
Learning Libraries of Subroutines for Neurally-Guided Bayesian Program Learning

Anonymous Author(s)

Affiliation

Address

email

Abstract

1 Successful approaches to program induction require a hand-engineered domain-
2 specific language (DSL), constraining the space of allowed programs and imparting
3 prior knowledge of the domain. We contribute a program induction algorithm
4 called ECC that learns a DSL while jointly training a neural network to efficiently
5 search for programs in the learned DSL. We use our model to synthesize functions
6 on lists, edit strings, and solve symbolic regression problems, showing how the
7 model learns a domain-specific library of program components for expressing
8 solutions to problems in the domain.

9 1 Introduction

10 Imagine that you are asked to write a program to edit some text, and told that you should change the
11 text “Nancy FreeHafer” to “Dr. Nancy”. From this example, you likely infer that “Jane Goodall”
12 should be “Dr. Jane”, drawing upon your prior knowledge of text, like that words are separated by
13 spaces. For few-shot program learning problems like these, the program synthesis and AI communities
14 have had great success engineering program learning systems, not just for text editing [1] but also
15 for graphics programs [2], planning procedures [26], and many others. However, the success of
16 these systems hinges upon having a carefully hand-engineered **Domain Specific Language (DSL)**.
17 DSLs impart prior knowledge of a domain by providing a restricted set of finely-tuned programming
18 primitives: for text editing, these are primitives like appending and splitting on characters. In this
19 work, we consider the problem of building agents that solve program learning tasks, and also the
20 problem of acquiring the prior knowledge necessary to quickly solve these tasks (Figure 1). Our
21 solution is an algorithm that learns a DSL while jointly training a neural network to write programs
in that learned DSL.

TASK	$\begin{array}{ccc} \text{Nancy FreeHafer} & \longrightarrow & \text{Dr. Nancy} \\ \text{Jane Goodall} & \longrightarrow & ??? \end{array}$
PROGRAM	$f(s) = (f_0 \text{ "Dr. " } (f_2 s \text{ " "}))$
LIBRARY (DSL)	$\begin{array}{l} f_0(a,b) = (\text{fold } a \ b \ (\text{lambda } (x \ y) \\ \quad (\text{cons } x \ y))) \\ (f_0: \text{Appends lists (of characters)}) \\ f_1(s,c) = (\text{fold } s \ s \ (\text{lambda } (x \ a) \\ \quad (\text{if } (= c \ x) \ \text{nil} \ (\text{cons } x \ a)))) \\ (f_1: \text{Take characters from } s \text{ until } c \text{ reached}) \end{array}$

Figure 1: **Task:** Few-shot learning problem. Model solves tasks by writing **programs**, and jointly learns a **library** of reusable subroutines that are shared across multiple tasks, called a **Domain Specific Language (DSL)**. Program writing is guided by a neural network trained jointly with the library.

22

23 We take inspiration from two sources: (1) Good software engineers compose libraries of reusable
24 subroutines that are shared across related programming tasks. Returning to Figure 1, a good string


Domain	Example Task		Part of the learned DSL
Lists	[7 0 2]	→ [7 0 2 4]	(foldr nil (lambda (a b) (cons a b)))
	[3 9]	→ [3 9 4]	(appends lists)
Strings	Temple Anna H	→ TAH	(map (lambda (x) (if (= x a) b x)))
	Lara Gregori	→ LG	(replace occurrences of a w/ b)
Regression			(+ (* real x) real)
			(a linear function of x)

Figure 2: Examples of structure found in DSLs learned by our algorithm. ECC builds a new DSL by discovering and reusing useful subroutines.

25 processing library should support appending strings and splitting on spaces – exactly the prior
26 knowledge needed to solve the task in Figure 1. (2) Skilled human programmers can quickly
27 recognize what kinds of programming idioms and library routines would be useful for solving the
28 task at hand, even if they cannot instantly work out the details. We combine these two ideas into
29 an algorithm called ECC, which takes as input a collection of programming **tasks**, and then jointly
30 solves three problems: (1) Writing programs that solve the tasks; (2) Composing a library of domain-
31 specific subroutines – which allow the agent to more compactly write programs in the domain, and
32 (3) Training a neural network to recognize which library components are useful for which kinds of
33 tasks. Together, the library and neural net encode the domain specific knowledge needed to quickly
34 write programs.

35 Our algorithm is called **Explore/Compress/Compile** (ECC), because it iterates between three dif-
36 ferent steps: an **Explore** step uses the DSL and neural network to explore the space of programs,
37 searching for ones that solve the tasks; a **Compress** step modifies the structure of the DSL by discov-
38 ering regularities across programs found by the previous Explore step; and a **Compile** step, which
39 improves the program search procedure by training a neural network to write programs in the current
40 DSL, in the spirit of “amortized” or “compiled” inference [9]. We call the neural net a **recognition**
41 **model** (c.f. Hinton 1995 [10]). The learned DSL distills commonalities across programs that solve
42 tasks, helping the agent solve related program induction problems. The neural recognition model
43 ensures that searching for programs remains tractable even as the DSL (and hence the search space
44 for programs) expands.

45 Because any model may be encoded as a (deterministic or probabilistic) program, we carefully
46 delineate the scope of problems considered here. We think of ECC as learning to solve the kinds of
47 problems that humans can solve relatively quickly – once they acquire the relevant domain expertise.
48 These correspond to short programs – if you have an expressive DSL. Even with a good DSL, program
49 search may be intractable, so we amortize the cost of program search by training a neural network to
50 assist the search procedure.

51 We apply ECC to four domains: list processing; FlashFill-style [1] string editing; and symbolic
52 regression. For each of these we initially provide a generic set of programming primitives. Our
53 algorithm then discovers its own DSL for expressing solutions in the domain (Tbl. 2).

54 Prior work on program learning has largely assumed a fixed, hand-engineered DSL, both in clas-
55 sic symbolic program learning approaches (e.g., Metagol: [5], FlashFill: [1]), neural approaches
56 (e.g., RobustFill: [6]), and hybrids of neural and symbolic methods (e.g., Neural-guided deductive
57 search: [7], DeepCoder: [8]). A notable exception is the EC algorithm [12], which also learns a
58 library of subroutines. We were inspired by EC, and go beyond it by giving a new algorithm for
59 learning DSLs, as well as a way of combining DSL learning with neurally guided program search.
60 In the experiments section, we compared directly with EC, showing empirical improvements. The
61 contribution of this work is then an effective algorithm for learning DSLs which also trains a neural
62 net to search for programs in the DSL.

63 2 The ECC Algorithm

64 Our goal is to induce a DSL while finding programs solving each of the tasks. We take inspiration
65 primarily from the Exploration-Compression algorithm for bootstrap learning [12]. Exploration-

66 Compression alternates between exploring the space of solutions to a set of tasks, and compressing
 67 those solutions to suggest new search primitives for the next exploration stage. We extend these
 68 ideas into an inference strategy that iterates through three steps: an **Explore** step uses the current
 69 DSL and recognition model to search for programs that solve the tasks. The **Compress** and **Compile**
 70 steps update the DSL and the recognition model, respectively. Crucially, these steps synergistically
 71 bootstrap off each other:

72 **Exploration: Searching for programs.** Our program search is informed by both the DSL and the
 73 recognition model. When these improve, we can solve more tasks.

74 **Compression: Improving the DSL.** We induce the DSL from the programs found in the exploration
 75 phase, aiming to maximally compress (or, raise the prior probability of) these programs. As we solve
 76 more tasks, we hone in on DSLs that more closely match the domain.

77 **Compilation: Learning a neural recognition model.** We update the recognition model by training
 78 on two data sources: samples from the DSL (as in the Helmholtz Machine’s “sleep” phase), and
 79 programs found by the search procedure during exploration. As the DSL improves and as search
 80 finds more programs, the recognition model gets more data to train on, and better data.

81 2.1 Hierarchical Bayesian Framing

82 ECC takes as input a set of *tasks*, written X , each of which is a program synthesis problem. It has at
 83 its disposal a domain-specific *likelihood model*, written $\mathbb{P}[x|p]$, which scores the likelihood of a task
 84 $x \in X$ given a program p .¹ Its goal is to solve each of the tasks by writing a program, and also to
 85 infer a DSL, written \mathcal{D} . We equip \mathcal{D} with a real-valued weight vector θ , and together (\mathcal{D}, θ) define a
 86 generative model over programs. We frame our goal as maximum a posteriori (MAP) inference of
 87 (\mathcal{D}, θ) given X . Writing J for the joint probability of (\mathcal{D}, θ) and X , we want the \mathcal{D}^* and θ^* solving:

$$J(\mathcal{D}, \theta) \triangleq \mathbb{P}[\mathcal{D}, \theta] \prod_{x \in X} \sum_p \mathbb{P}[x|p] \mathbb{P}[p|\mathcal{D}, \theta]$$

$$\mathcal{D}^* = \arg \max_{\mathcal{D}} \int J(\mathcal{D}, \theta) d\theta \quad \theta^* = \arg \max_{\theta} J(\mathcal{D}^*, \theta) \quad (1)$$

88 The above equations summarize the problem from the point of view of an ideal Bayesian learner.
 89 However, Eq. 1 is wildly intractable because evaluating $J(\mathcal{D}, \theta)$ involves summing over the infinite
 90 set of all programs. In practice we will only ever be able to sum over a finite set of programs. So, for
 91 each task, we define a finite set of programs, called a *frontier*, and only marginalize over the frontiers:
 92 **Definition.** A *frontier of task x* , written \mathcal{F}_x , is a finite set of programs s.t. $\mathbb{P}[x|p] > 0$ for all $p \in \mathcal{F}_x$.

93 Using the frontiers we define the following intuitive lower bound on the joint probability, called \mathcal{L} :

$$J \geq \mathcal{L} \triangleq \mathbb{P}[\mathcal{D}, \theta] \prod_{x \in X} \sum_{p \in \mathcal{F}_x} \mathbb{P}[x|p] \mathbb{P}[p|\mathcal{D}, \theta] \quad (2)$$

94 ECC does approximate MAP inference by maximizing this lower bound on the joint probability,
 95 alternating maximization w.r.t. the frontiers (Exploration) and the DSL (Compression):

96 **Program Search: Maxing \mathcal{L} w.r.t. the frontiers.** Here (\mathcal{D}, θ) is fixed and we want to find new
 97 programs to add to the frontiers so that \mathcal{L} increases the most. \mathcal{L} most increases by finding programs
 98 where $\mathbb{P}[x, p|\mathcal{D}, \theta]$ is large.

99 **DSL Induction: Maxing $\int \mathcal{L} d\theta$ w.r.t. the DSL.** Here $\{\mathcal{F}_x\}_{x \in X}$ is held fixed, and so we can
 100 evaluate \mathcal{L} . Now the problem is that of searching the discrete space of DSLs and finding one
 101 maximizing $\int \mathcal{L} d\theta$. Once we have a DSL \mathcal{D} we can update θ to $\arg \max_{\theta} \mathcal{L}(\mathcal{D}, \theta, \{\mathcal{F}_x\})$.

102 Searching for programs is hard because of the large combinatorial search space. We ease this
 103 difficulty by training a neural recognition model, $q(\cdot|\cdot)$, during the compilation phase: q is trained to
 104 approximate the posterior over programs, $q(p|x) \approx \mathbb{P}[p|x, \mathcal{D}, \theta] \propto \mathbb{P}[x|p] \mathbb{P}[p|\mathcal{D}, \theta]$, thus amortizing
 105 the cost of finding programs with high posterior probability.

106 **Neural recognition model: tractably maxing \mathcal{L} w.r.t. the frontiers.** Here we train $q(p|x)$ to
 107 assign high probability to programs p where $\mathbb{P}[x, p|\mathcal{D}, \theta]$ is large, because including those programs
 108 in the frontiers will most increase \mathcal{L} .

¹For example, for string editing, the likelihood is 1 if the program predicts the observed outputs on the
 observed inputs, and 0 otherwise.

2.2 Exploration: Searching for Programs

Now our goal is to search for programs solving the tasks. We use the simple approach of enumerating programs from the DSL in decreasing order of their probability, and then checking if a program p assigns positive probability to a task ($\mathbb{P}[x|p] > 0$); if so, we incorporate p into the frontier \mathcal{F}_x .

To make this concrete we need to define what programs actually are and what form $\mathbb{P}[p|\mathcal{D}, \theta]$ takes. We represent programs as λ -calculus expressions. λ -calculus is a formalism for expressing functional programs that closely resembles Lisp, including variables, function application, and the ability to create new functions. Throughout this paper we will write λ -calculus expressions in Lisp syntax. Our programs are all strongly typed. We use the Hindley-Milner polymorphic typing system [13] which is used in functional programming languages like OCaml and Haskell. We now define DSLs:

Definition: (\mathcal{D}, θ) . A DSL \mathcal{D} is a set of typed λ -calculus expressions. A weight vector θ for a DSL \mathcal{D} is a vector of $|\mathcal{D}| + 1$ real numbers: one number for each DSL element $e \in \mathcal{D}$, written θ_e and controlling the probability of e occurring in a program, and a weight controlling the probability of a variable occurring in a program, θ_{var} .

Together with its weight vector, a DSL defines a distribution over programs, $\mathbb{P}[p|\mathcal{D}, \theta]$. In the supplement, we define this distribution by specifying a procedure for drawing samples from $\mathbb{P}[p|\mathcal{D}, \theta]$. Care must be taken to ensure that programs are well-typed and that variable scoping rules are obeyed. With this distribution in hand, we search for programs by enumerating λ -calculus expressions in decreasing order of their probability under (\mathcal{D}, θ) .

Why enumerate, when the program synthesis community has invented many sophisticated algorithms that search for programs? [14, 15, 3, 16, 4]. We have two reasons: (1) A key point of our work is that learning the DSL, along with a neural recognition model, can make program induction tractable, even if the search algorithm is very simple. (2) Enumeration is a general approach that can be applied to any program induction problem. Many of these more sophisticated approaches require special conditions on the space of programs.

A drawback of using an enumerative search algorithm is that we have no efficient means of solving for arbitrary constants that might occur in the program. In Sec. 4, we will show how to find programs with real-valued constants by automatically differentiating through the program and setting the constants using gradient descent.

2.3 Compilation: Learning a Neural Recognition Model

The purpose of training the recognition model is to amortize the cost of searching for programs. It does this by learning to predict, for each task, programs with high likelihood according to $\mathbb{P}[x|p]$ while also being probable under the prior (\mathcal{D}, θ) . Concretely, the recognition model q predicts, for each task $x \in X$, a weight vector $q(x) = \theta^{(x)} \in \mathbb{R}^{|\mathcal{D}|+1}$. Together with the DSL, this defines a distribution over programs, $\mathbb{P}[p|\mathcal{D}, \theta = q(x)]$. We abbreviate this distribution as $q(p|x)$. The crucial aspect of this framing is that the neural network leverages the structure of the learned DSL, so it is *not* responsible for generating programs wholesale. We share this aspect with DeepCoder [8] and [17].

How should we get the data to train q ? This is nonobvious because ECC is only weakly supervised (i.e., learns only from tasks and not from (program, task) pairs). One approach is to sample programs from the DSL, run them to get their input/outputs, and then train q to predict the program from the input/outputs. This approach is like how a Helmholtz machine trains its recognition model during its “sleep” phase [18]. The advantage of “Helmholtz machine” training is that we can draw unlimited samples from the DSL, training on a large amount of data. Another approach is self-supervised learning, training q on the (program, task) pairs discovered by the algorithm so far. The advantage of self-supervised learning is that the training data is much higher quality, because we are training on the actual tasks. Due to these complementary advantages, we train on both these sources of data.

Formally, q should approximate the true posteriors over programs: minimizing the expected KL-divergence, $\mathbb{E}[\text{KL}(\mathbb{P}[p|x, \mathcal{D}, \theta] \| q(p|x))]$, equivalently maximizing $\mathbb{E}[\sum_p \mathbb{P}[p|x, \mathcal{D}, \theta] \log q(p|x)]$, where the expectation is taken over tasks. Taking this expectation over the empirical distribution of tasks gives self-supervised training; taking it over samples from the generative model gives Helmholtz-machine style training. The objective for a recognition model (\mathcal{L}_{RM}) combines the Helmholtz machine

Example programs in frontiers	Proposed subexpression
<pre>(lambda (a b) (foldr b (cons "," a) (lambda (x z) (cons x z)))) (lambda (a b) (foldr a b (lambda (x z) (cons x z))))</pre>	<pre>(foldr a b (lambda (x z) (cons x z)))</pre>

Figure 3: The DSL induction algorithm proposes subexpressions of programs to add to the DSL. These subexpressions are taken from programs in the frontiers (left column), and can introduce new variables (right column: a and b). Here, the proposed subexpression appends two lists.

160 (\mathcal{L}_{HM}) and self supervised (\mathcal{L}_{SS}) objectives, $\mathcal{L}_{\text{RM}} = \mathcal{L}_{\text{SS}} + \mathcal{L}_{\text{HM}}$:

$$\mathcal{L}_{\text{HM}} = \mathbb{E}_{(p,x) \sim (\mathcal{D}, \theta)} [\log q(p|x)] \quad \mathcal{L}_{\text{SS}} = \mathbb{E}_{x \sim X} \left[\sum_{p \in \mathcal{F}_x} \frac{\mathbb{P}[x, p | \mathcal{D}, \theta]}{\sum_{p' \in \mathcal{F}_x} \mathbb{P}[x, p' | \mathcal{D}, \theta]} \log q(p|x) \right]$$

161 Evaluating \mathcal{L}_{HM} involves sampling programs from the current DSL, running them to get their outputs,
 162 and then training q to regress from the input/outputs to the program. Since these programs map
 163 inputs to outputs, we need to sample the inputs as well. Our solution is to sample the inputs from the
 164 empirical observed distribution of inputs in X .

165 2.4 Compression: Learning a Generative Model (a DSL)

166 The purpose of the DSL is to offer a set of abstractions that allow an agent to easily express solutions
 167 to the tasks at hand. Intuitively, we want the algorithm to look at the frontiers and generalize beyond
 168 them, both so the DSL can better express the current solutions, and also so that the DSL might expose
 169 new abstractions which will later be used to discover more programs. Formally, we want the DSL
 170 maximizing $\int \mathcal{L} d\theta$ (Sec. 2.1). We replace this marginal with an AIC approximation, giving the
 171 following objective for DSL induction:

$$\log \mathbb{P}[\mathcal{D}] + \arg \max_{\theta} \sum_{x \in X} \log \sum_{p \in \mathcal{F}_x} \mathbb{P}[x|p] \mathbb{P}[p|\mathcal{D}, \theta] + \log \mathbb{P}[\theta|\mathcal{D}] - \|\theta\|_0 \quad (3)$$

172 We induce a DSL by searching locally through the space of DSLs, proposing small changes to \mathcal{D} until
 173 Eq. 3 fails to increase. The search moves work by introducing new λ -expressions into the DSL. We
 174 propose these new expressions by extracting subexpressions from programs already in the frontiers.
 175 These subexpressions are fragments of the original programs, and can introduce new variables
 176 (Fig. 3), which then become new functions in the DSL. The idea of storing and reusing fragments of
 177 expressions comes from Fragment Grammars [19] and Tree-Substitution Grammars [20].

178 To define the prior distribution over (\mathcal{D}, θ) , we penalize the syntactic complexity of the λ -calculus
 179 expressions in the DSL, defining $\mathbb{P}[\mathcal{D}] \propto \exp\left(-\lambda \sum_{p \in \mathcal{D}} \text{size}(p)\right)$ where $\text{size}(p)$ measures the size
 180 of the syntax tree of program p , and place a symmetric Dirichlet prior over the weight vector θ .

181 Putting all these ingredients together,
 182 Alg. 1 describes how we combine pro-
 183 gram search, recognition model train-
 184 ing, and DSL induction.

185 3 Sequence 186 manipulating programs

187 We apply ECC to list processing (Sec-
 188 tion 3.1) and text editing (Section 3.2).
 189 For both these domains we use a bidi-
 190 rectional GRU [21] for the recogni-
 191 tion model, and initially provide the
 192 system with a generic set of list pro-
 193 cessing primitives: foldr, unfold, if, map, length, index, =, +, -, 0, 1, cons, car, cdr, nil, and
 194 is-nil.

Algorithm 1 The ECC Algorithm

Input: Initial DSL \mathcal{D} , set of tasks X , iterations I
Hyperparameters: Enumeration timeout T
 Initialize $\theta \leftarrow \text{uniform}$
for $i = 1$ **to** I **do**
 $\mathcal{F}_x^\theta \leftarrow \{p | p \in \text{enum}(\mathcal{D}, \theta, T) \text{ if } \mathbb{P}[x|p] > 0\}$ (**Explore**)
 $q \leftarrow \text{train recognition model, maximizing } \mathcal{L}_{\text{RM}}$ (**Compile**)
 $\mathcal{F}_x^q \leftarrow \{p | p \in \text{enum}(\mathcal{D}, q(x), T) \text{ if } \mathbb{P}[x|p] > 0\}$ (**Explore**)
 $\mathcal{D}, \theta \leftarrow \text{induceDSL}(\{\mathcal{F}_x^\theta \cup \mathcal{F}_x^q\}_{x \in X})$ (**Compress**)
end for
return \mathcal{D}, θ, q

```

f0(a,b) = (foldr a b (lambda (x y) (cons x y))))
  (f0: Appends lists (of characters))
f1(s,c) = (foldr s s (lambda (x a) (cdr (if (= c x) s a))))
  (f1: Drop first characters from s until c reached)
f2(s) = (unfold s empty? car (lambda (z) (f1 z SPACE)))
  (f2: Abbreviates a sequence of words)
f3(s,c) = (foldr s s (lambda (x a) (if (= c x) nil (cons x a))))
  (f3: Take characters from s until c reached)

```

Table 1: Some string editing learned subroutines

Temple Annalisa Haven 185 → TAH1	Nancy FreeHafer → Dr. Nancy
Lara Gregori Bradford → LGB	Andrew Cencici → Dr. Andrew
$f(s) = (f_2 \ s)$	$f(s) = (f_0 \text{"Dr."} \ (f_3 \ s \ \text{" "}))$

Figure 4: Two string edit tasks (top) and the programs ECC writes for them (bottom). f_0 and f_2 are subroutines written by ECC, defined in Tbl. 1.

3.1 List Functions

Synthesizing programs that manipulate data structures is a widely studied problem in the programming languages community [3]. We consider this problem within the context of learning functions that manipulate lists. We created 244 Lisp-style list manipulation tasks, each with 15 input/output examples (Tbl. ??). Our data set is challenging along two dimensions: many of the functions are very complicated, and the agent must learn to solve these complicated problems from only 244 tasks. Our data set primarily consists of arithmetic operations upon sequences, and so, in addition to the Lisp primitives provided for the text editing experiments, we additionally start the system out with the following primitives: `mod`, `*`, `>`, `is-square`, `is-prime`, 2, 3, 4, 5.

A complete repertoire of higher-order functions is a staple of functional programming standard libraries. Although we provided ECC with some of the standard higher-order functions, like `foldr` and `unfold`, we did not include others. When trained on these list functions, our system rediscovers and then reuses the higher-order function `filter` **Lucas: not sure if it actually does this! It would be cool if we can write something like this though.**

3.2 String Editing

Synthesizing programs that manipulate strings is a classic problem in the programming languages and AI literatures [17, 22], and algorithms that learn string editing programs ship in Microsoft Excel [1]. This prior work presumes a ready-made DSL, expertly crafted to suit string editing. We show ECC can instead start out with generic Lisp primitives and recover many of the higher-level building blocks that have made these other system successful. An obstacle here, however, is that our enumerative search procedure has no means of generating string constants, and so we incorporate string-valued parameters as a primitive, defining $\mathbb{P}[x|p]$ by marginalizing out the values of the string via dynamic programming. In Section 4, we will use a similar trick to synthesize programs containing real numbers using gradient descent.

We automatically generated 109 string editing tasks (Fig. 4) and model strings as lists of characters. At first, ECC cannot find any correct programs for most of the tasks. It assembles a DSL (Tbl. 1) that lets it rapidly explore the space of programs and find solutions to all of the tasks.

How well does the learned DSL generalized to real text-editing scenarios? We tested, but did not train, our system on problems from the SyGuS [23] program synthesis competition. Before any learning, ECC solves 32/108 of the problems with an average search time of 11 minutes. After learning, it solves 80/108, and does so much faster, solving them in an average of 29 seconds. As of the 2017 SyGuS competition, the best-performing algorithm solves 86/108 problems, and does so with a *different hand-engineered DSL for each problem*. Here we learned a single DSL that applied generically to all of the tasks, and perform comparably to the best prior work.

229 4 Symbolic Regression: Programs from visual input

230 We apply ECC to symbolic regression problems. Here, the agent observes points along the curve of a
 231 function, and must write a program that fits those points. We initially equip our learner with addition,
 232 multiplication, and division, and task it with solving 100 symbolic regression problems, each either a
 233 polynomial of degree 1–4 or a rational function. The recognition model is a convolutional network
 234 that observes an image of the target function’s graph (Fig. 5) – visually, different kinds of polynomials
 235 and rational functions produce different kinds of graphs, and so the recognition model can learn to
 236 look at a graph and predict what kind of function best explains it. A key difficulty, however, is that
 237 these problems are best solved with programs containing real numbers. Our solution to this difficulty
 238 is to allow the system to write programs with real-valued parameters, and then fit those parameters by
 239 automatically differentiating through the programs the system writes and use gradient descent to fit
 240 the parameters. We define the likelihood model, $\mathbb{P}[x|p]$, by assuming a Gaussian noise model for the
 241 input/output examples, and penalize the use of real-valued parameters using the BIC [24].

242 ECC learns a DSL containing templates for polynomials of different
 243 orders, as well as ratios of polynomials (Fig. 6). The algorithm also
 244 discovers programs that minimize the number of continuous degrees
 245 of freedom. For example, it learns to represent linear functions with
 246 the program `(* real (+ x real))`, which has two continuous de-
 247 grees of freedom, and represents quartic functions using the invented
 248 DSL primitive f_6 in Tbl. 6 which has five continuous parameters.
 249 This phenomenon arises from our Bayesian framing – both the bias
 250 towards shorter programs and the likelihood model’s BIC penalty.

251 4.1 Quantitative Results

252 We compare with four baselines on held-out tasks:

253 **Ours (no NN)**, which lesions the recognition model.

254 **RF/DC**, which holds the generative model (\mathcal{D}, θ) fixed and learns
 255 a recognition model only from samples from the fixed generative
 256 model. This is equivalent to our algorithm with $\lambda = \infty$ (Sec. 2.4)
 257 and $\mathcal{L}_{RM} = \mathcal{L}_{HM}$ (Sec. 2.3). We call this baseline RF/DC because
 258 this setup is closest to how RobustFill [6] and DeepCoder [8] are trained. We can not compare directly
 259 with these systems, because they are engineered for one specific domain, and do not have publicly
 260 available code and datasets.

261 **PCFG**, which lesions the recognition model, learns θ , and fixes \mathcal{D} . This is equivalent to ECC with
 262 $q(x) = \theta$ and $\lambda = \infty$, and is like learning the parameters of a PCFG while not learning its structure.

263 **Enum**, which enumerates a frontier without any learning – equivalently, our first exploration cycle.

264
 265

 266 $f_0(x) = (+ \ x \ \text{real})$
 267 $f_1(x) = (f_0 \ (* \ \text{real} \ x))$
 268 $f_2(x) = (f_1 \ (* \ x \ (f_0 \ x))$
 269 $\quad (f_2: \text{quadratics})$
 270 $f_3(x) = (/ \ (f_2 \ x) \ (f_0 \ x))$
 271 $\quad (f_3: \text{ratio of polynomials})$
 272

273 Figure 6: Some learned subrou-
 274 tines for symbolic regression. Sys-
 275 tem starts with addition, multipli-
 276 cation, division, and real numbers,
 277 and learns to build rational func-
 278 tions and polynomials up to 4th or-
 279 der.

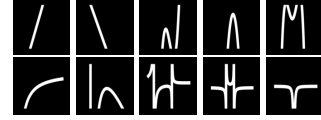


Figure 5: Recognition model input for symbolic regression. While the DSL learns subrou-
 tines for rational functions &
 polynomials, the recognition
 model jointly learns to look at
 a graph of the function (above)
 and predict which of those sub-
 routines is appropriate for ex-
 plaining the observation.

For each domain, we are interested both in how many tasks
 the agent can solve and how quickly it can find those solu-
 tions. Tbl. 2 compares our model against these baselines. Our
 full model consistently improves on the baselines, sometimes
 dramatically (string editing and symbolic regression). The
 recognition model consistently increases the number of solved
 held-out tasks, and lesioning it also slows down the conver-
 gence of the algorithm, taking more iterations to reach a given
 number of tasks solved (Fig. 7). This supports a view of the
 recognition model as a way of amortizing the cost of searching
 for programs.

5 Related Work

Our work is far from the first for learning to learn programs, an
 idea that goes back to Solomonoff [25]:

Deep learning: Much recent work in the ML community has
 focused on creating neural networks that regress from input/out-
 put examples to programs [6, 26, 17, 8]. These neural networks

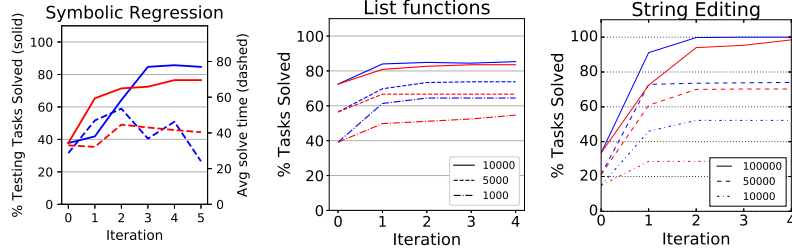


Figure 7: Learning curves for ECC both with (blue) and without (red) the recognition model as the frontier size is varied (solid/dashed/dotted lines).

are typically trained with strong supervision (i.e., with annotated ground-truth programs) on massive data sets (i.e., hundreds of millions [6]). Our work considers a weakly-supervised regime where ground truth programs are not provided and the agent must learn from a few hundred tasks.

Inventing new subroutines for program induction: Several program induction algorithms, most prominently the EC algorithm [12], take as their goal to learn new, reusable subroutines that are shared in a multitask setting. We find this work inspiring and motivating, and extend it along two dimensions: (1) we propose a new algorithm for inducing reusable subroutines, based on Fragment Grammars [19]; and (2) we show how to combine these techniques with bottom-up neural recognition models. Other instances of this related idea are [27], Schmidhuber’s OOPS model [28], and predicate invention in ILP [29].

Our work is an instance of Bayesian Program Learning (BPL; see [30, 12, 31, 27]). Previous BPL systems have largely assumed a fixed DSL (but see [27]), and our contribution here is a general way of doing BPL with less hand-engineering of the DSL.

	Ours	Ours (no NN)	RF/DC	PCFG	Enum
<i>List functions</i>					
% solved	86%	84%	60%	74%	70%
Solve time	0.8s	0.7s	1.0s	1.0s	1.1s
<i>String Editing</i>					
% solved	75%	%	33%	0%	30%
Solve time	29s	s	80s	–	
<i>Symbolic Regression</i>					
% solved	84%	75%	38%	38%	37%
Solve time	24s	40s	31s	55s	29s

Table 2: % solved w/ 5 sec timeout. Solve time: averaged over solved tasks. RF/DC: trained like RobustFill/DeepCoder. PCFG: model w/o structure learning. Enum: model w/o any learning.

6 Contribution and Outlook

We contribute an algorithm, ECC, that learns to program by bootstrapping a DSL with new domain-specific primitives that the algorithm itself discovers, together with a neural recognition model that learns how to efficiently deploy the DSL on new tasks. We believe this integration of top-down symbolic representations and bottom-up neural networks – both of them learned – could help make program induction systems more generally useful for AI. Many directions remain open. Two immediate goals are to integrate more sophisticated neural recognition models [6] and program synthesizers [14], which may improve performance in some domains over the generic methods used here. Another direction is to explore DSL meta-learning: Can we find a *single* universal primitive set that could effectively bootstrap DSLs for new domains, including the four domains considered, but also many others?

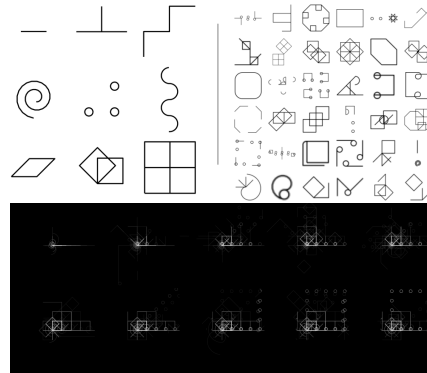


Figure 8: Future: dreams. Top left: hand crafted training targets. Top right: examples of discovered compiled new programs. Bottom: compiled program across iterations to highlight structure emergence.

References

- [1] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, volume 46, pages 317–330. ACM, 2011.
- [2] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Joshua B Tenenbaum. Learning to infer graphics programs from hand-drawn images. *arXiv preprint arXiv:1707.09627*, 2017.
- [3] John K Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *PLDI*, 2015.
- [4] Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. *ACM SIGPLAN Notices*, 50(10):107–126, 2015.
- [5] Stephen H Muggleton, Dianhuan Lin, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. *Machine Learning*, 100(1):49–73, 2015.
- [6] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. *arXiv preprint arXiv:1703.07469*, 2017.
- [7] Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-guided deductive search for real-time program synthesis from examples. *ICLR*, 2018.
- [8] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *ICLR*, 2016.
- [9] Tuan Anh Le, Atılım Güneş Baydin, and Frank Wood. Inference Compilation and Universal Probabilistic Programming. In *AISTATS*, 2017.
- [10] Geoffrey E Hinton, Peter Dayan, Brendan J Frey, and Radford M Neal. The "wake-sleep" algorithm for unsupervised neural networks. *Science*, 268(5214):1158–1161, 1995.
- [11] David D. Thornburg. Friends of the turtle. *Compute!*, March 1983.
- [12] Eyal Dechter, Jon Malmaud, Ryan P. Adams, and Joshua B. Tenenbaum. Bootstrap learning via modular concept discovery. In *IJCAI*, 2013.
- [13] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.
- [14] Armando Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, 2008.
- [15] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 305–316. ACM, 2013.
- [16] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *ACM SIGPLAN Notices*, volume 50, pages 619–630. ACM, 2015.

- 372 [17] Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lamp-
373 son, and Adam Kalai. A machine learning framework for
374 programming by example. In *ICML*, pages 187–195, 2013.
- 375 [18] Peter Dayan, Geoffrey E Hinton, Radford M Neal, and
376 Richard S Zemel. The helmholtz machine. *Neural compu-
377 tation*, 7(5):889–904, 1995.
- 378 [19] Timothy J. O’Donnell. *Productivity and Reuse in Lan-
379 guage: A Theory of Linguistic Computation and Storage*.
380 The MIT Press, 2015.
- 381 [20] Trevor Cohn, Phil Blunsom, and Sharon Goldwater. Induc-
382 ing tree-substitution grammars. *JMLR*.
- 383 [21] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre,
384 Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and
385 Yoshua Bengio. Learning phrase representations using rnn
386 encoder-decoder for statistical machine translation. *arXiv
387 preprint arXiv:1406.1078*, 2014.
- 388 [22] Tessa Lau. *Programming by demonstration: a machine
389 learning approach*. PhD thesis, 2001.
- 390 [23] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando
391 Solar-Lezama. Sygus-comp 2016: results and analysis.
392 *arXiv preprint arXiv:1611.07627*, 2016.
- 393 [24] Christopher M. Bishop. *Pattern Recognition and Machine
394 Learning*. 2006.
- 395 [25] Ray J Solomonoff. A system for incremental learning
396 based on algorithmic probability. Sixth Israeli Conference
397 on Artificial Intelligence, Computer Vision and Pattern
398 Recognition, 1989.
- 399 [26] Jacob Devlin, Rudy R Bunel, Rishabh Singh, Matthew
400 Hausknecht, and Pushmeet Kohli. Neural program meta-
401 induction. In *NIPS*, 2017.
- 402 [27] Percy Liang, Michael I. Jordan, and Dan Klein. Learning
403 programs: A hierarchical bayesian approach. In *ICML*,
404 2010.
- 405 [28] Jürgen Schmidhuber. Optimal ordered problem solver. *Ma-
406 chine Learning*, 54(3):211–254, 2004.
- 407 [29] Dianhuan Lin, Eyal Dechter, Kevin Ellis, Joshua B. Tenen-
408 baum, and Stephen Muggleton. Bias reformulation for
409 one-shot function induction. In *ECAI 2014*, 2014.
- 410 [30] Brenden M Lake, Ruslan Salakhutdinov, and Joshua B
411 Tenenbaum. Human-level concept learning through proba-
412 bilistic program induction. *Science*, 350(6266):1332–1338,
413 2015.
- 414 [31] Kevin Ellis, Armando Solar-Lezama, and Josh Tenenbaum.
415 Sampling for bayesian program learning. In *Advances in
416 Neural Information Processing Systems*, 2016.