Bootstrap Learning via Modular Concept Discovery

Eyal Dechter
MIT
USA
edechter@mit.edu

Jon Malmaud
MIT
USA
malmaud@mit.edu

Ryan P. Adams
Harvard University
USA
rpa@seas.harvard.edu

Joshua B. Tenenbaum
MIT
USA
jbt@mit.edu

Abstract

Suppose a learner is faced with a domain of problems about which it knows nearly nothing. It does not know the distribution of problems, the space of solutions is not smooth, and the reward signal is uninformative, providing perhaps a few bits of information but not enough to steer the learner effectively. How can such a learner ever get off the ground? A common intuition is that if the solutions to these problems share a common structure, and the learner can solve some simple problems by brute force, it should be able to extract useful components from these solutions and, by composing them, explore the solution space more efficiently. Here, we formalize this intuition, where the solution space is that of typed functional programs and the gained information is stored as a stochastic grammar over programs. We propose an iterative procedure for exploring such spaces: in the first step of each iteration, the learner explores a finite subset of the domain, guided by a stochastic grammar; in the second step, the learner compresses the successful solutions from the first step to estimate a new stochastic grammar. We test this procedure on symbolic regression and Boolean circuit learning and show that the learner discovers modular concepts for these domains. Whereas the learner is able to solve almost none of the posed problems in the procedure's first iteration, it rapidly becomes able to solve a large number by gaining abstract knowledge of the structure of the solution space.

1 Introduction

Imagine that you know nothing about electronics but are supplied with a collection of circuit elements and asked to build a number of useful digital devices, such as an adder, a flip-flop, a counter, a multiplexer, and so on. Your initial attempts will probably be uniformly ineffective. If your only feedback is based on success or failure, you may have trouble learning anything. If, on the other hand, you already know the basics of circuit design, you are far more likely to generate some devices that are at least somewhat successful. In this case, you get feedback that helps you improve your designs and your design skills.

This example illustrates the central "bootstrapping" challenge for attempts to create intelligent agents without building in vast amounts of pre-specied domain knowledge. Learning from experience is typically most effective in a system that already knows much of what it needs to know, because that is when mistakes are most informative. In a system that knows almost nothing, where most problem solving attempts entirely miss their mark, failures are unlikely to shed light on what will work and what will not. Uninformative solutions vastly outnumber the rest, and an uninformed agent has no way to know which solutions will be informative and which will not. If it is harder to learn the less an agent knows, how can a learner who knows almost nothing about a domain ever get off the ground?

Here we propose an answer to this challenge, based on an approach to discovering new and increasingly sophisticated concepts that build modularly on previously learned, simpler concepts. Like many problems in AI, the bootstrapping challenge and its solution can be cast in terms of search. In our circuit-building scenario, the goal is to find circuit wiring procedures that maximize certain criteria. Search algorithms typically aim to exploit local or topological structure in the search space – here the space of all circuit-wiring procedures. For example, local search algorithms move from one candidate solution to a neighboring one. Heuristic search techniques take advantage of topological structure to prune off large parts of the search space. This structure derives from domain-specic knowledge. In local search, domain knowledge provides the similarity metric between solutions; in heuristic search it provides the heuristic. Here we ask how to bootstrap domain-specic knowledge by extracting search structure from the space of possible solutions by attempting to solve multiple related problems in a given domain at once.

Our approach is motivated by the observation that many interesting problems - such as discovering scientic theories, designing new technologies, learning a skill such as playing an instrument or cooking - have solutions that are naturally specied as programs. Our proposed solution is inspired by the programmer who, having found various subroutines that can be reused to solve similar simple tasks, creates a function library to help manage the complexity of larger projects. The E.C. algorithm, which we introduce here, builds a library of

reusable program components and places a distribution over these, effectively learning from simple tasks a search space structure that enables it to solve more complex tasks.

In this work, we try to see how far this idea of reusable function definition can get us. We ask the following question: can an AI system discover the structure latent in the solutions to multiple related problems and thereby bootstrap effective exploration of the search space? Can discovering modular program components transform a problem from one in which search is intractable into a problem in which it becomes increasingly feasible with experience?

Concretely, the E.C. algorithm works as follows. It begins with a small library of program primitives and a distribution over programs synthesized from this library. In each iteration, it generates programs in order of highest probability according to the library learned in the previous iteration. From the discovered solutions, it finds new reusable functions that compress these solutions, and it re-estimates the distribution over programs by weighting these new functions according to their frequencies. In using compression as a guide to the choice of representation, we instantiate the idea that good representations are those which minimize the description length of typical elements in the domain.

We show that this iterative approach allows us to achieve very high performance in tasks where just searching according to the initial distribution over program primitives clearly fails.

2 Related Work

The idea of discovering and using reusable subcomponents is part of an old tradition in AI. For instance, it is one of the central themes in Herb Simon's "The Sciences of the Artificial" [1996]. Rendell's "Substantial Constructive Induction Using Layered Information Compression: Tractable Feature Formation in Search" presented within a classical search paradigm the idea that compression provides a heuristic for constructing good representations [1985]. Within the Inductive Logic Programming (ILP) literature, researchers have explored "Constructive Induction," generating new and reusable logic programming terms [Muggleton, 1987]. Later, predicate invention was also applied in the multitask setting [Khan *et al.*, 1998]. We apply these ideas to learning functional programs and believe that the modularity and compositionality afforded by this representation is necessary for these ideas to be successful.

Genetic programming (GP) [Koza, 1992; Koza *et al.*, 1996] has tackled the problem of program learning and has explored the notion that reusable functions are helpful. This work relies, however, on stochastic local search over the space of programs, and automatically discovering useful subroutines is seen as a helpful heuristic. More recently, Liang et al. [2010] used a stochastic grammar over combinatory logic to induce shared structure in multi-task program induction problems. We find this latter representation compelling and use it here, but we see both this and genetic programming as tackling the question of how to benefit from shared structure in situations where local search might be successful on its own. Such problems rely on domain knowledge – whether in the form of a fitness function or a likelihood function – to provide local structure to the search space.

But, as mentioned before, our question is how to solve these problems when we lack knowledge of this kind of built in structure.

The remainder of this paper will include two major sections. In the first, we describe in detail the E.C. algorithm and design choices we make in implementing it. We will then present results from two experiments in the domains of symbolic regression and Boolean function learning.

3 Extracting Concepts from Programs

Our goal is to solve the following multi-task program induction problem: suppose we are given a set of **tasks** $T = \{t_k\}_{k=1}^{K}$ where each task is a function $t_k : \mathcal{L} \to \{0, 1\}$ and where $\mathcal{L}$ is the set of expressions in our language. We say that expression $e \in \mathcal{L}$ **solves** t_k if $t_k(e) = 1$. Our goal is to solve as many of the tasks in T as we can.

We propose the **E.C. algorithm**, an iterative procedure to solve this problem. It will be useful in presenting the algorithm to define the following terminology: a **frontier** is the finite set of expressions that the algorithm considers in any one iteration. The **frontier size** N is the pre-specified number of elements in the frontier. We will say that a task is **hit** by the frontier if there is a expression in the frontier that solves it.

The E.C. algorithm maintains a distribution $\mathcal{D}$ over expressions in $\mathcal{L}$. At each iteration, it

1. **explores** the frontier – enumerating the N most probable expressions from the current distribution $\mathcal{D}$ – and
2. **compresses** the hit tasks in the frontier to estimate a new distribution.

3.1 Representing Typed Functional Programs as Binary Trees

Following [Liang *et al.*, 2010] and [Briggs and O'Neill, 2006], we use a simply typed combinatory logic – a variable-free subset of the polymorphic simply-typed lambda calculus [Pierce, 2002] – as our program representation language. In short, the combinatory logic introduces a *basis* of several primitive *combinators* such that any function can be written as applications of the basis combinators. Some common basis combinators are defined as follows:

$$\mathbf{I}\ x \to x \qquad \text{(identity)} \qquad (1)$$

$$\mathbf{S}\ f\ g\ x\ \to (f\ x)\ (g\ x) \qquad (2)$$

$$\mathbf{C}\ f\ g\ x\ \to (f\ x)\ g \qquad (3)$$

$$\mathbf{B}\ f\ g\ x\ \to f\ (g\ x) \qquad \text{(composition)} \qquad (4)$$

The basis combinators are theoretically enough to express any Turing computable function, but we will assume that our language has a variety of primitives and, together with the basis combinators, we call these the **primitive combinators**. Note that by default we consider all primitives to be in their "curried" form (e.g. a function like "+" adds two numbers x and y by being first applied to x, returning the function $(+\ x)$ and then being applied to y, returning $(+\ x\ y)$).

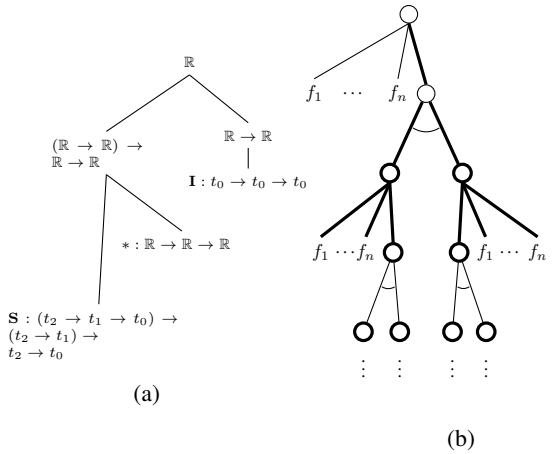


Figure 1: (a) The typed combinator $S * I$ ($f(x) = x^2$ over real values ($\mathbb{R}$)) represented as a binary tree. (b) The space of all typed combinators represented as an infinitely deep AND/OR tree.

The basis combinators can themselves be expressed in the lambda calculus (with definitions following directly from the equations above). The lambda calculus has two basic operations – application and abstraction – but in using the combinatory logic we sequester uses of the abstraction operation inside the combinators. In doing so, we have replaced the variable binding of the λ operator with the variable *routing* of these basis combinators. Our representation thus becomes variable-free. See [Liang *et al.*, 2010] for a more detailed discussion of this routing interpretation.

Using combinatory logic is very convenient for program synthesis [Briggs and O'Neill, 2006]: since every expression is the application of one expression to another – with this recursion bottoming out at the primitive combinators – each program is a binary tree. Most importantly, any subtree is itself a well-formed expression; this is not the case in the lambda calculus, since abstraction introduces long range dependencies between the λ operator and the variables to which that operator refers. In the lambda calculus, then, a subtree might have free variables, not bound to any enclosing λ.

As a simple example of our representation, consider the squaring function $\lambda x. * x\ x$. Using two of the basis combinators above, we can write the squaring function in combinatory logic as $S\ *\ I$ (taking $*$ as a primitive), where application associates to the left, so we drop the parentheses. When we apply this combinator to a value x, the action of the combinator is defined as $S\ *\ I\ x\ \to\ (*\ x)\ (I\ x)\ \to * x\ x$.

We can extend this representation with a simple polymorphic type system [Pierce, 2002]. In this representation, a type is either a type primitive (e.g. reals, integers, Booleans, etc.), a type variable (e.g. σ), or a function type $\tau_1 \to \tau_2$ of functions from source type τ_1 to target type τ_2. Any combinator can be represented as a binary tree (Figure 1a) whose leaves are typed primitive combinators and whose interior nodes represent typed applications of combinators (we annotate interior nodes with their types).

3.2 A Stochastic Grammar over Programs

We specify a distribution $\mathcal{D}$ over expressions in $\mathcal{L}$ as a stochastic grammar [Feldman *et al.*, 1969]. Many stochastic grammars have the desirable property that the probability of an expression is the product of the probabilities of its subcomponents. The distribution we specify will be a simple version of this, in which the probability of an expression is the product of the probability of its leaves.

Let $C = c_1, \ldots, c_N$ be the primitive combinators in $\mathcal{L}$. $\mathcal{D} = p_1, \ldots, p_N$ will associate with each primitive combinator c_i a prior probability p_i, $0 \le p_i \le 1, \sum_i p_i = 1$. The probability of a leaf with primitive combinator c_i will depend on which other primitive combinators could have been chosen in its place. This set will depend on the type that was requested by the parent of this leaf when the leaf's primitive was chosen; we call this type the *requesting* type of the leaf. For example, suppose that we have the primitive $(+1)$ of type $Int \to Int$ and the primitive NOT of type $Bool \to Bool$. If the requesting type is $\sigma \to Int$, then we can only use $(+1)$, but if the requesting type is $\sigma \to \sigma$ then we could use either $(+1)$ or NOT. We define C_τ for any type τ to be the set of primitive combinators in C that are consistent with τ, that is, whose types *unify* with τ (we use unification according to the standard type theory presentation [Pierce, 2002]).

Let C_e be the set of primitive combinators in the leaves of expressions e. We define the probability of an expression $e \in \mathcal{L}$ to be $p(e) = \prod_{c \in C_e} p(c|\tau(c))$ where $p(c|\tau(c))$ is the probability of using primitive c when the requesting type is $\tau(c)$. In turn, we define the conditional probability for each combinator c_n to be $p(c_n|\tau(c_n)) \propto \frac{p_n}{\sum_{c_j \in C_{\tau(c_n)}} p_j}$. That is, $p(c_n|\tau(c_n)$ is proportional to the probability of sampling c_n from the multinomial distribution defined by the p_n's, conditioned on the requesting type, with the constant of proportionality chosen to ensure that the sum over the probabilities of all expressions converges to a finite value (so that the distribution over expressions is proper). For binary trees, this is true whenever the constant of proportionality is less than $\frac{1}{4}$[1].

Now suppose that we have a set of expressions $\mathcal{E}$, and we wish to estimate the maximum likelihood values of the p_n's. If the choice of primitive at each leaf were not conditioned on the requesting type, we could just count all occurrences of each primitive in our expression set, and this would be proportional to the maximum likelihood estimate. This is the ML estimator for a multinomial distribution. However, the fact that we condition on the requesting type makes this a considerably more difficult optimization task. One of the simplest approximations we can make is to calculate the frequency with which each primitive combinator appears in $\mathcal{E}$ when the requesting type is such that it could have been chosen. This is a straightforward calculation, and we use it for its convenience, finding that our empirical results justify its use. Future inquiry will be needed to determine to what extent a more accurate estimator can improve these results.

[1] The probability mass M_d of all expressions of depth less than or equal to d can be written with the recurrence relation $M_d \le M_{d-1}^2 + M_1$. This has an asymptotic fixed point, as $d \to \infty$ if $x = x^2 + M_1$ has a solution, which is true if $M_1 \le 1/4$.

3.3 Best-first Enumeration of Programs

In order to explore the frontier of most promising programs, we need a procedure that enumerates the N most probable expressions. There has been recent interest in this problem, most of it focused on enumerating the shortest program satisfying some criterion [Katayama, 2005; Yakushev and Jeuring, 2009]. Our approach is to formulate the problem as a best-first exploration of an AND/OR tree (Figure 1b) [Nilsson, 1982; Hall, 1973]. In words, the enumeration procedure is best described by the following recursion: every program is either a primitive combinator OR it is a left child program applied to a right child program (that is, a left child AND a right child).

This can framed as the following AND/OR tree $\mathcal{G}$. Suppose we want a program of type τ. The root of $\mathcal{G}$ is an OR node of type τ. Its children are elements in C_τ (defined in the previous section) and one AND node of type τ with two children. Each child of an AND node has the same structure as the root node, with modified types: the type of its left child is $\sigma \to \tau$ (where σ is a fresh type variable). Since the type of the right child is constrained by the subtree rooted at the left child, we always expand the left child first. Once we have a complete left child program, we can use its target type as the type of the right child.

Recall that a valid partial solution $\mathcal{H}$ is a subtree of $\mathcal{G}$ satisfying the following properties: the root of $\mathcal{H}$ is the root of $\mathcal{G}$ and any OR node in $\mathcal{H}$ has at most one child. $\mathcal{H}$ is a complete solution (and thus a complete expression) if it is a partial solution whose leaves are leaves of $\mathcal{G}$, if each OR node in $\mathcal{H}$ has exactly one child, and if each AND node in $\mathcal{H}$ is connected to all of that node's children in $\mathcal{G}$.

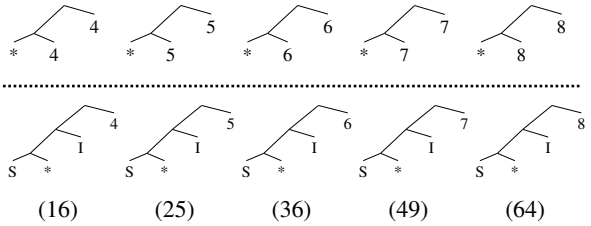
The value of a partial solution $\mathcal{H}$ is calculated by summing the value of its leaves, where each such value is the log of the probability assigned to that leaf in Section 3.2. If a leaf of $\mathcal{H}$ is a leaf of $\mathcal{G}$ then either it is an AND node, to which we assign a value of 0, or it corresponds to some primitive combinator c_n in the library and is the child of some typed OR node with type τ. We assign it a value of $\log p(c_n|\tau)$. If a leaf v of $\mathcal{H}$ is an OR node, then the value of any extension of it to a complete solution is at most $log(1/4)$. Thus, we have defined a value for partial solutions that upper bounds the value of any of its extensions.

Our best-first policy is to expand the next partial solution with the highest value. If a partial solution has multiple leaves that are not primitive combinators, we expand the left most one first, since it is the choice of left combinator that defines the type of its right combinator in an application pair. That is, the parent node applies its constraint on valid solutions via the left child, not the right child. We want to apply constraints as early as possible, so we expand the left child first.

3.4 Finding the Most Compressive Set of Solutions

Having enumerated the frontier, we want to assign solutions to hit tasks such that this set is maximally compressible. This will promote reusable modular subtrees.

A natural metric for the compressibility of a set of binary trees is the number of unique subtrees (counting leaves) that this set contains. As an example of this, suppose we have six tasks whose solutions are integers $4^2, 5^2, 6^2, 7^2$, and 8^2, and suppose our frontier contains the 10 expressions:



(16) (25) (36) (49) (64)

The top row consists of one type of representation of an integer squared, namely, the integer times itself. The bottom row contains another type of representation in which the function $f(x) = x^2$, represented in combinatory logic as $S * I$, is applied to the integer. The top and bottom row of each expression evaluate to the same integer. If we consider only the first column, the top tree has 4 unique subtrees and the bottom has 7. But each column adds 3 unique subtrees to the top row and only 2 to the bottom. So by the fifth column the top row has more unique subtrees than the bottom row. The increased compressibility of the bottom row is due to the fact that the squaring operation has been factored out as an abstraction. This is an example of how this compressibility metric favors abstraction when multiple expressions are being considered at once.

More sophisticated metrics than the one proposed here exist; we could use the negative logarithm of the probability of the solutions under a suitable probability distribution [Barron *et al.*, 1998]. Future work will explore whether this sophistication is worthwhile and which prior on grammars makes sense. For our purposes here, penalizing according to the number of unique subtrees appropriately and sufficiently rewards reuse of useful subtrees.

We want to choose a solution for each hit task such that the complete set of such solutions for all solved tasks is as compressible as possible. That is, let t_i be a task in the set of hit tasks, and let $\{e_i^k\}$ be the expressions in the frontier that solve it. Then our goal is to find

$$\{e_i^*\} = \arg\min_{\{e_i^k\}} |\cup_i e_i^k|,$$

where $|\cdot|$ denotes the number of unique trees in a set of expressions.

However, an exact optimization of this problem requires examining $O(M^K)$ assignments, where M is the maximum number of solutions for any one task, and K is the number of tasks. Since this is prohibitive, we relax the compressibility metric as follows: we define the cost of adding each new solution s_n to a partial set of solutions $s_1, \ldots, s_{n-1}$ to be the number of additional unique subtrees in s_n that are not present in s_{n-1}. That is, the penalty of adding a new expression to the set of expressions depends only on the last expression we added. Our solution thus becomes approximate, and that approximation is order-dependent, but a depth-first search on the defined search space goes from exponential in the solution degree to quadratic ($O(KM^2)$).

However, when the number of solutions in the frontier grows, this quadratic computation becomes prohibitive.

Therefore, in practice, we bound the number of potential expressions we are willing to consider at any one iteration to any one task to a small number. We find that a setting of this bound to 2 is computationally feasible and large enough to guide subcomponent discovery in a reliable way. It may be necessary to increase this bound as we scale this algorithm to larger problems.

3.5 Re-estimating the Stochastic Grammar

Given a set of chosen solution expressions, which we will call the *solution set*, want to re-estimate our stochastic grammar. Our inspiration for doing this is the Nevill-Manning algorithm for grammar-based compression [Nevill-Manning and Witten, 1997]. The idea of that compression algorithm is to compress any string by creating a grammar for that string such that a) no two symbols appear more than once in the grammar and b) every rule is used more than once. From a set of expressions, we generate a grammar according to these criteria.

This procedure generates a parse of the solution set with a new set of primitives, and we keep count of how many times each primitive occurred in the solutions set. To estimate the probabilities associated with each node production, we again traverse the solution set, but this time for each node we count which other primitive elements from the new grammar could have been used. In accordance with Section 3.2, we estimate the terminal production weight to be the ratio of the number of times that node was used in the first traversal divided by the number of times it was used in the second traversal.

4 Experiments.

4.1 Symbolic regression.

We first illustrate the performance of our learning algorithm on a symbolic regression problem. For each task in this problem, we want to find an algebraic function, symbolically specified, that maps a set of input values to output values. We choose this problem because it has a long history in AI, particularly in the genetic programming literature [Koza, 1993].

In the formulation we consider, each task t corresponds to a polynomial f, and an expression e solves t if it returns the same value when applied to $i \in 0, \ldots, 9$ that f does. In the experiment shown here, the set of problems corresponds to the set of all polynomials with degree less than or equal to 2. In the initial library, we include the basic combinators I, S, B, C and four arithmetic primitives 1, 0, $*$, and $+$. The choice of initial weights on these primitives can make a difference, as it determines the relative ordering of combinators in the enumeration step. To get a general sense of the algorithm's performance, we set the initial weights to be slightly different on each run, perturbing them around a uniform weighting.

In Figure 2, we show performance results as we vary the frontier size. Changing the frontier size changes the number of tasks the algorithm identifies correctly over the course of 15 algorithm iterations (all experiments in this work were run for 15 iterations). As a baseline, we enumerated $10000 * 15 = 150000$ expressions from the initial grammar. This is the total number of expressions that a run of the algorithm sees if it has a frontier size of 10000 and runs for 15 iterations. Thus, if the

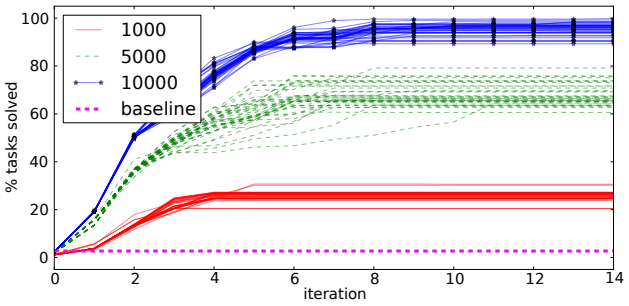


Figure 2: Learning curves as a function of frontier size. As frontier size is increased, curves plateau closer to 100% performance. A baseline search over 150000 expressions only hits 3% of the tasks (dashed pink line).

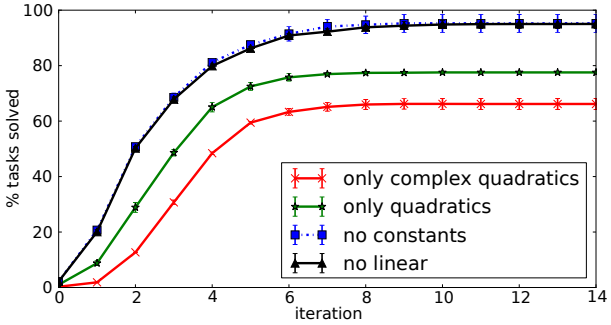


Figure 3: How do different task sets affect learning curves? Learning curves at a frontier size of 10000 for different task sets.

benefit accruing to the runs with larger frontier sizes is simply due to an increase in the number of expressions seen, rather than increased learning, we should see similarly good performance from this baseline run. In fact, however, the baseline run only hits 27 of the 1000 tasks (3%), whereas our algorithm nears 100% for a frontier of 10000.

What does the E.C. algorithm learn in this task? Inspecting the top weighted primitives of the final grammars, we find many incrementers (e.g. (+1), (+2), etc.), several versions of expressions that translate to functions like $x * f(x)$ and $f(x * f(x))$, and combinations of these, like $x * (x + 3) + 3$ and $x * (x + 2)$. Informally, we see expressions that apply functions to themselves, building up complex functions with relatively few unique primitives.

To what degree is "conceptual bootstrapping" responsible for the improved performance. That is, to what extent does the presence of simple problems account for the ability to learn complex functions? One hypothesis is that to succeed on a set of tasks, the set must contain a "learnable" curriculum, a set of tasks that serve as stepping stones from an impoverished representation to a rich one. We can test this hypothesis by varying the set of tasks to which the E.C. algorithm is exposed. If it is true, then we should see a nonlinear response to reducing the number of easy tasks, as the curriculum changes from being learnable to being unlearnable.

In Figure 3, we present learning curves (frontier

size 10000) corresponding to various versions of our original symbolic regression task set. Recall that the original set consisted of all polynomials of degree two or less and with coefficients between 0 and 9. In the "no constant" task set, we remove the 9 constant functions in this set. In the "no linear" task set, we remove 90 linear functions from the set. We observe that performance on those task sets does not decrease. However, when we remove both the linear functions and the constant function ("only quadratics"), we see a sharp drop in the algorithm's performance. When we further restrict the task set to only "complex" quadratics (which we define as quadratics whose coefficients are greater than zero), we observe another comparable drop in performance. When we go one step further and restrict the task set to quadratics with coefficients greater than 1, performance drops to 0 because no task is hit in the initial frontier.

This data has in it the nonlinearity that we predicted – that a minimal curriculum of simple tasks is sufficient to achieve high performance – but also suggests that, at least in this domain, this is not an all-or-none effect. Though the performance drops significantly once no linear functions are included in the task set, learning does still occur, and the learning curves still have the same basic shape.

4.2 Boolean Function Learning

As another demonstration of our approach, we investigate the E.C. algorithm's ability to learn Boolean functions using only the logical primitive NAND and the basic combinators. It is well known that a Boolean circuit can be constructed for any Boolean function given only the NAND gate. Here, the basic combinators take the place of the wiring normally found in a circuit.

We constructed two task sets. The first of these was constructed explicitly to contain familiar modular structure. In this set of tasks, we sampled 1000 Boolean circuits using AND, OR, and NOT gates. To accomplish this, we first sampled either 1, 2 or 3 inputs, then between 1 and 5 gates, randomly wiring the inputs of each new gate to the output of one of the existing gates in the circuit. We continued this sampling procedure until we had 1000 "connected" circuits, i.e., circuits all of whose outputs are wired to an input except for the last one (the output gate). This resulted in 1000 tasks consisting of 82 unique Boolean functions, with a distribution of tasks as shown in Figure 5a. In Figure 6a, we present learning curves for frontier sizes 100, 500 and 10000, using the same procedure as in Section 4.1. In Figure 5, we show that the distribution over Boolean functions enumerated from the grammar over expressions is much more similar to the true function distribution after learning than before.

This problem is particularly suitable for inspecting the constituents of the induced grammar $\mathcal{G}$. We might hope that our algorithm recovers the constituent logic gates that were used to build up the tasks. Table 4 shows a few of the top ranked expressions in the library. These include the basic logic gates; we also include two higher-order expression E_1 and E_2, to stress that the representation the E.C. algorithm is using allows it to build concepts that are more expressive that just sub-circuits.

In a second experiment, we include all 272 Boolean truth

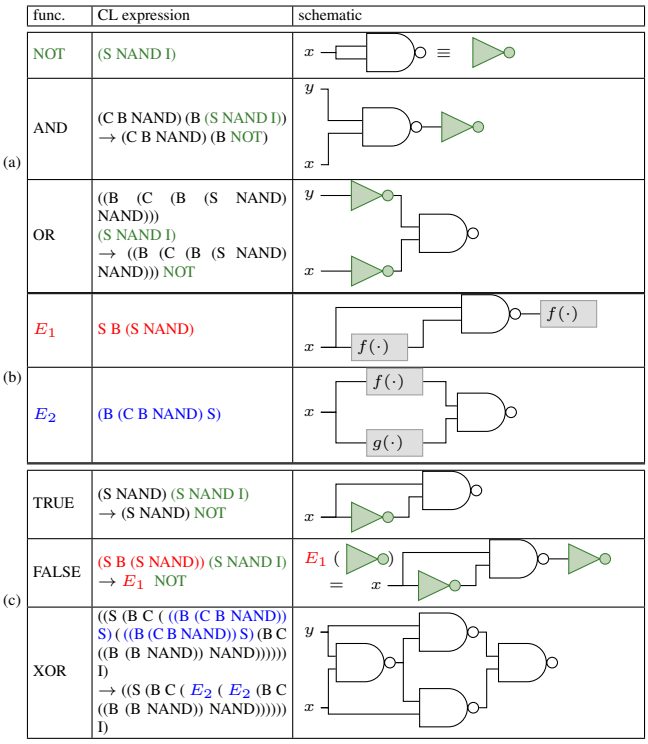


Figure 4: Some learned primitives from the circuit based Boolean function learning experiment. We highlight compression using NOT, E_1 and E_2. (a) The three elementary gates used to generate the task set. (b) Two higher-order primitives in the induced grammar. Note how E_1 can be used to define FALSE by taking the NOT function as an argument. (c) Additional common Boolean functions found in the learned grammar.

tables with three or fewer inputs as tasks. This is a more difficult problem. There are two kinds of learning that the E.C. algorithm might accomplish: first, many expressions in $\mathcal{L}$ map to a single Boolean function, so it needs to learn primitives that allow it to span many different functions instead of generating very simple functions redundantly. The second kind of learning involves the distribution of the functions themselves. In the circuit based Boolean experiment, that structure is apparent in the distribution in Figure 5. In this second experiment, we remove the second kind of structure. In Figure 6b, we show 10 runs of the E.C. algorithm on the second experiment with a frontier size of 2000: note how there are several local minima that the majority of the runs get stuck in with performance around 50%, but several of the runs seem to take different trajectories to more successful representations.

5 Conclusion

This work brings together ideas from various areas of research – hierarchical Bayes, learning to learn, minimum description length learning, and learning programs – to provide a proof of concept for how bootstrap learning of abstract composable concepts can rapidly facilitate problem solving in the absence of domain specific knowledge. We show that this

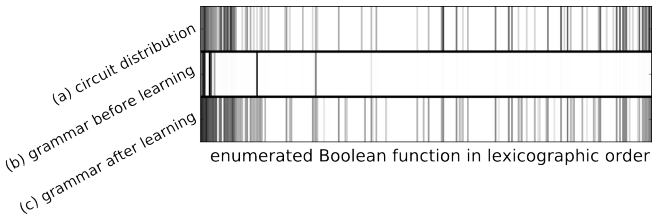


Figure 5: Distribution of Boolean functions among (a) the set of 1000 circuits randomly constructed from AND, OR and NOT gates (see text); b) the first 1000 expressions from the grammar over expressions *before* learning; c) the first 1000 expressions from the grammar over expressions *after* learning. Gray levels indicate percent of instances (circuits/expressions) which evaluate to that function. Functions increase in domain size from left to right; each function can be represented as a binary string corresponding to the output column of the truth table, and functions are ordered with respect to these strings.

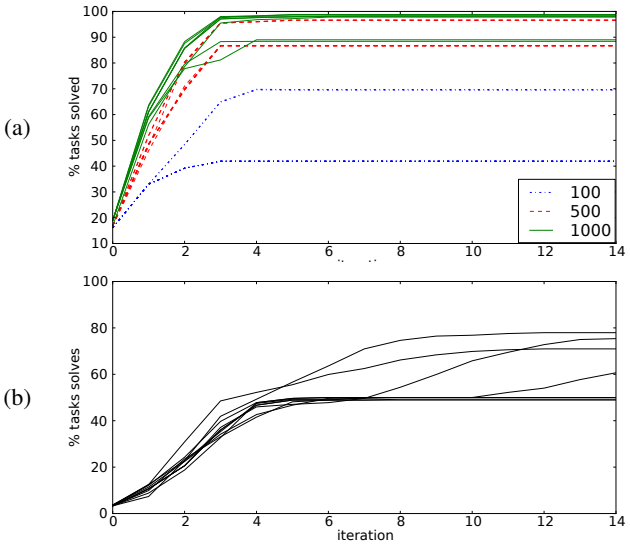


Figure 6: Learning curves from Boolean function learning experiments. (a) Learning curves for various frontier sizes on task set of circuit based Boolean functions. (b) 10 learning curves for frontier size of 2000 on a task set consisting of all Boolean functions of cardinality 3 or smaller. Note how most of the trajectories get stuck in a local minimum around 50%.

approach can learn successfully even in domains for which the solution space is not smooth and the error signal is all or none, where the only sense of locality to guide learning is the modular and compositional structure of the solutions that the algorithm itself builds.

The problem presented here was chosen to highlight the vastness of the search space confronting an agent with minimal background knowledge: the reward function is binary, it is deterministic, and it cannot provide feedback about partial solutions. In ongoing work, we are exploring a more probabilistic version of the algorithm that can take into account continuous rewards, noisy data and partial solutions.

Our specific implementation of this approach involved a number of choices which should be explored further in future work. Our claim here is not that the E.C. algorithm is optimal, but that it captures key features of how people solve the problem of learning new systems of concepts in acquiring deep domain expertise. Much like a human researcher investigating a new area of study, our algorithm tries different models, watches them fail or succeed, and abstracts out the model elements that capture relevant differences. It recombines and reuses successful patterns of reasoning that span many problems in a domain.

At their best, human processes of discovery clearly go beyond what our algorithm is capable of. The algorithm can get stuck in plateaus after a rapid initial increase in performance, due to its circular nature: it only inspects the region of the solution space that is typical of solutions already found. Humans can become stuck in similar ways, but at least sometimes they realize that they are stuck and attempt to find another class of concepts for the remaining unsolved tasks. Our algorithm requires, as human learners do, a sufficiently graded "curriculum," or spectrum of problems to solve – simple problems provide necessary stepping stones to building complex representations. Also like humans, our algorithm can adaptively "self-pace" its ways through the curriculum, figuring out which problems it is ready to tackle when, which problems are appropriately difficult given what it currently knows. Yet people can sometimes more intelligently compose and modify the set of problems under consideration, designing their own stepping stones to expand the set of problems they can solve. Making bootstrap learning systems smarter in these more human-like ways is a prime target for future research.

6 Acknowledgements

This work was supported by ARO MURI contract W911NF-08-1-0242 and by a grant from Google. It was also funded in part by DARPA Young Faculty Award N66001-12-1-4219.

References

[Barron *et al.*, 1998] Andrew R. Barron, Jorma Rissanen, and Bin Yu. The minimum description length principle in coding and modeling. *IEEE Transactions on Information Theory*, 44(6):2743–2760, 1998.

[Briggs and O'Neill, 2006] Forrest Briggs and Melissa O'Neill. Functional genetic programming with combinators. In *Proceedings of the Third Asian-Pacific workshop on Genetic Programming, ASPGP*, pages 110–127, 2006.

[Feldman *et al.*, 1969] Jerome A. Feldman, James Gips, James J. Horning, and Stephen Reder. *Grammatical complexity and inference*. Stanford University, 1969.

[Hall, 1973] Patrick A. V. Hall. Equivalence between AND/OR graphs and context-free grammars. *Commun. ACM*, 16(7):444–445, 1973.

[Katayama, 2005] Susumu Katayama. Systematic search for lambda expressions. In Marko C. J. D. van Eekelen, editor, *Trends in Functional Programming*, volume 6 of *Trends in Functional Programming*, pages 111–126. Intellect, 2005.

[Khan *et al.*, 1998] Khalid Khan, Stephen Muggleton, and Rupert Parson. Repeat learning using predicate invention. In David Page, editor, *ILP*, volume 1446 of *Lecture Notes in Computer Science*, pages 165–174. Springer, 1998.

[Koza *et al.*, 1996] John R. Koza, Forrest H. Bennett III, David Andre, and Martin A. Keane. Reuse, parameterized reuse, and hierarchical reuse of substructures in evolving electrical circuits using genetic programming. In Tetsuya Higuchi, Masaya Iwata, and Weixin Liu, editors, *ICES*, volume 1259 of *Lecture Notes in Computer Science*, pages 312–326. Springer, 1996.

[Koza, 1992] John R. Koza. Hierarchical automatic function definition in genetic programming. In L. Darrell Whitley, editor, *FOGA*, pages 297–318. Morgan Kaufmann, 1992.

[Koza, 1993] John R. Koza. *Genetic programming - on the programming of computers by means of natural selection*. Complex adaptive systems. MIT Press, 1993.

[Liang *et al.*, 2010] Percy Liang, Michael I. Jordan, and Dan Klein. Learning programs: A hierarchical Bayesian approach. In *International Conference on Machine Learning (ICML)*, pages 639–646, 2010.

[Muggleton, 1987] Stephen Muggleton. Duce, an oracle-based approach to constructive induction. In John P. McDermott, editor, *IJCAI*, pages 287–292. Morgan Kaufmann, 1987.

[Nevill-Manning and Witten, 1997] Craig G. Nevill-Manning and Ian H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.

[Nilsson, 1982] N.J. Nilsson. Principles of artificial intelligence. *Symbolic Computation, Berlin: Springer, 1982*, 1, 1982.

[Pierce, 2002] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1 edition, 2002.

[Rendell, 1985] Larry A. Rendell. Substantial constructive induction using layered information compression: Tractable feature formation in search. In Aravind K. Joshi, editor, *IJCAI*, pages 650–658. Morgan Kaufmann, 1985.

[Simon, 1996] Herbert A. Simon. *The Sciences of the Artificial*. MIT Press, 1996.

[Yakushev and Jeuring, 2009] Alexey Rodriguez Yakushev and Johan Jeuring. Enumerating well-typed terms generically. In Ute Schmid, Emanuel Kitzelmann, and Rinus Plasmeijer, editors, *AAIP*, volume 5812 of *Lecture Notes in Computer Science*, pages 93–116. Springer, 2009.