An Application of Stochastic Context Sensitive Grammar Induction to Transfer Learning

Eray Özkural

Gök Us Sibernetik Araştırma ve Geliştirme Ltd. Şti.

Abstract. We generalize Solomonoff's stochastic context-free grammar induction method to context-sensitive grammars, and apply it to transfer learning problem by means of an efficient update algorithm. The stochastic grammar serves as a guiding program distribution which improves future probabilistic induction approximations by learning about the training sequence of problems. Stochastic grammar is updated via extrapolating from the initial grammar and the solution corpus. We introduce a data structure to represent derivations and introduce efficient algorithms to compute an updated grammar which modify production probabilities and add new productions that represent past solutions.

1 Introduction

The present paper addresses the problem of stochastic context-sensitive grammar (SCSG) guided universal induction. We update the guiding stochastic grammar, i.e., guiding probability distribution (GPD), such that the information in the current solution and past solutions may be transferred to future solutions in an efficient way, and the running time of future solutions may be decreased by suitably extrapolating the GPD from the initial GPD and the solution corpus. If an induction system's probability distribution of programs is fixed, then the system does not have any real long-term learning ability that it can exploit during induction. We can alleviate this problem by changing the probability distribution so that we extrapolate from the already invented solution programs, allowing more difficult problems to be solved later. We can modify the GPD and the reference machine to encode useful algorithmic information from past solutions. GPD may be improved so that it makes relevant programs more likely, and the reference machine may be augmented with new subprograms.

We assume that we have a good approximation algorithm to solve stochastic operator induction, set induction and sequence prediction problems. It has been explained in references how Levin Search may be used for this purpose, however, any other appropriate search method that can search the universal set of programs/models, or a large model class, including genetic programming is admissible. The only significant condition we require is that the approximation algorithm uses the GPD specified by the stochastic grammar as the a priori distribution of programs, regardless of the search method. The approximation algorithm thus required must return a number of programs/stochastic models that have high a priori probability (with respect to GPD) and fit the data well. For instance, a satisfactory solution of the (universal) operator induction problem will return a number of programs that specify a conditional probability distribution function

of output data given input data (more formally, an operator $O^j(A|Q)$ that assigns probabilities to an answer A, given question Q). Thus, a set of operator O^J's are assumed to be returned from an operator induction solver.

2 Background

Universal inductive inference theory was proposed by Ray Solomonoff in 1960's [10, 11], and its theoretical properties were proven in [12]. Levin search is a universal problem solution method which searches all possible solution programs according to an order of induction, and it is bias-optimal [3]. The first incremental general purpose incremental machine learning system was described in [13]. Later work of Solomonoff proposed an improved general purpose incremental machine learning system, including the abstract design of a powerful artificial intelligence system that can solve arbitrary time-limited optimization problems, and also explains in detail how Levin search may be used to solve universal induction problems [14]. Solomonoff later proposed the guiding probability distribution (GPD) update problem, and recommended PPM, genetic programming, echo state machines, and support vector regression as potential solution methods [17].

In a theoretical paper, Solomonoff described three kinds of universal induction: sequence induction, set induction and operator induction, defining the solutions as optimization problems [15]. Sequence induction model predicts the next bit for any bit string. Set induction allows us to predict which bit string would be added to a set of bitstrings, modeling clustering type of problems. Operator induction learns the conditional probability between question and answer pairs written as bit strings, allowing us to predict the answer to any unseen question, solving in theory any classification/regression type of problem. A practical method for context-free grammar discovery as relevant to the present paper was first proposed in [16], including an a priori distribution for context-free grammars, which we shall refer to later. Schmidhuber proposed an adaptive Levin search method called Optimal Ordered Problem Solver (OOPS) that changes the probability distribution dynamically, with a simple probability model of programs that is suitable for low-level machine languages, by assigning an instruction probability to each program, and using an instruction that bumps the probability of an instruction [9]. This work is significant in that it shows that Levin search can be used to solve conceptually difficult problems in practice, and that adaptive Levin search is a promising strategy for incremental learning. Our preceding study adapted the practical approach of OOPS to stochastic context-free grammars instead of instruction probabilities for deterministic function induction and function inversion problems using Scheme as the reference machine [8], which also provides preliminary experimental support but otherwise does not address stochastic problems.

Solomonoff's seminal contribution was the universal distribution. Let M be a universal computer. A priori probability of a program π is $P(\pi) = 2^{-|\pi|}$ for prefix-free M's where $|\pi|$ denotes length of binary program π. He defined the probability that a string $x \in \{0,1\}^*$ is generated by a random program as:

$$P_M(x) = \sum_{M(\pi)=x*} 2^{-|\pi|} \tag{1}$$

where $P_M(x)$ is the algorithmic probability of x, $x*$ is any continuation of x. This particular definition was necessary so that we could account for programs (including non-terminating programs) that generate a sequence of bits, to be used in sequence prediction. We shall denote it by just $P(\cdot)$ in the rest of the paper, as we can discern probability of programs from bit strings. P_M is also called the universal prior for it may be used as the prior in Bayesian inference. Note that P is a semi-measure, but it may be suitably normalized for prediction applications.

3 Guiding Probability Distribution

An induction program uses a reference machine M, and an a priori probability distribution of programs in M, $P(\cdot)$, to achieve induction. The a priori probability distribution of programs is encoded as a stochastic context-sensitive grammar of programs in the present paper, and corresponds to the GPD.

In our present system, we have a fast memory update algorithm comprised of three steps that modify a Stochastic Context-Sensitive Grammar (SCSG) of programs, which is the explicit long-term memory representation of our memory system. A SCSG may be formally defined as a tuple $G = (N, \Sigma, P, S, Pr)$ where N is the set of non-terminals, Σ is the set of terminals, P is the set of productions of the form $\alpha A \beta \rightarrow \alpha \gamma \beta$ where a non-terminal A in the context of string α to the left and β to the right expands to string γ in the same context. S is the start symbol, and $Pr(p_i)$ assigns a probability to each production p_i. The calculation of the a priori probability of a sentence depends on the obvious 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 naturally $P(\alpha_n) = \prod_{1 \leq i \leq n} Pr(p_i)$, and the probabilities assigned to each sentence must conform to probability axioms. In context-sensitive grammars, the consistency axiom is achieved by ensuring that the sum of probabilities of each production which has a left hand side of $\alpha A \beta$ adds up to 1 for each A, and any α and β. It is absolutely important that the probability distribution specified by the guiding distribution is consistent. Otherwise, some universal induction algorithms may go into infinite loops. This is the reason why we limited the grammar to context-sensitive, because we cannot yet handle unrestricted grammars. Thus, universal induction approximation is also required to detect consistency errors during solution, and abort when necessary.

4 Transfer Learning

The transfer learning system works on multiple induction problems. The induction problem may be any induction problem such as sequence prediction, set induction, operator induction [15] or any other reasonable extension of universal induction such as Hutter's extension to sequence prediction [2], program learning [4] and grammar driven genetic programming in general [6, 5] . The induction problem is solved (i.e., approximated, since it is semi-computable), using genetic programming or Levin search. Alternatively, any appropriate search method can be used. At this stage, we run the fast memory update algorithm on the latest solution, which we call solution N. We iterate until no more problems remain.

The transfer learning program contains a reference machine M, which is in our system usually a universal computer that can specify any stochastic model, and it is an essential input to the universal induction approximation. However, M can also be any more restricted model class so as to promote efficiency. For instance, it can be constrained to only primitive recursive functions, or other model classes in statistics such as Markov models (forgoing universality). The universal induction approximation is any valid approximation to the kinds of induction problems supported by the system, usually just Levin search using reference machine M as the reference machine and SCSG of programs as the a priori program distribution. However, it is known that genetic programming can solve the induction problem just as well. We assume that at least sequence prediction, set induction and operator induction problems can be solved by giving appropriate parameters to universal induction approximation. It is known that the most difficult problem among them, operator induction, can be solved by finding in available time a set of operator models $O^j(\cdot|\cdot)$ such that $\sum_j a_n^j$ is as large as possible where

$$a_n^j = P(O^j(\cdot|\cdot)) \prod_{i=1}^{n} O^j(A_i|Q_i). \tag{2}$$

That is, we use a universal prior to determine the operators that both have high a priori probability ($P(O^j())$, and fit the data well ($\prod O^j()$), i.e. they have high goodness of fit. $P(\cdot)$ may be calculated using SCSG of programs. The solution corpus contains the stochastic models inferred by the induction algorithm for each solution, and associated information such as formal parameters of each model, and their derivation in SCSG of programs. The fast memory update algorithm , uses the solution corpus to improve SCSG of programs, so that the a priori probability of models/programs in the solution corpus increases, while the grammar does not grow prohibitively, as will be explained in detail. The fast memory update algorithm is the most important part of the system, as it is the critical step for extrapolating from solution corpus a better grammar that accelerates future solutions. We call the algorithm "fast" in the sense that it does not have an exponential running time complexity with respect to output size as in Levin search; we merely use ordinary enumeration and data mining algorithms that are reasonably fast. Please also see [1] for an algorithm that applies SCSG's to trajectory learning, which our work differs chiefly in that we use grammar induction to learn program distributions.

5 Formalization

Some formalization may make the process clearer. Let G_0 be the initial SCSG that acts as a probability distribution (GPD) for all programs in the reference machine M . Let the training sequence be a sequence $D = [d_1, d_2, ..., d_N]$ of N induction problems where each d_i is the data for either of sequence prediction, set induction or operator induction problems. The type of induction may be assumed to be represented with a $[t_i]$ sequence. Alternatively, the induction types may contain another extension of universal induction that is supported by the induction algorithm used. Then, we assume that the universal induction approximation finds a set of probability models that have high a priori probability and explain the data well. Each such model x may be run on the

reference machine M with additional arguments that correspond to the induction problem type $M(x, arg_1, arg_2, ...)$. For instance, if we are inferring the next element of a sequence, the probability model could take a list $B = b_1 b_2 ... b_n$, and the next element observed: $M(s, b_1 b_2 b_3 ... b_n, b_{n+1}) = P(b_{n+1} | b_1 b_2 b_3 ... b_n)$, the formal parameters for the problem being worked on is thus easily determined according to problem type. The solution of the induction problem i yields m probability models, which we will store in the solution corpus. Sequence $S = [s_i]$ where solution i (s_i) represents the m solutions as a set $\{s_{i,j}\}$ of individual solution programs. Additional useful information may be associated with each solution program, as will be explained later. Thereafter, the update algorithm works, and using both the entire solution corpus S and the previous grammars G_0 through G_{N-1} , it produces a new grammar G_N. In practice, it is possible to use only G_0 and G_{N-1} although according to some implementation choices made, other grammars may have to be kept in memory, as well.

Following are the steps of the fast memory update algorithm, respectively. The step to update production probabilities, updates production probabilities in SCSG of programs based on the initial grammar G_0 and the solution corpus, calculating the production probabilities in the program derivations among the solution corpus and then extrapolating from them and the initial probabilities in G_0. The memoization step adds a production for each model in solution N to generate it by a call to a subprogram $s_{i,j}$, which has already been added to reference machine M . The derivation compression step finds regularities in the derivations of each program in the solution corpus and adds shortcut productions to SCSG of programs suitably to compress these regularities in future solutions. To be able to make this compression, it is essential that the derivations are represented using a derivation lattice, as will be explained next.

6 Derivation Lattice

A derivation lattice of a derivation of a SCSG G is a bipartite directed graph $L(G) = (V, E)$ with two disjoint vertex sets V_s (set of symbols) and V_p (set of productions), such that $V_s \subseteq N \cup \Sigma$ (all symbols in the grammar) and $V_p \subseteq P(G)$ (all productions). Each production $\alpha A \beta \rightarrow \alpha \gamma \beta$ in the derivation is represented by a corresponding vertex p_i in the lattice, and an incoming edge is present for each symbol in $\alpha A \beta$ and an outgoing edge is present for each symbol in γ. Furthermore, each incoming and outgoing edge is given a consecutive integer label, starting from 1, indicating the order of the symbol in production left-hand and right-hand side, respectively. The production vertices are labeled with the production number in the SCSG and their probabilities in SCSG. Additionally, if there are multiple input symbols in the derivation, their sequence order must be given by symbol vertex labels, and the set of all input symbols $I(L(G))$ must be determined. The traversal of the leaf nodes according to increasing edge labels in the lattice will give the output string of the lattice and is denoted as $O(L(G))$. (G is dropped from notation where it is clear.) The level of a vertex u in the lattice is analogous to any directed acyclic graph, it may be considered as the minimum graph distance from the root vertices $I(L)$ of the lattice to u. This way, the parse trees used in lower-order grammars are elegantly generalized to account for context-sensitive grammars, and all the necessary information is self-contained in a data structure. Alternatively, a

hypergraph representation may be used in similar manner to achieve the same effect, since hypergraphs are topologically equivalent to bipartite graphs. We denote the set of derivation lattices of all solution programs $s_{i,j} \in S$ by $L(S)$. Please see the Appendix for an example [7].

7 Update Algorithm

The solution corpus requires a wealth of information for the solution to each induction problem in the past. Recall that during solution, each induction problem yields a number of programs. For solution i, the programs found are stored. The derivation lattice for each program and the a priori probability of each program, both according to SCSG of programs are stored, as well as the formal parameters of the program so that they may be called conveniently in the future. Additional information may be stored in the solution corpus, such as the time it took to compute the solution, and other relevant output which may be obtained from universal induction approximation.

The SCSG of programs stores the current grammar G and it also maintains the initial grammar G_0, which is used to make sure that the universality of the grammar is always preserved, and required for some of the extrapolation methods. G_0 is likely constructed by the programmer so that a consistent, and sensible probability distribution is present even for the first induction problem. The probabilities cannot always be given uniformly, since doing so may invalidate consistency. Additionally, the entire history of grammars may be stored, preferably on secondary storage.

7.1 Updating Production Probabilities

The step to update production probabilities works by updating the probabilities in SCSG of programs as new solutions are added to the solution corpus. For this, however, the search algorithm must supply the derivation lattice that led to the solution, or the solution must be parsed using the same grammar. Then, the probability for each production $p_i = \alpha A \beta \rightarrow \alpha \gamma \beta$ in the solution corpus can be easily calculated by the ratio of the number (n_1) of productions with the form $\alpha A \beta \rightarrow \alpha \gamma \beta$ in the derivations of the solutions in solution corpus to the number (n_2) of all productions in the corpus that match the form $xAy \rightarrow xzy$, for any x, y, z, that is to say, any production that expands A given arbitrary context and right-hand side. We cannot replace the probabilities calculated this way (Laplace's rule) over the initial probabilities in G_0, as initially there will be few solutions, and most probabilities n_1/n_2 for a production will be zero, making some programs impossible to generate. We can use the following solution. It is likely that the initial distribution G_0 will have been prepared by the programmer through testing it with solution trials. The number of initial trials thus considered is estimated, for instance, 10 could be a good value, let this number be n_3. Then, instead of n_1/n_2 we can use $(n_1 + p_0 * n_3)/(n_2 + n_3)$ where p_0 is the probability of p_i in G_0. Alternatively, we can use various smoothing methods to solve this problem, for instance exponential smoothing can be used 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 solution corpus, s_t is the smoothed probability value in SCSG of programs after kth problem and α is the smoothing factor. Note that if we use a moving average like exponential moving average, we do not give equal weight to all solutions, the most recent solutions have more weight. This may be considered to be equivalent to a kind of short-term activation of memory patterns in SCSG of programs. The rapid activation and inhibition of probabilities with a sufficiently high α is similar to the change of focus of a person. As for instance, shortly after studying a subject, we view other problems in terms of that subject. Therefore, it may also be suitable to employ a combination of time agnostic and time dependent probability calculations, to simulate both long-term and short-term memory like effects.

7.2 Memoization of Solutions

The memoization step, on the other hand, recalls precisely each program in a solution. The problem solver has already added the program of each solution i as a subprogram with a name $s_{i,j}$ for jth program of the solution, to the reference machine. For each solution, we also have the formal parameters for the solution. For instance, in an operator induction problem, there are two parameters $O^j(A_i|Q_i)$, the answer A_i and the question Q_i parameters, and the output is a probability value. Therefore, this step is only practical if reference machine M has a function call mechanism, and we can pass parameters easily. For most suitable M, this is true. Then, we add an alternative production to the grammar for each $s_{i,j}$. For instance, in LISP language, we may add an alternative production to the expression head (corresponding to LISP S-expressions) for an operator induction solution for solution i, and add individual productions for each $s_{i,j}$:

$$
\begin{aligned}
&\text{expression} \rightarrow_{p_i} \text{solution-i} \\
&\quad \text{solution-i} \rightarrow_{p_{i,1}} (s_{i,1}\ \text{expression expression}) \\
&\quad \text{solution-i} \rightarrow_{p_{i,1}} (s_{i,2}\ \text{expression expression}) \\
&\qquad \ldots \\
&\quad \text{solution-i} \rightarrow_{p_{i,j}} (s_{i,j}\ \text{expression expression})
\end{aligned}
$$

Naturally, the question of how to assign the probability values arises. We recommend setting p_i to a fixed, heuristic, initial value c_0, such as 0.1, and re-normalize other productions of the same head expression so that they add up to $1 - c_0$, preserving consistency. We expect the update production probabilities method to adjust the p_i in subsequent problems, however, initially it must be given a significant probability so that we increase the probability that it will be used shortly, otherwise, according to the smoothing method employed, the update production probabilities method may rapidly forget this new production. It is simple to set $p_{i,j}$. They may be determined by the formula:

$$
p_{i,j} = \frac{P(s_{i,j})}{\sum_j P(s_{i,j})} \tag{3}
$$

where $P(s_{i,j})$ is given by GPD (SCSG of programs) and should already be available in the solution corpus. In practice, it does not seem costly to maintain all $s_{i,j}$ in the solution corpus, as the induction algorithm will likely find only a small number of them at most (e.g., only 3-4). This may be intuitively understood as the number of alternative ways we understand a problem: we may be able to represent the problem with a diagram, or with a mathematical formula, but it is difficult to multiply the correct models arbitrarily. If we decide to spend a long time on a single induction problem, we may be able to come up with several alternative models, however, the number would not be infeasibly large for any non-trivial problem. That is, the stochastic memory system never entirely forgets any solution model $s_{i,j}$, it may only assign it a low probability.

7.3 Derivation Compression Algorithm

The derivation compression step adds common derivations to the SCSG of programs. With each solution program $s_{i,j}$ a derivation lattice $L_{i,j}$ is associated, and stored in the solution corpus along with the solution. After we have solved the nth problem, the statistics of the solution corpus change. The general idea is to use the non-trivial statistics of the solution corpus, in addition to production frequencies, to update the SCSG of programs. A sub-derivation L' of a given derivation lattice L is a derivation itself, and it is a subgraph of L and it is a proper derivation lattice, as well. For instance, we can follow a path from each input symbol to an output symbol, and it can only contain well-formed and complete productions in the current SCSG of programs. Such a subderivation corresponds to a derivation $\alpha \Rightarrow^* \beta$ where $\alpha = I(L)$ and $\beta = O(L)$, and L has a probability that corresponds to the product of probabilities of productions in L' (as in any other derivation lattice). We observe that we can represent any such derivation with a corresponding production $\alpha \rightarrow \beta$.

We find all sub-derivations L' that occur with a frequency above a given threshold t among all derivation lattices in the solution corpus. Well-known frequent subgraph mining methods may be used for this purpose. There are various efficient algorithms for solving the aforementioned mining problem which we shall not explain in much detail. However, it should be noted that if a sub-lattice L' is frequent, all sublattices of L' are also frequent, which suggests a bottom-up generate-and-test method to discover all frequent sub-lattices. The important point is that there is a complete search method to find all such frequent sub-lattices for any given t. Let us assume that $F = \{L_s \mid L_s$ is a sub-lattice of at least t $L_{i,j}$ lattices$\}$. Any frequent sub-graph mining algorithm may be used, and then we discard frequent sub-graphs that are not derivation lattices. Now, each $L_s \in F$ corresponds to an arbitrary production $\alpha \rightarrow \beta$. Our method incorporates such rules in the SCSG of programs, so that we will short-cut the same derivation steps in subsequent induction problems, when they would result in a solution program. However, we cannot add arbitrary derivations because we are using context-sensitive grammars. We first restrict F to only context-free productions, by searching for derivation lattices, that start with only one non-terminal. These would typically be expansions of commonly used non-terminals in the grammar, such as expression or definition. We remove the rest that do not correspond to context-free productions.

For each such new context-free sub-derivation $A \Rightarrow^* \beta$ discovered, we try to find a frequent context for the head non-terminal A. We assume that we can locate each

L_s in the pruned F set among $L_{i,j}$. The context of each production p_i in a derivation lattice can be calculated easily as follows. We give two methods. The sub-graph L_i that contains the derivations $p_1, p_2, \ldots, p_i$ are determined. $I(L_i)$ will contain the entire context for A up to derivation in the order of production applications, and since we know the location of A we can split the context into $\alpha A \beta$, obtaining a derivation $\alpha A \beta \Rightarrow^* \alpha\gamma\beta$. Alternatively, the level-order traversal of the lattice up to and including the level of p_i will tend to give a balanced context (left and right context will have close length).

After we obtain contexts for each frequent sub-derivation, we can use a sequence mining technique to discover frequent context-sensitive derivations in the solution corpus. Let $\alpha = a_1 a_2 \ldots a_n$ and $\beta = b_1 b_2 \ldots b_n$. Note that if sub-derivation $\alpha A \beta \Rightarrow^* \alpha\gamma\beta$ is frequent in the solution corpus, $a_2 \ldots a_n A \beta \Rightarrow^* a_2 \ldots a_n \gamma \beta$ and $\alpha A b_1 \ldots b_{n-1} \Rightarrow^* \alpha\gamma b_1 \ldots b_{n-1}$ are also frequent (in the solution corpus). Therefore, a simple bi-directional sequence mining method may be used to start with derivations of the form $\Lambda A \Lambda \Rightarrow^* \beta$ (the context is null), extending the context of A one symbol at a time to left and right, alternately, with candidate symbols from the already determined contexts of A in $L_{i,j}$ and then testing if the extended context is frequent, iterating the extension in both directions until an infrequent context is encountered, and outputting all frequent contexts found this way. That is to say, the well-known sequence mining algorithms are extended here to the bi-directional case, which is not difficult to implement, however, in our case, it is also an extremely effective method. After this second data mining step, we will have determined all sub-derivations with a frequency of at least t, of the form:

$$\alpha A \beta \Rightarrow^* \alpha\gamma\beta \tag{4}$$

which is precisely the general form of context-sensitive productions. In similar fashion to memoization of programs, we split the head non-terminal into two parts, the productions in G_0 and the new productions we have added:

$$A \rightarrow_{p_f} A_f \tag{5}$$

The rest of productions of A are normalized so that they add up to $1 - p_f$. p_f is initially given a high enough value (such as 0.1 in our present implementation), and this has to be done only once when the first frequent sub-derivation for A head is discovered. We expect that p_f will be adjusted appropriately during subsequent memory updates. The frequent sub-derivations may now be converted to productions of the form

$$\alpha A_f \beta \rightarrow_{p_i} \alpha\gamma\beta \tag{6}$$

and added to solution corpus. The problem of determining the probability of each production is inherent in the frequency values calculated during mining of frequent sub-derivations. The formula

$$p_i = \frac{|\{\alpha A \beta \Rightarrow^* \alpha\gamma\beta \in L(S)\}|}{|\{\bigcup_{\alpha,\beta,\gamma} \alpha A \beta \Rightarrow^* \alpha\gamma\beta \in L(S)\}|} \tag{7}$$

where we use multisets instead of sets (sets with duplicates), and the set union operator is meant to concatenate lists, assigns a probability value to each production in proportion to the computed frequencies of the corresponding frequent sub-derivation in the

solution corpus, which is an appropriate application of Laplace's rule. That is to say, we simply count the number of times the sub-derivation $\alpha A\beta \Rightarrow^* \alpha\gamma\beta$ has occurred in the solution corpus, and divide it by the number of times any production $xAy \Rightarrow^* xzy$ has occurred for any strings x, y, z.

If t is given too low, such as 2, an inordinate number of frequent sub-lattices will be found, and processing will be infeasible. We use a practical method to address this problem. We start with $t = n/2$, and then we use a test that will be explained below to determine if the productions are significant enough to add to SCSG of programs, if the majority of them are significant, we halve t, ($t = t/2$), and continue iteration until $t < 2$ at which point we terminate.

We can determine if a discovered production is significant by applying the set induction method to the GPD update problem. Recall that set induction predicts the most likely addition to a set, given a set of n items. A recognizer $R_i(\cdot)$ assigns probabilities to individual bit strings (items), this can be any program. Let there be a set of observed items $x_1, x_2, \ldots, x_n \in X$. What is the probability that a new item x_{n+1} is in the same set X? It is defined inductively as:

$$P(x \in X) = \sum_i P(R_i) \prod_j R_i(x_j) \tag{8}$$

using a prior $P(\cdot)$. Since we cannot find all such R_i, we try to find, using an induction algorithm to find R_i's that maximizes this value. The resulting R_i's give a good generalization of the set from observed examples. We now constrain this general problem to a simpler model of SCSG's following Solomonoff's probability model for context-free grammars:

$$P_G(G) = P(|N|).P(|\Sigma|).\frac{(k-1)!\prod_{i=1}^{k} n_i!}{k-1+\sum_{i=1}^{k} n_i!} \tag{9}$$

where $k = |N| + |\Sigma| + 2$ is the number of kinds of symbols, and n_i is the number of occurrences of each kind of symbol, since a simple string coding of the grammar requires k kinds of symbols, including all non-terminals and terminals, as well as two punctuation symbols sep_1 and sep_2 to separate alternative productions of the same head, and heads, respectively. The probabilities of integers can be set as Rissanen's integer distribution as Solomonoff has previously done. The same method can be applied to context-sensitive grammars, requiring only the additional change that, the context in $\alpha A\beta \rightarrow \alpha\gamma\beta$, that is α and β must also be coded, using sep_1. Note that this is a crude probability model and neglects probabilities, however, it can be used to determine whether a production or a set of productions added to the SCSG of programs improves the SCSG's goodness of fit Ψ:

$$\Psi = P_G(G_i) \prod_j P_{G_i}(s_j) \tag{10}$$

for each solution program s_j in the corpus, where by $P_{G_i}(s_j)$ we mean the probability assigned by grammar G_i to program s_j. This is a crucial aspect of the method, we only add context-sensitive productions that improve the extrapolation of the grammar, avoiding superfluous productions that may weaken its generalization power. Recall

Equation 8, first we simplify it by maintaining only the best R_i, that is the R_i that contributes most to probability of membership. We assume that our previous SCSG G_{N-1} was such a model. Now, we try to improve upon it, by evaluating candidate productions to be added to G_{n-1}. Let a new production be added this way, and let this grammar be called G'. Let Equation 9 be used to calculate the a priori probability of G_i, coded appropriately as a string, which corresponds to $P(R_i)$ in Equation 8. $R_i(x_j)$ term corresponds to the probability assigned to the solution program x_j by $SCSG\ G_i$. We thus, first calculate the goodness of fit for G_{n-1}, and then we calculate the goodness of the fit for G' (by re-parsing all solution programs in the solution corpus) and we only add the new production chosen this way if the goodness of fit Ψ increases. Otherwise, we decide that the gain from fitting the solution programs better does not compensate for the decrease in a priori probability, and we decide to discard the candidate productions. Alternatively, we can also test productions in batches, rather than one by one. Our preferred method is, however, for m candidate productions, we sort them in order of decreasing probability (Equation 7), and then we can test them one by one. We also maintain the number of successfully added productions so that we can decide to continue with the exploration of frequent sub-derivations.

It must be emphasized that more refined probability models of SCSG's will result in better generalization power for the derivation compression step, as a crude probability model may mistakenly exclude useful productions from being added. This is not a limitation of our proposal.

After the derivation compression step, all the solutions $s_{i,j}$ must be re-parsed with the latest grammar G_N, and updated in the solution corpus so that they all refer consistently to the same grammar, to avoid any inconsistency in subsequent runs of the derivation compression step.

8 Discussion

We show that we can extend OOPS while restoring the vital property of bias-optimality. We propose using a SCSG for GPD that extrapolates algorithmic information from already solved induction problems. We introduce a data structure for representing SCSG derivations. SCSG addresses the transfer learning problem by storing information about past solutions. A fast memory update algorithm is proposed which is comprised of three steps: updating production probabilities, memoization of programs and derivation compression. All of these steps use statistics of the solution corpus, which contains all solution programs that the induction approximation program discovers. They then modify or add productions in the grammar. Probabilities of productions are extrapolated from initial grammar and solution corpus. All new solution programs are memoized and added to the grammar as new productions with similarly conceived probabilities. Derivation compression is achieved with discovery of frequent sub-derivations in the solution corpus and result in additional productions. We decide which new productions to keep by SCSG induction, which is a straightforward extension of a stochastic context-free grammar induction method of Solomonoff. The resulting transfer learning architecture is quite practical since it maintains universality of Solomonoff induction, while presenting an extremely fast update algorithm for GPD's that are appropriate for complex

reference machines like LISP. In the future, we plan to demonstrate the performance of our transfer learning algorithms on extensive training sequences involving both deterministic and stochastic problems.

Bibliography

[1] Jing Huang, D. Schonfeld, and V. Krishnamurthy. A new context-sensitive grammars learning algorithm and its application in trajectory classification. In *Image Processing (ICIP) 2012*, pages 3093–3096, Sept 2012.

[2] Marcus Hutter. Optimality of universal bayesian sequence prediction for general loss and alphabet. *JMLR*, pages 971–1000, Nov 2003.

[3] L.A. Levin. Universal problems of full search. *Problems of Information Transmission*, 9(3):256–266, 1973.

[4] Moshe Looks. Scalable estimation-of-distribution program evolution. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, 2007.

[5] Moshe Looks, Ben Goertzel, and Cassio Pennachin. Learning computer programs with the Bayesian optimization algorithm. In *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 1, pages 747–748, Washington DC, USA, 25-29 June 2005. ACM Press.

[6] Robert I. McKay, Nguyen Xuan Hoai, Peter Alexander Whigham, Yin Shan, and Michael O'Neill. Grammar-based genetic programming: a survey. *Genetic Programming and Evolvable Machines*, 11(3-4):365–396, 2010.

[7] Eray Özkural. An application of stochastic context sensitive grammar induction to transfer learning: Appendix. Published on www at `https://examachine.net/papers/derivation-lattice.pdf`.

[8] Eray Özkural. Towards heuristic algorithmic memory. In Jürgen Schmidhuber, Kristinn R. Thórisson, and Moshe Looks, editors, *AGI*, volume 6830 of *Lecture Notes in Computer Science*, pages 382–387. Springer, 2011.

[9] Juergen Schmidhuber. Optimal ordered problem solver. *Machine Learning*, 54:211–256, 2004.

[10] Ray J. Solomonoff. A formal theory of inductive inference, part i. *Information and Control*, 7(1):1–22, March 1964.

[11] Ray J. Solomonoff. A formal theory of inductive inference, part ii. *Information and Control*, 7(2):224–254, June 1964.

[12] Ray J. Solomonoff. Complexity-based induction systems: Comparisons and convergence theorems. *IEEE Trans. on Information Theory*, IT-24(4):422–432, July 1978.

[13] Ray J. Solomonoff. A system for incremental learning based on algorithmic probability. In *Proceedings of the Sixth Israeli Conference on Artificial Intelligence*, pages 515–527, Tel Aviv, Israel, December 1989.

[14] Ray J. Solomonoff. Progress in incremental machine learning. In *NIPS Workshop on Universal Learning Algorithms and Optimal Search*, Whistler, B.C., Canada, December 2002.

[15] Ray J. Solomonoff. Three kinds of probabilistic induction: Universal distributions and convergence theorems. *The Computer Journal*, 51(5):566–570, 2008. Christopher Stewart Wallace (1933-2004) memorial special issue.

[16] Ray J. Solomonoff. Algorithmic probability: Theory and applications. In M. Dehmer and F. Emmert-Streib, editors, *Information Theory and Statistical Learning*, pages 1–23. Springer Science+Business Media, N.Y., 2009.

[17] Ray J. Solomonoff. Algorithmic probability, heuristic programming and agi. In *Third Conference on Artificial General Intelligence*, pages 251–157, 2010.