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 CoCoSea 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 text, 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 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 "Jane Goodall" goes to "Dr. Jane", drawing upon your prior knowledge of text, like that words are separated by spaces. Few-shot learning problems like these are commonplace in both human and machine learning. We cast these few-shot learning problems as each being a program synthesis problem. The programming languages and AI communities have built many successful program synthesis algorithms, spanning text editing (e.g., FlashFill [1]), motor programs [30], procedural graphics [2], planning procedures [26], and many others. However, the success of these systems hinges upon a carefully hand-engineered **Domain Specific Language (DSL)**. DSLs impart prior knowledge of a domain by providing a restricted set of finely-tuned programming primitives: for text editing, these are primitives like appending and splitting on characters. In this work, we consider the problem of building agents that solve program learning tasks, and also the problem of acquiring the prior knowledge necessary to quickly solve these tasks (Figure 1). Our solution is an algorithm that learns a DSL while jointly training a neural network to help write programs in the learned DSL.

TASK	Nancy FreeHafer $\longrightarrow$ Dr. Nancy Jane Goodall $\longrightarrow$???
PROGRAM	f(`s`) = `(`f_0 `"Dr." (`f_1 `s ' '))`
DSL	f_0(`a`,`b`) = `(foldr a b (`λ `(x y) (cons x y)))` (f_0: *Appends lists (of characters)*) f_1(`s`,`c`) = `(foldr s s (`λ `(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 **program**s, 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.

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 text editing 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 weave these ideas together into an algorithm called CoCoSea (**Co**mpress/**Co**mpile/**Sea**rch), which takes as input a collection of programming **tasks**, and then jointly solves three problems: (1) Searching for programs that solve the tasks; (2) Composing a library (DSL) 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 DSL components are useful for which kinds of tasks. Together, the DSL and neural net encode the domain specific knowledge needed to quickly write programs.

CoCoSea iterates through three different steps: a **Search** step uses the DSL and neural network 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 during search; and a **Compile** step, which improves the search procedure by training a neural network to write programs in the current DSL, in the spirit of "amortized" or "compiled" inference [9]. We call the neural net a **recognition model** (c.f. Hinton 1995 [10]). The learned DSL distills commonalities across programs that solve tasks, helping the agent solve related programming 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 think of CoCoSea 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 search by training a neural network to assist the search procedure.

We apply CoCoSea to three domains: list processing; text editing (in the style of FlashFill [1]); and symbolic regression. For each of these we initially provide a generic set of programming primitives. Our algorithm then constructs its own DSL for expressing solutions in the domain (Tbl. 1).

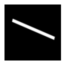
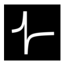
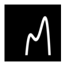
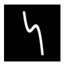
List Functions	Text Editing	Symbolic Regression
[7 0 2]→[7 0 2 9] [9 3]→[9 3 9] $f(\ell)$ =(f_1 ℓ 9) [7 3]→False [3]→False [9 0 0]→True [0]→True [0 7 3]→True $f(\ell)$ =(f_3 ℓ 0)	+106 769-438→106.769.438 +83 973-831→83.973.831 f(s) =(f_0 "." "," (f_0 "." " " (cdr s)))	 f(x) =(f_1 x)  f(x) =(f_6 x)
[2 7 8 1]→8 [3 19 14]→19 $f(\ell)$ =(f_2 ℓ)	Temple Anna H →TAH Lara Gregori→LG f(s) =(f_2 s)	 f(x) =(f_4 x)  f(x) =(f_3 x)
f_0(ℓ,r) =(foldr r ℓ cons) (f_0: *Append lists* r *and* ℓ) f_1(ℓ,k) =(f_0 (cons k nil) ℓ) (f_1: *Append the number* k *to* ℓ) f_2(ℓ) =(foldr ℓ 0 (λ (x a) (if (> a x) a x))) (f_2: *Maximum element in list* ℓ) f_3(ℓ,k) =(foldr ℓ (is-nil ℓ) (λ (x a) (if a a (= k x)))) (f_2: *Whether* ℓ *contains* k)	f_0(s,a,b) =(map (λ (x) (if (= x a) b x)) s) (f_0: *Performs character substitution*) f_1(s,c) =(foldr s s (λ (x a) (cdr (if (= c x) s a)))) (f_1: *Drop characters from* s *until* c *reached*) f_2(s) =(unfold s is-nil car (λ (z) (f_1 z " "))) (f_2: *Abbreviates a sequence of words*)	f_0(x) =(+ x real) f_1(x) =(f_0 (* real x)) f_2(x) =(f_1 (* x (f_0 x))) f_3(x) =(f_0 (* x (f_2 x))) f_4(x) =(f_0 (* x (f_3 x))) (f_4: *4th order polynomial*) f_5(x) =(/ real x) f_6(x) =(f_4 (f_0 x)) (f_6: *rational function*)

Table 1: Top: Tasks from each domain, each followed by the programs CoCoSea discovers for them. Bottom: Several examples from learned DSL. Notice that learned DSL primitives can call each other.

Prior work on program learning has largely assumed a fixed, hand-engineered DSL, both in classic symbolic program learning approaches (e.g., Metagol: [5], FlashFill: [1]), neural approaches (e.g., RobustFill: [6]), and hybrids of neural and symbolic methods (e.g., Neural-guided deductive search: [7], DeepCoder: [8]). A notable exception is the EC algorithm [12], which also learns a library of subroutines. We were inspired by EC, and go beyond it by giving a new algorithm that learns DSLs, as well as a way of combining DSL learning with neurally guided program search. In the experiments section, we compared directly with EC, showing empirical improvements.

The new contribution of this work is thus an algorithm for learning DSLs which jointly trains a neural net to search for programs in the DSL.

2 The CoCoSea 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 [12]. 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: a **Search** 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 bootstrap each other:
Search: Solving tasks. 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 search 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 search. 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

CoCoSea 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.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 $\mathscr{L}$:

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

CoCoSea does approximate MAP inference by maximizing this lower bound on the joint probability, alternating maximization w.r.t. the frontiers (Search) 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 $\mathbb{P}[x|p]\mathbb{P}[p|\mathcal{D}, \theta]$ 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 $q(p|x)$ to assign high probability to programs p where $\mathbb{P}[x|p]\mathbb{P}[p|\mathcal{D},\theta]$ is large, because including those programs in the frontiers will most increase $\mathscr{L}$.

2.2 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: $(\mathcal{D},\theta)$**.** 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 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 $\mathbb{P}[p|\mathcal{D},\theta]$.

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.

However, a drawback of enumerative search is that we have no efficient means of solving for arbitrary constants that might occur in a 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 $(\mathcal{D},\theta)$. 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 we are considering a weakly supervised setting (i.e., learning 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 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 search. 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, $\mathbb{E}\left[\text{KL}\left(\mathbb{P}[p|x,\mathcal{D},\theta]\|q(p|x)\right)\right]$, 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 ($\mathcal{L}_{\text{RM}}$) combines the Helmholtz machine

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

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

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:

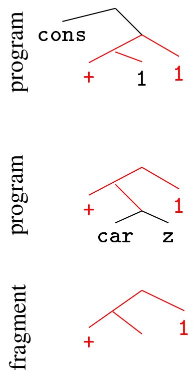
$$\log \mathbb{P}[\mathcal{D}] + \arg\max_{\theta} \sum_{x\in X} \log \sum_{p\in\mathcal{F}_x} \mathbb{P}[x|p]\mathbb{P}[p|\mathcal{D},\theta] + \log \mathbb{P}[\theta|\mathcal{D}] - \|\theta\|_0 \tag{3}$$

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 fragments of programs already in the frontiers (Tbl. 2). An important point here is that we are *not* simply adding subexpressions of programs to $\mathcal{D}$, as done in the EC algorithm [12] and other prior work [29]. Instead, we are extracting fragments that unify with programs in the frontiers. This idea of storing and reusing fragments of expressions comes from Fragment Grammars [19] and Tree-Substitution Grammars [20], and is closely related to the idea of antiunification [**?**].

Algorithm 1 The COCOSEA 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
    $\mathcal{F}_x^\theta \leftarrow \{p | p \in \text{enum}(\mathcal{D},\theta,T) \text{ if } \mathbb{P}[x|p] > 0\}$   (Search)
    $q \leftarrow$ train recognition model, maximizing $\mathcal{L}_{\text{RM}}$   (Compile)
    $\mathcal{F}_x^q \leftarrow \{p | p \in \text{enum}(\mathcal{D},q(x),T) \text{ if } \mathbb{P}[x|p] > 0\}$   (Search)
    $\mathcal{D},\theta \leftarrow$induceDSL($\{\mathcal{F}_x^\theta \cup \mathcal{F}_x^q\}_{x\in X}$)   (Compress)
end for
return $\mathcal{D},\theta,q$
```

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



Example programs in frontiers	Proposed λ-expression
`(λ (ℓ) (map (λ (x) (index x ℓ)) (range (- (length ℓ) 1))))` `(λ (ℓ) (map (λ (x) (index x ℓ)) (range (+ 1 1))))`	`(map (λ (x) (index x ℓ)) (range α))`
`(λ (s) (map (λ (x) (if (= x '.') '-' x))) s)` `(λ (s) (map (λ (x) (if (= x '-') ',' x))) s)`	`(λ (s) (map (λ (x) (if (= x α) β x))) s)`

Figure 2: **Left:** syntax trees of two programs sharing common structure, highlighted in red, from which we extract a fragment and add it to the DSL (bottom). **Right:** actual programs, from which we extract fragments that (top) slice from the beginning of a list or (bottom) perform character substitutions.

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

3 Programs that manipulate sequences

We apply CoCoSea 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 Processing

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 with some involvement of integer arithmetic.

We created 236 human-interpretable list manipulation tasks, each with 15 input/output examples (Tbl. 2). Our data set is interesting in three major ways: many of the tasks require complex solutions; the tasks were not generated from some latent DSL, and the agent must learn to solve these complicated problems from only 236 tasks. Our data set assumes arithmetic operations as well as sequence operations, so we additionally provide our system with the following arithmetic primitives: `mod`, `*`, `>`, `is-square`, `is-prime`.

Name	Input	Output
repeat-2	[7 0]	[7 0 7 0]
drop-3	[0 3 8 6 4]	[6 4]
rotate-2	[8 14 1 9]	[1 9 8 14]
count-head-in-tail	[1 2 1 1 3]	2
keep-mod-5	[5 9 14 6 3 0]	[5 0]
product	[7 1 6 2]	84

Table 2: Some tasks in our list function domain. See the supplement for the complete data set.

We evaluated CoCoSea on random 50/50 test/train split. Interestingly, we found that the recognition model provided little benefit for the training tasks. However, it yielded faster search times on held out tasks, allowing more tasks to be solved before timing out. The system composed 38 new subroutines, yielding a more expressive DSL more closely matching the domain (left of Tbl. 1, right of Fig. 2). See the supplement for a complete list of DSL primitives discovered by CoCoSea.

3.2 Text Editing

Synthesizing programs that edit text is a classic problem in the programming languages and AI literatures [17, 22], and algorithms that learn text editing programs ship in Microsoft Excel [1]. This prior work presumes a hand-engineered DSL. We show CoCoSea can instead start out with generic sequence manipulation primitives and recover many of the higher-level building blocks that have made these other text editing systems successful.

Because our enumerative search procedure has no means of generating string constants, we have the enumerator propose programs with string-valued holes – e.g., in Fig. 1, we enumerate `(`f_0` string (`f_2` s ' '))` – and define $\mathbb{P}[x|p]$ by marginalizing out the values of the strings via dynamic programming. In Section 4, we will use a similar trick to synthesize programs containing real numbers, but using gradient descent instead of dynamic programming.

We automatically generated 109 text editing tasks with 4 input/output examples each At first, CoCoSea cannot find any correct programs for most of the tasks. After three iterations, it assembles a DSL (Fig. 1 & center of Tbl. 1) that lets it rapidly explore the space of programs and find solutions to all of the tasks. The learned DSL contains 10 new functions, listed in the supplement.

How well does the learned DSL generalized to real text-editing scenarios? We tested, but did not train, our system on the 108 text editing problems from the SyGuS [23] program synthesis competition. Before any learning, CoCoSea solves 3.7% of the problems with an average search time of 235 seconds. After learning, it solves 74.1%, and does so much faster, solving them in an average of 38 seconds. As of the 2017 SyGuS competition, the best-performing algorithm solves 79.6% of the problems. But, SyGuS comes with a different hand-engineered DSL *for each text editing 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 CoCoSea 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. 3) – visually, different kinds of polynomials 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].

CoCoSea learns a DSL containing 13 new functions, most of which are templates for polynomials of different orders or ratios of polynomials. It also learns to find 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_4 in the rightmost column of Fig. 1 which has five continuous parameters. This phenomenon arises from our Bayesian framing – both the implicit bias towards shorter programs and the likelihood model's BIC penalty.

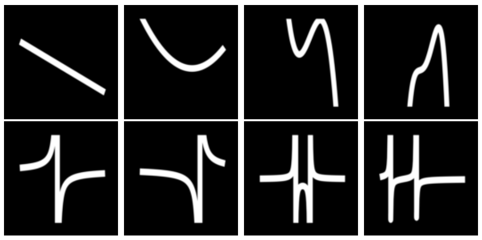


Figure 3: Recognition model input for symbolic regression. DSL learns subroutines for polynomials (top row) and polynomials (bottom row) while the recognition model jointly learns to look at a graph of the function (above) and predict which of those subroutines best explains the observation.

5 Quantitative Results

We compare with ablations of our model on held out tasks. The purpose of this ablation study is both to examine the role of each component of CoCoSea, as well as to compare with prior approaches in the literature: A head-to-head comparison of program synthesizers is complicated by the fact that each system, including ours, makes idiosyncratic assumptions about the space of programs and the statement of tasks. Nevertheless, much prior work can be modeled as special cases of our setup. We compare with the following baselines:
No NN, which lesions the recognition model.
NPS, which does not learn the DSL, instead learning the recognition model from samples drawn from the fixed DSL. We call this NPS (Neural Program Synthesis) because this setup is closest to how RobustFill [6] and DeepCoder [8] are trained.
SE, which lesions the recognition model and restricts the DSL learning algorithm to only add subexpressions of programs in the frontiers to the DSL. This is how prior approaches have learned libraries of functions [12, 27, 29].
Enum, which enumerates a frontier without any learning – equivalently, our first search step.

For each domain, we are interested both in how many tasks the agent can solve and how quickly it can find those solutions. Tbl. 3 compares our model against these alternatives. Our full model consistently improves on the baselines, sometimes dramatically (text 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. 4). This supports a view of the recognition model as a way of amortizing the cost of searching for programs.

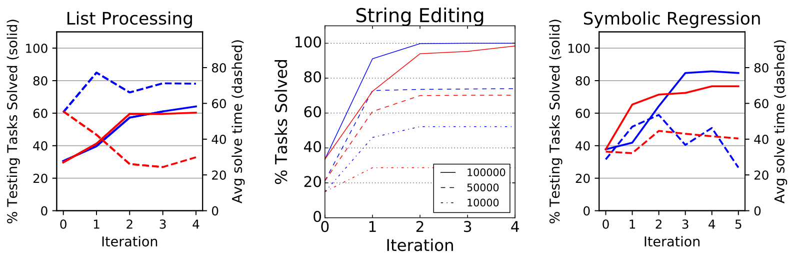


Figure 4: Learning curves for CoCoSea both with (blue) and without (red) the recognition model. Solid lines: % holdout testing tasks solved. Dashed lines: Average solve time.

6 Related Work

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

	Ours	No NN	SE	NPS	PCFG	Enum
List Processing						
% solved	**79%**	76%	71%	35%	62%	37%
Solve time	4.1s	5.8s	10.6s	34.7s	43.4s	20.2s
Text Editing						
% solved	**74%**	43%	30%	33%	0%	4%
Solve time	38s	65s	38s	80s	–	235s
Symbolic Regression						
% solved	**84%**	75%	62%	38%	38%	37%
Solve time	**24**s	40s	28s	31s	55s	29s

Table 3: % solved before timeout. Solve time: averaged over solved tasks. SE: NPS: trained like RobustFill/DeepCoder. PCFG: model w/o structure learning. Enum: model w/o any learning.

Deep learning: Much recent work in the ML community has focused on creating neural networks that regress from input/output examples to programs [6, 26, 17, 8]. CoCoSea's recognition model draws heavily from this line of work, particularly from [17]. We see these prior works as operating in a different regime: typically, they train 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 at most a few hundred tasks, which is facilitated by our "Helmholtz machine" style recognition model training (Sec. 2.3).

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 Inductive Logic Programming [29].

Bayesian Program Learning: 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.

7 Contribution and Outlook

We contribute an algorithm, CoCoSea, 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 three domains considered, but also many others?

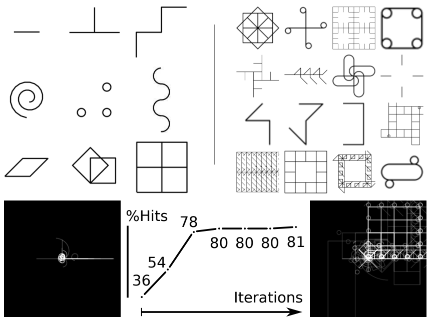


Figure 5: Top left: a few example of the shapes used as targets. Top right: a few examples of discovered new shapes. Bottom: learning curve across iterations in percent in the middle, and on both side average of the compiled new shapes before any traning and at the end of the last iteration. See supplementary material for more details.

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.

[17] 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.

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

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

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

[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.