Learning Libraries of Subroutines for Neurally–Guided Bayesian Program Learning

Anonymous Author(s)
Affiliation
Address
`email`

Abstract

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

1 Introduction

Imagine that you are asked to edit some text, and told that you should change the text "Nancy FreeHafer" to "Dr. Nancy". From this example, you likely infer that "Pushmeet Kohli" should be changed to "Dr. Pushmeet", drawing upon your prior knowledge of text, like that words are separated by spaces and that one commonly prepends titles like "Dr." In this work, we consider the problem of building agents that solve few-shot learning tasks like these, and also the problem of acquiring the prior knowledge necessary to quickly solve these tasks (Figure 1). We think of solutions to these tasks as being best represented by programs, and so our problem can be stated as follows: how should an agent learn to write programs? We take inspiration from two sources: (1) Good software engineers compose libraries of reusable subroutines that are shared across related programming tasks. Returning to Figure 1, a good string processing library should support appending strings and splitting on spaces – exactly the prior knowledge needed to solve the task in Figure 1. (2) Skilled human programmers can quickly recognize what kinds of programming idioms and library routines would be useful for solving the task at hand, even if they cannot instantly work out the details. We combine these two

TASK	Nancy FreeHafer $\longrightarrow$ Dr. Nancy Pushmeet Kohli $\longrightarrow$???
PROGRAM	f(`s`) = (f_0 `"Dr." (`f_2 `s " "))`
LIBRARY (DSL)	f_0(`a`,`b`) = `(fold a b (lambda (x y) (cons x y)))` (f_0: *Appends lists (of characters)*) f_1(`s`,`c`) = `(fold s s (lambda (x a) (if (= c x) nil (cons x a))))` (f_1: *Take characters from* `s` *until* `c` *reached*)

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.

ideas into a new algorithm called ECC, which takes as input a collection of programming **tasks**, and then jointly solves three problems: (1) Writing programs that solve the tasks; (2) Composing a

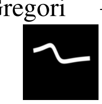
Domain	Example Task	Part of the learned DSL
Lists	`[7 0 2]` ⟶ `[7 0 2 4]` `[3 9]` ⟶ `[3 9 4]`	`(foldr nil (lambda (a b) (cons a b)))` *(appends lists)*
Strings	Temple Anna H ⟶ TAH Lara Gregori ⟶ LG	`(map (lambda (x) (if (= x a) b x)))` *(replace occurrences of a w/ b)*
Regression		`(+ (* real x) real)`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

library of domain-specific subroutines – which allow the agent to more compactly write programs in the domain, and (3) Training a neural network to recognize which library components are useful for which kinds of tasks. Together, the library and neural net encode the domain specific knowledge needed to quickly write programs.

We call the learned library of subroutines a **Domain-Specific Language** (**DSL**), a term widely used in the program synthesis community [14]. Prior work on program learning has largely assumed a fixed, hand-engineered DSL, both in classic symbolic program learning approaches (e.g., Metagol: [1], FlashFill: [2]), neural approaches (e.g., RobustFill: [3]), and hybrids of neural and symbolic methods (e.g., Neural-guided deductive search: [4], DeepCoder: [5]). The contribution of this work is an algorithm for learning DSLs while jointly training a neural net to search for programs in the DSL.

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

Our algorithm is called **Explore/Compress/Compile** (ECC), because it iterates between three different steps: an **Explore** step uses the DSL to explore the space of programs, searching for ones that solve the tasks; a **Compress** step modifies the structure of the DSL by discovering regularities across programs found by the previous Explore step; and a **Compile** step, which improves the program search procedure by training a neural network to write programs in the current DSL, in the spirit of "amortized" or "compiled" inference [6]. We call the neural net a **recognition model** (c.f. Hinton 1995 [7]). The learned DSL distills commonalities across programs that solve tasks, helping the agent solve related program induction problems. The neural recognition model ensures that searching for programs remains tractable even as the DSL (and hence the search space for programs) expands.

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

2 The ECC Algorithm

Our goal is to induce a DSL while finding programs solving each of the tasks. We take inspiration primarily from the Exploration-Compression algorithm for bootstrap learning [8]. Exploration-Compression alternates between exploring the space of solutions to a set of tasks, and compressing those solutions to suggest new search primitives for the next exploration stage. We extend these ideas into an inference strategy that iterates through three steps: an **Explore** step uses the current DSL and recognition model to search for programs that solve the tasks. The **Compress** and **Compile** steps update the DSL and the recognition model, respectively. Crucially, these steps synergistically bootstrap off each other:

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

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

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

2.1 Hierarchical Bayesian Framing

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

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

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

The above equations summarize the problem from the point of view of an ideal Bayesian learner. However, Eq. 1 is wildly intractable because evaluating $J(\mathcal{D}, \theta)$ involves summing over the infinite set of all programs. In practice we will only ever be able to sum over a finite set of programs. So, for each task, we define a finite set of programs, called a *frontier*, and only marginalize over the frontiers:

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 $p \in \mathcal{F}_x$.

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

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

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

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

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

Searching for programs is hard because of the large combinatorial search space. We ease this difficulty by training a neural recognition model, $q(\cdot|\cdot)$, during the compilation phase: q is trained to 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 the cost of finding programs with high posterior probability.

Neural recognition model: tractably maxing $\mathscr{L}$ w.r.t. the frontiers. Here we train a neural network, q, to predict a distribution over programs conditioned on a task. The objective of q is to assign high probability to programs p where $\mathbb{P}[x, p|\mathcal{D}, \theta]$ is large, because including those programs in the frontiers will most increase $\mathscr{L}$.

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

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

programs that closely resembles the Lisp programming language. λ-calculus includes 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 [9] which is used in functional programming languages like OCaml and Haskell. We now define DSLs:

Definition: $(\mathcal{D}, \theta)$**.** 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 primitive $e \in \mathcal{D}$, written θ_e and controlling the probability of primitive 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 $(\mathcal{D}, \theta)$.

Why enumerate, when the program synthesis community has invented many sophisticated algorithms that search for programs? [10, 11, 12, 13, 14]. 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 the recognition model is to amortize the cost of searching for programs. It does this by learning to predict programs which are probable under $(\mathcal{D}, \theta)$ while also assigning high likelihood to a task according to $\mathbb{P}[x|p]$. Concretely, the recognition model q is a neural network that 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 [5] and [15].

We want a recognition model that closely approximates the true posteriors over programs. We formulate this as minimizing the expected KL-divergence, $\mathbb{E}\left[\text{KL}\left(\mathbb{P}[p|x, \mathcal{D}, \theta] \| q(p|x)\right)\right]$, or equivalently maximizing $\mathbb{E}\left[\sum_p \mathbb{P}[p|x, \mathcal{D}, \theta] \log q(p|x)\right]$, where the expectation is taken over tasks. One could take this expectation over the empirical distribution of tasks, like how an autoencoder is trained [16]; or, one could take this expectation over samples from the generative model, like how a Helmholtz machine is trained [17]. We found it useful to maximize both an autoencoder-style objective $\mathcal{L}_{\text{AE}}$ and a Helmholtz-style objective $\mathcal{L}_{\text{HM}}$, giving the objective for a recognition model, $\mathcal{L}_{\text{RM}} = \mathcal{L}_{\text{AE}} + \mathcal{L}_{\text{HM}}$:

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

The $\mathcal{L}_{\text{HM}}$ objective is essential for data efficiency: all of our experiments train ECC on only a few hundred tasks, which is too little for a high-capacity neural network q. Once we bootstrap a $(\mathcal{D}, \theta)$, we can draw unlimited samples from $(\mathcal{D}, \theta)$ and train q on those samples.

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

Example programs in frontiers	Proposed subexpression
`(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))))`	`(foldr a b (lambda (x z) (cons x z)))`

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.

2.4 Compression: Learning a Generative Model (a DSL)

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

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

We induce a DSL by searching locally through the space of DSLs, proposing small changes to $\mathcal{D}$ until Eq. 3 fails to increase. The search moves work by introducing new λ-expressions into the DSL. We propose these new expressions by extracting subexpressions from programs already in the frontiers. These subexpressions are fragments of the original programs, and can introduce new variables (Fig. 3), which then become new functions in the DSL. The idea of storing and reusing fragments of expressions comes from Fragment Grammars [18] and Tree-Substitution Grammars [19].

To define the prior distribution over $(\mathcal{D}, \theta)$, we penalize the syntactic complexity of the λ-calculus 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 of the syntax tree of program p, and place a symmetric Dirichlet prior over the weight vector θ.

Putting all these ingredients together, Alg. 1 describes how we combine program search, recognition model training, and DSL induction.

Algorithm 1 The ECC Algorithm

Input: Initial DSL $\mathcal{D}$, set of tasks X, iterations I
Hyperparameters: Enumeration timeout T
Initialize $\theta \leftarrow$ 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)**
  $q \leftarrow$ 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 $\mathcal{D}, \theta, q$

3 Sequence manipulating programs

We apply ECC to list processing (Section 3.1) and text editing (Section 3.2). For both these domains we use a bidirectional GRU [21] for the recognition model, and initially provide the system with a generic set of list processing primitives: `foldr`, `unfold`, `if`, `map`, `length`, `index`, `=`, `+`, `-`, `0`, `1`, `cons`, `car`, `cdr`, `nil`, and `is-nil`.

3.1 List Functions

Synthesizing programs that manipulate data structures is a widely studied problem in the programming languages community [12]. 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

f_0(a,b) = `(foldr a b (lambda (x y) (cons x y)))))`
(f_0: *Appends lists (of characters))*
f_1(s,c) = `(foldr s s (lambda (x a) (cdr (if (= c x) s a))))`
(f_1: *Drop first characters from* `s` *until* `c` *reached*)
f_2(s) = `(unfold s empty? car (lambda (z) (`f_1` z SPACE)))`
(f_2: *Abbreviates a sequence of words*)
f_3(s,c) = `(foldr s s (lambda (x a) (if (= c x) nil (cons x a))))`
(f_3: *Take characters from* `s` *until* `c` *reached)*

Table 1: Some string editing learned subroutines

Temple Annalisa Haven 185 ⟶ TAH1 Lara Gregori Bradford ⟶ LGB	Nancy FreeHafer ⟶ Dr. Nancy Andrew Cencici ⟶ Dr. Andrew
f(s) =`(`f_2` s)`	f(s) =`(`f_0` "Dr. " (`f_3` s " "))`

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.

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 [15, 22], and algorithms that learn string editing programs ship in Microsoft Excel [2]. 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.

4 Symbolic Regression: Programs from visual input

We apply ECC to symbolic regression problems. Here, the agent observes points along the curve of a function, and must write a program that fits those points. We initially equip our learner with addition, multiplication, and division, and task it with solving 100 symbolic regression problems, each either a polynomial of degree 1–4 or a rational function. The recognition model is a convolutional network that observes an image of the target function's graph (Fig. 5) – visually, different kinds of polynomials

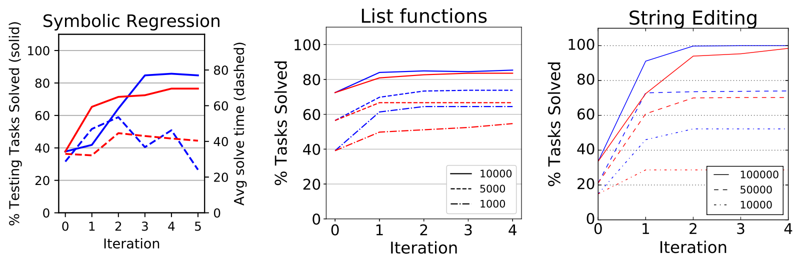


Figure 7: Learning curves for ECC both with (blue) and without (red) the recognition model as the frontier size is varied (solid/dashed/dotted lines).

and rational functions produce different kinds of graphs, and so the recognition model can learn to look at a graph and predict what kind of function best explains it. A key difficulty, however, is that these problems are best solved with programs containing real numbers. Our solution to this difficulty is to allow the system to write programs with real-valued parameters, and then fit those parameters by automatically differentiating through the programs the system writes and use gradient descent to fit the parameters. We define the likelihood model, $\mathbb{P}[x|p]$, by assuming a Gaussian noise model for the input/output examples, and penalize the use of real-valued parameters using the BIC [24].

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

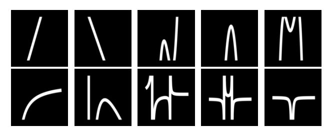


Figure 5: Recognition model input for symbolic regression. While the DSL learns subroutines for rational functions & polynomials, the recognition model jointly learns to look at a graph of the function (above) and predict which of those subroutines is appropriate for explaining the observation.

4.1 Quantitative Results

We compare with four baselines on held-out tasks:
Ours (no NN), which lesions the recognition model.
RF/DC, which holds the generative model $(\mathcal{D}, \theta)$ fixed and learns a recognition model only from samples from the fixed generative model. This is equivalent to our algorithm with $\lambda = \infty$ (Sec. 2.4) and $\mathcal{L}_{\text{RM}} = \mathcal{L}_{\text{HM}}$ (Sec. 2.3). We call this baseline RF/DC because this setup is closest to how RobustFill [3] and DeepCoder [5] are trained. We can not compare directly with these systems, because they are engineered for one specific domain, and do not have publicly available code and datasets.
PCFG, which lesions the recognition model, learns θ, and fixes $\mathcal{D}$. This is equivalent to ECC with $q(x) = \theta$ and $\lambda = \infty$, and is like learning the parameters of a PCFG while not learning its structure.
Enum, which enumerates a frontier without any learning – equivalently, our first exploration cycle.

$f_0(\texttt{x}) = \texttt{(+ x real)}$
$f_1(\texttt{x}) = \texttt{(}f_0\ \texttt{(* real x))}$
$f_2(\texttt{x}) = \texttt{(}f_1\ \texttt{(* x (}f_0\ \texttt{x)))}$
(f_2: quadratics)
$f_3(\texttt{x}) = \texttt{(/ (}f_2\ \texttt{x) (}f_0\ \texttt{x))}$
(f_3: ratio of polynomials)

Figure 6: Some learned subroutines for symbolic regression. System starts with addition, multiplication, division, and real numbers, and learns to build rational functions and polynomials up to 4th order.

For each domain, we are interested both in how many tasks the agent can solve and how quickly it can find those solutions. 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 convergence 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/output examples to programs [3, 26, 15, 5]. These neural networks are typically trained with strong supervision (i.e., with annotated ground-truth programs) on massive data sets (i.e., hundreds of millions [3]). 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 [8], 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 [18]; 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].

	Ours	Ours (no NN)	RF/DC	PCFG	Enum
		List functions			
% solved	**86%**	84%	60%	74%	70%
Solve time	0.8s	0.7s	1.0s	1.0s	1.1s
		String Editing			
% solved	**75%**	%	33%	0%	30%
Solve time	29s	s	80s	–	
		Symbolic Regression			
% 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.

Our work is an instance of Bayesian Program Learning (BPL; see [30, 8, 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.

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 [3] and program synthesizers [10], 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?

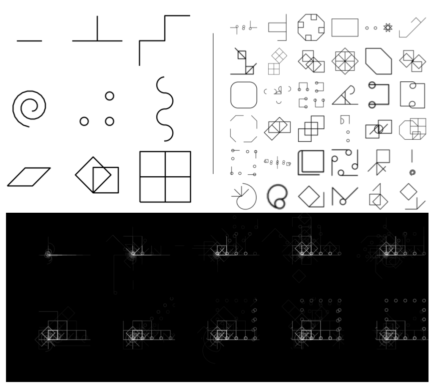


Figure 8: Future: dreams. Top left: hand crafted traning targets. Top right: examples of discovered compiled new programs. Bottom: compiled programm across iterations to highlight structure emergence.

References

[1] 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.

[2] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, volume 46, pages 317–330. ACM, 2011.

[3] 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.

[4] 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.

[5] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *ICLR*, 2016.

[6] Tuan Anh Le, Atılım Güneş Baydin, and Frank Wood. Inference Compilation and Universal Probabilistic Programming. In *AISTATS*, 2017.

[7] 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.

[8] Eyal Dechter, Jon Malmaud, Ryan P. Adams, and Joshua B. Tenenbaum. Bootstrap learning via modular concept discovery. In *IJCAI*, 2013.

[9] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[10] Armando Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, 2008.

[11] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 305–316. ACM, 2013.

[12] John K Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *PLDI*, 2015.

[13] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *ACM SIGPLAN Notices*, volume 50, pages 619–630. ACM, 2015.

[14] Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. *ACM SIGPLAN Notices*, 50(10):107–126, 2015.

[15] Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. A machine learning framework for programming by example. In *ICML*, pages 187–195, 2013.

[16] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 2006.

[17] Peter Dayan, Geoffrey E Hinton, Radford M Neal, and Richard S Zemel. The helmholtz machine. *Neural computation*, 7(5):889–904, 1995.

[18] Timothy J. O'Donnell. *Productivity and Reuse in Language: A Theory of Linguistic Computation and Storage*. The MIT Press, 2015.

[19] Trevor Cohn, Phil Blunsom, and Sharon Goldwater. Inducing tree-substitution grammars. *JMLR*.

[20] J.D. Lafferty. *A Derivation of the Inside-outside Algorithm from the EM Algorithm*. Research report.

[21] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[22] Tessa Lau. *Programming by demonstration: a machine learning approach*. PhD thesis, 2001.

[23] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Sygus-comp 2016: results and analysis. *arXiv preprint arXiv:1611.07627*, 2016.

[24] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. 2006.

[25] Ray J Solomonoff. A system for incremental learning based on algorithmic probability. Sixth Israeli Conference on Artificial Intelligence, Computer Vision and Pattern Recognition, 1989.

[26] Jacob Devlin, Rudy R Bunel, Rishabh Singh, Matthew Hausknecht, and Pushmeet Kohli. Neural program meta-induction. In *NIPS*, 2017.

[27] Percy Liang, Michael I. Jordan, and Dan Klein. Learning programs: A hierarchical bayesian approach. In *ICML*, 2010.

[28] Jürgen Schmidhuber. Optimal ordered problem solver. *Machine Learning*, 54(3):211–254, 2004.

[29] Dianhuan Lin, Eyal Dechter, Kevin Ellis, Joshua B. Tenenbaum, and Stephen Muggleton. Bias reformulation for one-shot function induction. In *ECAI 2014*, 2014.

[30] Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

[31] Kevin Ellis, Armando Solar-Lezama, and Josh Tenenbaum. Sampling for bayesian program learning. In *Advances in Neural Information Processing Systems*, 2016.