Discovering Solutions with Low Kolmogorov Complexity and High Generalization Capability

In A. Prieditis and S. Russell, eds., Machine Learning: Proceedings of the 12th international conference, *488–496, Morgan Kaufmann, San Francisco, CA, 1995.*

Jürgen Schmidhuber
IDSIA, Corso Elvezia 36, 6900-Lugano, Switzerland*
juergen@idsia.ch

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 (→ 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 extension 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). Experiments focus on the task of discovering "algorithmically simple" neural networks with low Kolmogorov complexity and high generalization capability. These experiments demonstrate 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.

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 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 (see Li and Vitanyi (1993) for an excellent overview, see Schmidhuber (1994c) for an application to fine arts).

Purpose of paper. This work and the experiments to be presented herein are intended (1) to demonstrate

*Alternatively: TUM, 80290 München, Germany.

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, (3) to point to problems concerning "incremental learning", and (4) to mention initial steps towards solving them.

Outline. Section 2 briefly reviews basic concepts of algorithmic complexity theory relevant to machine learning, including Levin complexity (an extension of Kolmogorov complexity) and Levin's universal optimal search algorithm. 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 (working) implementation of (a probabilistic variant of) universal search. Experiments in section 4 focus on the task of finding "simple" neural nets with excellent generalization capability. Section 5 goes beyond sections 1-4, by addressing "incremental" learning situations.

2 BASIC CONCEPTS

Each Turing machine (TM) C (mapping bitstrings to bitstrings, without loss of generality) computes a partial function $f_C : \{0,1\}^* \rightarrow \{0,1\}^*$ (f_C is undefined where C does not halt). The **Kolmogorov complexity** $K_U(s)$ of a finite string s is the length of the shortest program p that computes s on a universal Turing machine U and halts, where the set of possible halting programs forms a prefix code (no halting program can be the prefix of another one):

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

where $\mid p \mid$ denotes the length of p. The invariance theorem (Solomonoff, 1964; Kolmogorov, 1965; Chaitin, 1969) states that $K_{U_1}(s) = K_{U_2}(s) + O(1)$ for two universal machines U_1 and U_2 and for all s. Therefore we may drop the index U and write K instead of K_U.

Machine learning 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

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

We would like to select H such that $P(H \mid D)$ is maximal. Taking the log of this equation leads to the principle of minimum description length (Wallace, 1968; Rissanen, 1978). 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, as will be seen next.

Universal prior (Levin, 1974; Solomonoff, 1964; Gacs, 1974; Chaitin, 1975). Define $P_U(s)$, the *a priori probability* of a bitstring s, as the probability of guessing a halting program that computes s on universal TM U. Here, the way of guessing is defined by the following procedure: whenever the scanning head of U's input tape (initially blank) shifts right to a field that has not been scanned before, do: With probability $\frac{1}{2}$ fill it with a 0; with probability $\frac{1}{2}$ fill it with a 1. We obtain

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

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. 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 a priori probabilities to hypotheses (or other computable objects).

Dominance of shortest programs. It can be shown (Levin, 1974; Chaitin, 1975) that

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

The probability of a string is dominated by the probabilities of its shortest programs. This justifies "Occam's razor": in equation (1), the *a priori* most probable H are those with minimal Kolmogorov complexity.

Dominance of universal prior. Suppose there are infinitely many enumerable solution candidates (strings): amazingly, P_U dominates all discrete enumerable semimeasures P (including probability distributions, see e.g. Li and Vitanyi (1993) for details) in the following sense: for each P there is a constant c such that $P_U(s) >= cP(s)$ for all strings s.

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. In what follows, a (not necessarily halting) program is a string on U's input tape which can be scanned completely by U. Let U scan a program q 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.

Levin's universal optimal search algorithm (Levin, 1973; 1984). Suppose we are looking for the solution (a string) to a given problem. Levin's universal search algorithm 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. Amazingly, for a broad class of problems, including inversion 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.

3 PROBABILISTIC SEARCH

Levin's algorithm was considered of interest for theoretical purposes (see e.g. Allender, 1992; and Li and Vitanyi, 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.** See also Heil's more recent diploma thesis at TUM (1995). Solomonoff (1986) himself apparently implemented very restricted versions. In what follows, however, I will focus on the (working) 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.

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. 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 written 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 arguments 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 a very similar alternative implementation, the original universal search algorithm was used to systematically generate all solution candidates in order of their Levin complexities. See also Heil's diploma thesis at TUM (1995)).

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 described 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 *Instruction-Pointer* to $OracleAddress$. This makes reruns easy. 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).

2. Self-sizing programs. 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 (more probable).

3. Probabilistic setting. Why use a probabilistic search algorithm instead of the original universal search procedure? 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 (1994) and Stefan Heil (1995)). With the examples below, however, total 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.

4 "SIMPLE" NEURAL NETS

Previous work on the relationship between neural network complexity and generalization capability, e.g. Baum and Haussler (1989), Hinton (1993), Nowlan (1992), MacKay (1992), Hassibi (1993), Vapnik (1992), Maass (1994), is not general in the sense of Solomonoff, Kolmogorov, and Levin. Likewise, 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. For such reasons, most of the remainder of this paper is devoted to simulations of the more general method based on the uni-

versal prior, self-sizing programs, and the probabilistic search algorithm preferring candidates with low Levin complexity over candidates with high Levin complexity. With certain toy problems, it will be demonstrated that the approach can lead to generalization results unmatchable by more traditional neural net algorithms.

With the following experiments, an average 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.

4.1 TASK 1: 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 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.

Why conventional neural net algorithms fail to solve this problem. Since the training set is very small, 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 gradient descent algorithms. Note, however, that scaling the inputs differently 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.

Probabilistic search. 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*[1]. 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 generating networks fitting the 3 training exemplars were found in 20 out of 100000 runs. *18 of these 20 led to perfect generalization on the* $\binom{100}{3} - 3 = 161,697$ *unseen test examples.* Typical programs used (conditional) jumps to generate correct solutions. See (Schmidhuber, 1994a) for details.

Comment. With the task 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. Nowlan and Hinton encourage groups of weights with

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

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.

4.2 TASK 2: ADDING INPUT POSITIONS

The task. We use the same perceptron-like network and the same input data as above. The goal is different and harder to achieve, 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.

Why conventional neural net algorithms fail to solve this problem: for the same reasons they fail to solve the previous problem (see section 4.1). The training set is too small to obtain reasonable generalization on the test set.

RESULTS. Programs generating networks 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. *8 of the 10 successful runs led to perfect generalization on the 161,697 unseen test examples.* Typical programs used *Allocate*, *Increment*, *WriteWeight*, and (conditional) jumps to generate correct solutions.

Solution example. The first weight vector fitting the training data was found after 6,902,963 runs. 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 $3.92 * 10^{-16}$. What the program does is this (statements are marked with their addresses):

```
(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. See (Schmidhuber, 1994a) for additional details, and for more elegant alternative solutions found by the system.

4.3 WRITES WITH 2 ARGUMENTS

We keep the task from section 4.2. The primitive *"WriteWeight"* is redefined, however, 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*.

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 1. 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.

Numerous additional experiments (including rather complicated maze tasks) were performed by Norbert Jankowski and Stefan Heil at TUM. Some are described e.g. in Heil's diploma thesis (1995).

General remarks. 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 our search 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.

5 INCREMENTAL LEARNING

With many typical "incremental" learning situations in the real world, there is more informative feedback than with the tasks above, where there is none. The original universal search procedure as formulated by Levin is not designed for optimal use of error feedback in "incremental" learning. However, there appears to be more than one reasonable way of appropriately extending universal search. Some ideas are given in Solomonoff's (1986) and Paul's (1991) more recent theoretical work. Others are presented in (Schmidhuber, 1994a), where *mutations* of previously useful programs are listed in order of their Levin complexities, until additional improvements are found. (Schmidhuber, 1994a) also presents the first experimental results. They show that "incremental" extensions can allow for much faster learning but tend to find less elegant programs.

Ongoing research. Very recently, and for the first time, incremental learning in general environments was put on a basis that appears theoretically sound: Schmidhuber (1994b) goes beyond the current paper, by presenting a novel machine learning paradigm called the *"incremental self-improvement paradigm"*. In principle, a probabilistic system based on this paradigm is able to use previous experience to improve itself, and to improve the way it improves itself, etc. Essentially, the system uses previous experience to learn to modify context-dependent primitive probabilities in a way that leads to more success per time interval, thus learning to make better and better use of its computational resources. The basic ideas are briefly described in the following, concluding subsection.

5.1 THE INCREMENTAL SELF-IMPROVEMENT PARADIGM

This novel machine learning paradigm (Schmidhuber, 1994b) goes beyond the non-incremental approach presented above. To maximize cumulative payoff (reinforcement, reward) to be obtained throughout its entire span of life in a given environment, a system based on incremental self-improvement continually attempts to compute action subsequences leading to faster and faster payoff intake. Since the system is designed such that action subsequences may in fact represent the execution of arbitrary algorithms (including learning algorithms), the approach is very general (but theoretically sound). Its central novel aspects are as follows (Schmidhuber (1994b) exemplifies them by describing a concrete implementation):

1. Computing self-modifications. Like in sections 1-4, the initially highly random actions of the system actually are primitive instructions of a Turing machine equivalent programming language, which allows for implementing arbitrary (learning) algorithms. Action subsequences represent either (1) "normal" interactions with the environment, or (2) "self-modification sequences". Self-modification sequences can compute arbitrary modifications of probabilities of future action subsequences, including future self-modification

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 1: *Used storage after execution of a successful program for the adding perceptron, using a "WriteWeight" primitive with two arguments.*

sequences: the learning system is able to modify itself in a universal way. There is no explicit difference between "learning", "meta-learning", and other kinds of information processing.

2. Life is one-way. Each action of the learning system (including probability modifying actions executed by self-modification sequences) is viewed as a singular event in the history of system life. Unrealistic concepts such as "exactly repeatable training iterations", "boundaries between trials", "epochs", etc. are thrown overboard. In general, the environment cannot be reset. Life is one-way. There is only *one* lifelong training episode. Learning is inductive inference from non-repeatable experiences.

3. Evaluations of self-modification sequences. The system has a time-varying utility value, which is the average payoff per time since system start-up. Each completed self-modification sequence also has a time-varying utility value. This value is the average amount of payoff per time measured since the sequence began execution. *Unlike with previous systems, evaluations of utility take into account all the computation time required for learning, including the time required for evaluating utility.*

4. Recursive definition of "useful" self-modification sequences. The system keeps track of probability modifications computed by completed self-modification sequences that it considers *useful*. *Usefulness* is defined recursively. If there are no previous *useful* self-modification sequences (e.g. at system start-up), a completed self-modification sequence is considered *useful* only for as long as its utility value exceeds the system's utility value. More recent completed self-modification sequences are considered *useful* for as long as they have higher utility values than all preceding self-modification sequences currently considered *useful*.

Essentially, the system only keeps modifications to its probability values that originated from *useful* self-modification sequences.

5. Acceleration of payoff intake / Theoretical soundness. It can be shown that over time, the system tends to make better and better use of its computational resources. In fact, it can be shown that it accelerates payoff (reinforcement) intake in the long run in the following sense: at every time step in the life of the system (except during execution of self-modification sequences), all (self-computed) valid modifications to its strategy have been followed by faster average payoff intake than all previous valid modifications (and system start-up itself). Perhaps somewhat surprisingly, the nature of the environment does not matter (for instance, the interface to the possibly non-deterministic environment does not have to be Markovian).

Unlike the non-incremental system described in sections 2-4 of this paper, incremental self-improvement appears to be a promising way of dealing with lifelong incremental learning. A system based on incremental self-improvement has already been implemented and tested on simple toy tasks (Schmidhuber, 1994b). As expected, the experimental results are consistent with the theoretical predictions.

6 ACKNOWLEDGEMENTS

Thanks to Ray Solomonoff, Peter Dayan, Martin Eldracher, Sepp Hochreiter, Margit Kinder, Daniel Prelinger, Mark Ring, Jan Storck, Erik Winfree, and Gerhard Weiß, for valuable comments on (Schmidhuber, 1994a).

References

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

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

[3] C. H. Bennett. 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, 1988.

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

[5] G.J. Chaitin. A theory of program size formally identical to information theory. *Journal of the ACM*, 22:329–340, 1975.

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

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

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

[9] S. Heil. Universelle Suche und inkrementelles Lernen, diploma thesis, 1995. Fakultät für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München.

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

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

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

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

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

[15] M. Li and P. M. B. Vitányi. *An Introduction to Kolmogorov Complexity and its Applications*. Springer, 1993.

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

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

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

[19] W. Paul and R. J. Solomonoff. Autonomous theory building systems, 1991. Manuscript, revised 1994.

[20] J. Rissanen. Modeling by shortest data description. *Automatica*, 14:465–471, 1978.

[21] J. Schmidhuber. Discovering problem solutions with low Kolmogorov complexity and high generalization capability. Technical Report FKI-194-94, Fakultät für Informatik, Technische Universität München, 1994. Short version in A. Prieditis and S. Russell, eds., Machine Learning: Proceedings of the Twelfth International Conference, Morgan Kaufmann Publishers, pages 488–496, San Francisco, CA, 1995.

[22] J. Schmidhuber. Low-complexity art. Technical Report FKI-197-94, Fakultät für Informatik, Technische Universität München, 1994.

[23] J. Schmidhuber. On learning how to learn learning strategies. Technical Report FKI-198-94, Fakultät für Informatik, Technische Universität München, November 1994. Revised January 1995.

[24] R.J. Solomonoff. A formal theory of inductive inference. Part I. *Information and Control*, 7:1–22, 1964.

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

[26] P. Utgoff. Shift of bias for inductive concept learning. In R. Michalski, J. Carbonell, and T. Mitchell, editors, *Machine Learning*, volume 2, pages 163–190. Morgan Kaufmann, Los Altos, CA, 1986.

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

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