Towards Heuristic Algorithmic Memory

Eray Özkural

Bilkent University Computer Engineering Department
Ankara, Turkey

Abstract. We propose a long-term memory design for artificial general intelligence based on Solomonoff's incremental machine learning methods. We introduce four synergistic update algorithms that use a Stochastic Context-Free Grammar as a guiding probability distribution of programs. The update algorithms accomplish adjusting production probabilities, re-using previous solutions, learning programming idioms and discovery of frequent subprograms. A controlled experiment with a long training sequence shows that our incremental learning approach is effective.

1 Introduction

Teramachine is a universal induction system that features integrated *long-term* memory, as a candidate for Solomonoff's "Phase 1 machine" that he proposed to use as the basis of a powerful AGI system called Alpha [1]. We propose an automatic memory which is recalled appropriately during induction. After each induction problem, the solution is stored in the memory, which is a realization of Solomonoff's idea of guiding probability density function (pdf) of programs. The present system may be viewed as an advanced version of OOPS [2]. We update the guiding pdf after each induction problem so that the *heuristic* solutions that we invent are stored as *algorithmic* information in our *memory* system. Hence, our memory design is called Heuristic Algorithmic Memory (HAM).

If an induction system's probability distribution of programs is fixed, then the system does not have any real long-term learning ability. We can solve this problem by changing the probability distribution so that we extrapolate from the already invented solution programs, allowing more difficult problems to be solved [3]. Modifying the probability distribution essentially defines an *implicit program code.* Thus, after each solution we are implicitly modifying the reference machine. Relative to the implicit universal code, Levin search [4] still has an optimal order of complexity and is effective for approximating Solomonoff induction [5]. The extraction of algorithmic information from solutions affords an effective kind of time-space tradeoff, which works extremely favorably in terms of additional space requirement. The successful extraction of each single bit of mutual algorithmic information among two problems may potentially result in a speed-up of two for the latter problem. However, re-using algorithmic information from previous solutions entails a coding cost which manifests itself as a time penalty during program search (Levin search in our work).

The reader is referred to [2,1,6] for a background on incremental machine learning. A longer version of this paper is available on the aRxiV [7], and a previous version explains the R5RS Scheme grammar which we use [8].

2 Stochastic Context-Free Grammar Updates

A Stochastic Context-Free Grammar (SCFG) is a Context-Free Grammar augmented by a probability value on each production. For each head non-terminal, the probabilities of its productions must sum to one. We can extend Levin Search procedure to work with a SCFG that assigns probabilities to each sentence in the language. For this, we need two things, first a generation logic for individual sentences, and second a search strategy to enumerate the sentences that meet the termination condition of LSearch [2]. In the present system, we use leftmost derivation to generate a sentence, intermediate steps are thus left-sentential forms [9, Chapter 5]. The calculation of the a priori probability of a sentence depends on the fact that in a derivation $S \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow ... \Rightarrow \alpha_n$ where productions $p_1, p_2, ..., p_n$ have been applied in order to start symbol S, the probability of the sentence α_n is $P(\alpha_n) = \prod_{1 \leq i \leq n} p_i$. Note that the productions in a derivation are conditionally independent. While this makes it much easier for us to calculate probabilities of sentential forms, it limits the expressive power of pdf. Note that search algorithm details are beyond the scope of this paper.

The most critical part of our design is updating the SCFG so that the discovered solutions in a training sequence will be more probable in subsequent searches. We propose four synergistic update algorithms for HAM. Our SCFG structure extends the usual productions with production procedures, which dynamically generate productions.

2.1 Modifying Production Probabilities

The simplest kind of update is modifying the probabilities as new solutions are added to the solution corpus. For this, however, the search algorithm must supply the derivation that led to the solution (which we do), or the solution must be parsed using the same grammar. Then, the probability for each production $A \rightarrow \beta$ in the solution corpus can be easily calculated by the ratio of frequency of productions $A \rightarrow \beta$ in the solution corpus to the frequency of productions in the corpus with a head of A. The production procedures are excluded from this update as they can be variant. However, we cannot simply write the probabilities calculated this way over the initial probabilities, as initially there will be few solutions, and most probabilities will be zero. We use exponential smoothing to solve this problem:

$$s_0 = p_0$$
$$s_t = \alpha p_t + (1 - \alpha) s_{t-1}$$

where p_0 is the initial probability, p_t is the probability in the corpus for problem t, s_t the smoothed probability for problem t, and α is the smoothing factor. We used a smoothing factor of 0.125. See [10] for the application of smoothing in a similar problem. Other methods like Laplace's rule may be used instead [1].

2.2 Re-using Previous Solutions

In the course of a training sequence, the solutions can be incorporated in full by adding the solutions to the grammar. In the case of Scheme, there could be many possible implementations. The simplest design is to add all the solutions to the library of the Scheme interpreter, add a hook non-terminal previous-solution to the grammar, and then extend the previous-solution with the syntax to call the new solution. We assume that this syntax is provided in the problem definition. The new solution among other previous solutions is given a probability of γ in the hope that this solution will be re-used soon, and then the probabilities of the old productions of previous-solution are normalized so that they sum to $1 - \gamma$. We currently use a γ of 0.5. If it is difficult to add the solutions to the Scheme interpreter as in our case, then all the solutions can be added as `define` blocks in the beginning of the program produced, which requires avoiding redundant definitions [7].

2.3 Learning Programming Idioms

Programmers do not only learn of concrete solutions to problems, but they also learn abstract programs, or program schemas. One way to formalize this is that they learn sentential forms. If we can extract appropriate sentential forms, we can add these to the grammar, as well. We construct the derivation tree from the leftmost derivation, with an obvious algorithm that we will omit. The current abstraction algorithm starts with the derivation sub-trees rooted at each expression in the current solution. For each derivation sub-tree, we prune the leaves from the bottom-up. At each pruning step, an abstract expression is output. The pruning is iterated until a few symbols remain. Every abstract expression thus found is added to a new non-terminal that contains the abstract expressions of the current solution with equal probability. The new non-terminal is added to the top-level non-terminal abstract-expression with 0.5 probability, which is itself one of the productions for expression. These productions may later be modified and used by update algorithms one and two. Note that the orthogonality of the language helps us in integrating programming idioms into HAM. Thus, several sentential forms are learnt from a single solution in this fashion corresponding to different syntactic abstractions. We anticipate that the system will eventually learn complex programming idioms like recursion patterns and data constructors.

2.4 Frequent Sub-program Mining

Mining the solution corpus further enhances the guiding probability distribution. Frequent sub-programs in the solution corpus, i.e., sub-programs that occur with a frequency above a given support threshold, can be added again as alternative productions to the commonly occurring non-terminal expression in the Scheme grammar. For instance, if the solution corpus contains several `(lambda (x y) (* x y) )` subprograms, the frequent sub-program mining would discover that and we can add it as an alternative expression to the Scheme grammar.

We would like to find all frequent subprograms that occur twice or more so that we can increase the probability of such sub-programs accordingly. We first interpret the problem of finding frequent sub-programs as a syntactic problem, disregarding semantic equivalences between sub-programs. Once formulated in our program representations of derivation trees as labelled rooted frequent sub-tree mining, the frequent sub-program mining algorithm is a reasonable extension of traditional frequent pattern mining algorithms. We have implemented a BFS patterned fast mining algorithm by exploiting the property that every sub-tree of a frequent tree is frequent (see [11] for an advanced algorithm). We find frequent sub-trees (with a support threshold of 2 currently) of all sub-trees of derivation trees rooted at expression in the solution corpus. At each update, a non-terminal hook frequent-expression in the grammar is rewritten by assigning probabilities according to the frequency of each frequent sub-program. Note that most frequent expressions are abstract (i.e., sentential forms).

3 Experiments

Our experimental tests were carried out at TUBITAK ULAKBIM High Performance Computing Center on 144 AMD Opteron cores. We know of *no* previous demonstration of realistic experiments over a long training sequence for general purpose machine learning. Solomonoff had stated: "It cannot be emphasized too strongly, that the goal of early training sequence design, is not to solve hard problems, but to get problem solving information into the machine. Since Lsearch is easily adapted to parallel search, there is a tendency to try to solve fairly difficult problems on inadequately trained machines. The success of such efforts is more a tribute to progress in hardware design then to our understanding and exploiting machine learning." [12, Section 6]. We can show the effectiveness of our memory system leaving no place for doubt through *controlled experiments*. We run the entire training sequence with updates turned off and on. If the update algorithms cause a significant speed-up over search with no update, we can conclude that the update algorithms are effective. We use Conceptual Jump Size (CJS) to calculate the difficulty of a problem. CJS $= t_i/p_i$ where t_i is the running time of solution program and p_i is its a priori probability. The upper bound of Levin Search's running time is 2.CJS [12, Appendix A]. Our experiments are preferred to calculating CJS's by hand, as in these experiments we are using Scheme R5RS in its full glory. Note that we are interested in *only* detecting whether any information transfer occurs across problems rather than trying to solve difficult problems with a machine that has *no long-term memory.* The running time of a trial program is measured in Scheme execution cycles, which is the number of primitive Scheme operations (e.g., CAR) that are evaluated.

We have developed a training sequence composed of operator induction problems. For each problem, we have a set of input and output pairs, and we approximate operator induction [1,13]. Training sequence 1 contains, in order, the square function `sqr`, the addition of two variables `add`, a function to test if the argument is zero `is0`, all of which have 3 example pairs, fourth power of a number `pow4` with just 2 example pairs, boolean `nand`, and `xor` functions with 4 example

Table 1. Performance of training sequence 1 with no update, $|HAM| = 17145$

Problem	Time	Trials	Errors	Cycles	Max Cyc.	p_i	t_i	CJS	$H(s_i)$
sqr	16.28	5.34×10^5	1.57×10^5	5.46×10^6	2.05×10^8	2.19×10^{-7}	37	1.68×10^8	22.12
add	19.9759	1.03×10^6	3.13×10^5	1.13×10^7	4.1×10^8	9.77×10^{-8}	40	4.09×10^8	23.28
is0	7.57	41210	9531	430336	1.10×10^7	3.95×10^{-6}	34	8.59×10^6	17.94
pow4	1759.45	3.34×10^8	1.38×10^8	3.24×10^9	2.55×10^{11}	1.67×10^{-10}	26	1.55×10^{11}	32.47
nand	3497.17	6.48×10^8	2.71×10^8	6.69×10^9	5.13×10^{11}	2.01×10^{-10}	56	2.78×10^{11}	32.21
xor	1848.8	3.38×10^8	1.3×10^8	3.54×10^9	2.53×10^{11}	2.01×10^{-10}	52	2.58×10^{11}	32.21
all	7150.06								

Table 2. Performance of training sequence 1 with update

Problem	Time	Trials	Errors	Cycles	Max Cyc.	p_i	t_i	CJS	$H(s_i)$	$\|HAM\|$
sqr	11.4	6.34×10^5	1.81×10^5	6.64×10^6	2.35×10^8	2.19×10^{-7}	37	1.68×10^8	22.12	17318
add	7.63	2.46×10^5	8.52×10^4	3.39×10^6	8.19×10^7	0.33×10^{-6}	40	1.19×10^8	21.5	17515
is0	2.72	10202	2969	136363	2.14×10^6	0.13×10^{-4}	34	2.60×10^6	16.22	17566
pow4	6.45	2.62×10^5	8.92×10^4	3.6×10^6	9.86×10^7	0.72×10^{-6}	54	7.39×10^7	20.38	17617
nand	209.53	2.55×10^7	1.12×10^7	3.72×10^8	1.51×10^{10}	0.50×10^{-8}	56	1.11×10^{10}	27.57	17962
xor	4.22	43749	14216	667625	1.18×10^7	0.47×10^{-5}	57	1.19×10^7	17.68	18438
all	245.1									

pairs each. Tables 1 and 2 convey the performance of our system on training sequence 1 without update and with update, respectively.

For each problem, we give the time in seconds, number of trials, number of Scheme errors, number of Scheme execution cycles spent, number of maximum Scheme cycles allocated to search, a priori probability of solution (p_i), running time of solution in Scheme cycles (t_i), Conceptual Jump Size, the length of the implicit program code of the solution ($H(s_i) = -lg(p_i)$) and the size of HAM in bytes after the update, respectively. Total time for the training sequence is also given. The initial time limit is 10^6 cycles.

The overall speed-up of training sequence 1 with updates is 29.17 compared to the tests with no HAM update. This result indicates a consistent success of transfer learning in a long training sequence. The search time for the solutions in Table 2 tend to decrease compared to Table 1. The memory size has increased only 1293 bytes, for storing information for 6 operator induction problems, which corresponds to %7.5 increase in memory for 29.17 speed-up, which is a very favorable time-space trade-off. The solution of logical functions took longer than previous problems in Table 1, but we saw significant time savings in Table 2. Previous solutions are re-used aggressively. In Table 2, `pow4` solution `(define (pow4 x ) (define (sqr x ) (* x x)) (sqr (sqr x ) ))` re-uses the `sqr` solution and takes only 2.62×10^6 trials, its CJS speeds up 2097.4 times over the case with no update, and the search achieves 272 speed-up in running time.

4 Conclusion and Future Work

We have proposed four update algorithms for incremental machine learning. The effectiveness of our update logic has been demonstrated with experiments in one *long* training sequence, a feat that has not been accomplished before to the best of our knowledge. In the future, we plan to implement Q/A induction and the Phase 2 of Solomonoff's Alpha system [1].

References

1. Solomonoff, R.J.: Progress in incremental machine learning. NIPS Workshop on Universal Learning Algorithms and Optimal Search (2002)
2. Schmidhuber, J.: Optimal ordered problem solver. Machine Learning 54, 211–256 (2004)
3. Solomonoff, R.J.: A system for incremental learning based on algorithmic probability. In: Proceedings of the Sixth Israeli Conference on Artificial Intelligence, Tel Aviv, Israel, pp. 515–527 (1989)
4. Levin, L.: Universal problems of full search. Problems of Information Transmission 9(3), 256–266 (1973)
5. Solomonoff, R.J.: Optimum sequential search. Technical report, Oxbridge Research (1984)
6. Solomonoff, R.J.: Algorithmic probability: Theory and applications. In: Dehmer, M., Emmert-Streib, F. (eds.) Information Theory and Statistical Learning, pp. 1–23. Springer Science+Business Media, N.Y (2009)
7. Özkural, E.: Teraflop-scale incremental machine learning. CoRR abs/1103.1003 (2011), `http://arxiv.org/abs/1103.1003`
8. Özkural, E.: Gigamachine: incremental machine learning on desktop computers. Draft (2009), `http://examachine.net/papers/gigamachine-draft.pdf`
9. Hopcroft, J.E., Rajeev Motwani, J.U.: Introduction to Automata Theory, Languages, and Computation, 2nd edn. Addison Wesley, Reading (2001)
10. Merialdo, B.: Tagging english text with a probabilistic model. Computational Linguistics 20, 155–171 (1993)
11. Zaki, M.J.: Efficiently mining frequent trees in a forest. In: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining. KDD 2002, pp. 71–80. ACM Press, New York (2002)
12. Solomonoff, R.J.: Algorithmic probability, heuristic programming and agi. In: Third Conference on Artificial General Intelligence, pp. 251–157 (2010)
13. Solomonoff, R.J.: Three kinds of probabilistic induction: Universal distributions and convergence theorems. The Computer Journal 51(5), 566–570 (2008)