D., editor, *The Theory and Application of Minimal-Length Encoding (Preprint of Symposium papers of AAAI 1990 Spring Symposium)*.

Utgoff, P. (1986). Shift of bias for inductive concept learning. In *Machine Learning*, volume 2. Morgan Kaufmann, Los Altos, CA.

Valiant, L. G. (1987). A theory of the learnable. *Communications of the ACM*, 27:1134–1142.

Vapnik, V. (1992). Principles of risk minimization for learning theory. In Lippman, D. S., Moody, J. E., and Touretzky, D. S., editors, *Advances in Neural Information Processing Systems 4*, pages 831–838. San Mateo, CA: Morgan Kaufmann.

Wallace, C. S. and Boulton, D. M. (1968). An information theoretic measure for classification. *Computer Journal*, 11(2):185–194.

Watanabe, O. (1992). *Kolmogorov complexity and computational complexity.* EATCS Monographs on Theoretical Computer Science, Springer.

Watkins, C. (1989). *Learning from Delayed Rewards.* PhD thesis, King's College.

Weigend, A. S., Huberman, B. A., and Rumelhart, D. E. (1990). Predicting the future: A connectionist approach. *International Journal of Neural Systems*, 1:193–209.

Werbos, P. J. (1974). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.* PhD thesis, Harvard University.

Williams, R. J. (1988). Toward a theory of reinforcement-learning connectionist systems. Technical Report NU-CCS-88-3, College of Comp. Sci., Northeastern University, Boston, MA.

Wolpert, D. H. (1993). On overfitting avoidance as bias. Technical Report SFI TR 93-03-016, Santa Fe Institute, NM 87501.

Zvonkin, A. K. and Levin, L. A. (1970). The complexity of finite objects and the algorithmic concepts of information and randomness. *Russian Math. Surveys*, 25(6):83–124.