Learning Reusable Components 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 solve symbolic regression problems, edit strings, and synthesize functions on lists, showing how the model learns a domain-specific library of program components for expressing solutions to problems in the domain.

1 Introduction

Automatically inducing programs from examples is a long-standing goal of artificial intelligence. Recent work has successfully used symbolic search techniques (e.g., Metagol: [1], FlashFill: [2]), neural networks trained from a corpus of examples (e.g., RobustFill: [3]), and hybrids of neural and symbolic methods (e.g., Neural-guided deductive search: [4], DeepCoder: [5]) to synthesize programs for task domains such as string transformations, list processing, and robot navigation and planning. However, all these approaches – symbolic, neural and neural-symbolic – rely upon a hand-engineered *Domain-Specific Language* (DSL). DSLs contain an inventory of restricted programming primitives, encoding domain-specific knowledge about the space of programs. In practice we often have only a few input/output examples for each program to be induced, and thus success often hinges on having a good DSL that provides a crucial inductive bias for what would otherwise be an unconstrained search through the space of all computable functions. Here we ask, to what extent can we dispense with such highly hand-engineered domain-specific languages?

We propose *learning* the DSL by inducing a library of domain–specific subroutines. We consider the setting where we have a collection of related programming tasks, each specified by a set of input/output examples. Starting from a weaker or more general library of primitives, we give an algorithm for constructing a richer, more powerful, and better-tuned DSL. Our algorithm is called **Explore/Compress/Compile** (ECC) because it iterates between three different phases: an **Explore** phase uses the DSL to explore the space of programs, searching for ones that solve the tasks; a **Compress** phase modifies the DSL by discovering regularities in the programs found by the previous Explore phase; and a **Compile** phase, 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 [7]. We call the neural net a **recognition model** (c.f. Hinton 1995 [6]). 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.

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

Submitted to 32nd Conference on Neural Information Processing Systems (NIPS 2018). Do not distribute.

solve the kinds of problems that humans can solve relatively quickly – once they acquire the relevant domain expertise. These correspond to short programs – once you have the right DSL. Even with the right DSL, program search may be intractable, so we amortize the cost of program search by training a neural network to assist the search procedure.

We apply ECC to three domains: symbolic regression; FlashFill-style [2] string processing; and Lisp-style functions on lists. For each of these we provide a generic set of programming primitives in a Lisp-like language, including conditionals, variables, and higher-order functions. Our algorithm then discovers its own domain-specific vocabulary for expressing solutions in the domain (Tbl. 1).

Domain	Part of the learned DSL
Regression	`(+ (* real x) real)` *(a linear function of x)*
Strings	`(join ” (split c s))` *(delete occurrences of c in string s)*
Lists	`(map (lambda (x) (+ x k)) l)` *(add k to every element of list l)*

Figure 1: Examples of structure found in DSLs learned by our algorithm. ECC builds a new DSL by discovering and reusing useful subroutines.

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** cycle uses the current DSL and recognition model to search for programs that solve the tasks. The **Compress** and **Compile** cycles update the DSL and the recognition model, respectively. Crucially, these steps bootstrap off each other (Fig. 2):

Exploration: Searching for programs. Our program search is informed by both the DSL and the recognition model. When these improve, we find more programs solving the 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 richer 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 both more data to train on, and better data.

Sec. 2.1 frames this 3-step procedure as probabilistic inference. Sec. 2.2 explains how we search for programs that solve the tasks; Sec. 2.3 explains how we train a neural network to search for programs; and Sec. 2.4 explains how we induce a DSL from programs.

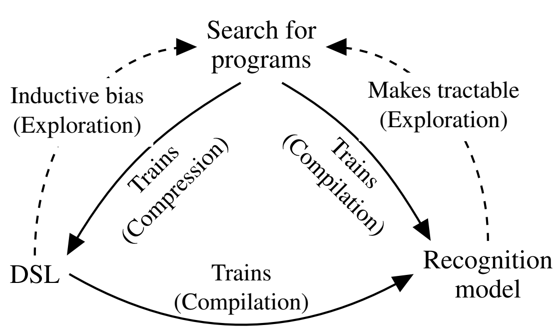

Figure 2: ECC solves for programs, the DSL, and a neural network called the *recognition model*. Each of these steps bootstrap off of the others in an iterative inference algorithm.

2.1 Probabilistic Framing

ECC takes as input a set of *tasks*, written X, each of which is a program induction problem. It has at its disposal a *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:

$$J(\mathcal{D},\theta) \triangleq \mathbb{P}[\mathcal{D},\theta] \prod_{x\in X} \sum_{p} \mathbb{P}[x|p]\mathbb{P}[p|\mathcal{D},\theta], \quad \mathcal{D}^* = \arg\max_{\mathcal{D}} \int J(\mathcal{D},\theta)\; \mathrm{d}\theta, \quad \theta^* = \arg\max_{\theta} J(\mathcal{D}^*,\theta) \tag{1}$$

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, $q(p|x) \propto \mathbb{P}[x,p|\mathcal{D},\theta]$, 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 that solve the tasks. In this work we use the simple search strategy of enumerating programs from the DSL in decreasing order of their probability, and then checking if an enumerated 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 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 [11] which is used in functional programming languages like OCaml. Type variables are always written using lowercase Greek letters and we write $\alpha \to \beta$ to mean a function that takes an input of type α and returns something of type β. We use the notation $p : \tau$ to mean that the λ-calculus expression p has the type τ. For example, to describe the type of the identity function we would say `(lambda (x) x)`: $\alpha \to \alpha$. We say a type α *unifies* with τ if every expression $p : \alpha$ also satisfies $p : \tau$. Furthermore, the act of *unifying* a type α with τ is to introduce constraints on the type variables of α to ensure that α unifies with τ. See Supplement for more detail on program representation. With this notation in hand 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 a weight controlling the probability of a variable occurring in a program, θ_{var}.

Algorithm 1 Generative model over programs

function sampleProgramFromDSL($\mathcal{D}, \theta, \tau$):
Input: DSL $\mathcal{D}$, weight vector θ, type τ
Output: a program whose type unifies with τ
return sample($\mathcal{D}, \theta, \varnothing, \tau$)

function sample($\mathcal{D}, \theta, \mathcal{E}, \tau$):
Input: DSL $\mathcal{D}$, weight vector θ, environment $\mathcal{E}$, type τ
Output: a program whose type unifies with τ
if $\tau = \alpha \to \beta$ **then**
  var $\leftarrow$ an unused variable name
  body $\sim$ sample($\mathcal{D}, \theta, \{\text{var} : \alpha\} \cup \mathcal{E}, \beta$)
  return `(lambda (var) body)`
end if
primitives $\leftarrow \{p | p : \tau' \in \mathcal{D} \cup \mathcal{E}$
  if τ can unify with yield(τ')$\}$
Draw $e \sim$ primitives, w.p. $\propto \theta_e$ if $e \in \mathcal{D}$
  w.p. $\propto \frac{\theta_{var}}{|\text{variables}|}$ if $e \in \mathcal{E}$
Unify τ with yield(τ').
$\{\alpha_k\}_{k=1}^{K} \leftarrow$ argTypes(τ')
for $k = 1$ **to** K **do**
  $a_k \sim$ sample($\mathcal{D}, \theta, \mathcal{E}, \alpha_k$)
end for
return $(e\ a_1\ a_2\ \cdots\ a_K)$
where:

$$\text{yield}(\tau) = \begin{cases} \text{yield}(\beta) & \text{if } \tau = \alpha \to \beta \\ \tau & \text{otherwise.} \end{cases}$$

$$\text{argTypes}(\tau) = \begin{cases} [\alpha] + \text{argTypes}(\beta) & \text{if } \tau = \alpha \to \beta \\ [] & \text{otherwise.} \end{cases}$$

Alg. 1 is a procedure for drawing samples from the generative model $(\mathcal{D}, \theta)$. In practice, we enumerate programs rather than sampling them. Enumeration proceeds by a depth-first search over the random choices made by Alg. 1; we wrap the depth-first search in iterative deepening to build λ-calculus expressions in order of their probability.

Why enumerate, when the program synthesis community has invented many sophisticated algorithms that search for programs? [12, 13, 14, 15, 16]. 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. 3.1, 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 accelerate the search for programs. It does this by learning to predict programs which are probable under $(\mathcal{D}, \theta)$ while also assigning high likelihood for 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 [22].

We want a recognition model that closely approximates the true posteriors over programs. We formulate this as minimizing the expected KL-divergence, $\mathbb{E}\left[\mathrm{KL}\left(\mathbb{P}[p|x,\mathcal{D},\theta]\|q(p|x)\right)\right]$, 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 observed empirical distribution of tasks, like how an autoencoder is trained [17]; or, one could take this expectation over samples from the generative model, like how a Helmholtz machine is trained [18]. 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, $\mathcal{L}_{\text{RM}} = \mathcal{L}_{\text{AE}} + \mathcal{L}_{\text{HM}}$:

$$\mathcal{L}_{\text{HM}} = \mathbb{E}_{p\sim(\mathcal{D},\theta)}\left[\log q(p|x)\right] \text{ s.t. } x = [\![p]\!] \quad \mathcal{L}_{\text{AE}} = \mathbb{E}_{x\sim X}\left[\sum_{p\in\mathcal{F}_x}\frac{\mathbb{P}\left[x,p|\mathcal{D},\theta\right]}{\sum_{p'\in\mathcal{F}_x}\mathbb{P}\left[x,p'|\mathcal{D},\theta\right]}\log q(p|x)\right]$$

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.

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. In the ECC algorithm we infer the DSL from a collection of frontiers. 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.

Recall from Sec. 2.1 that we want the DSL maximizing $\int \mathscr{L}\,\mathrm{d}\theta$. 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 [19] and Tree-Substitution Grammars [20].

We define a prior distribution over DSLs which penalizes the sizes of the λ-calculus expressions in the DSL, and put a Dirichlet prior over the weight vector:

$$\mathbb{P}[\mathcal{D}] \propto \exp\left(-\lambda\sum_{p\in\mathcal{D}}\text{size}(p)\right) \qquad \mathbb{P}[\theta|\mathcal{D}] = \text{Dir}(\theta|\alpha) \tag{4}$$

where $\text{size}(p)$ measures the size of the syntax tree of program p, λ is a hyperparameter that acts as a regularizer on the size of the DSL, and α is a concentration parameter controlling the smoothness of the prior over θ.

To appropriately score each proposed $\mathcal{D}$ we must reestimate the weight vector θ. Although this problem may seem very similar to estimating the parameters of a probabilistic context free grammar (PCFG), for which we have effective approaches like the Inside/Outside algorithm [**?**], a DSL in ECC may be context-sensitive due to the presence of variables in the programs and also due to the polymorphic typing system. In the Supplement we derive a tractable MAP estimator for θ.

Example programs in frontiers	Proposed subexpression
`(lambda (a b) (fold b (cons "," a) (lambda (x z) (cons x z))))` `(lambda (a b) (fold a b (lambda (x z) (cons x z))))`	`(fold 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.

Algorithm 2 The ECC Algorithm

Input: Initial DSL $\mathcal{D}$, set of tasks X, iterations I
Hyperparameters: Enumeration timeout T
Output: DSL $\mathcal{D}$, weight vector θ, recognition model $q(\cdot)$
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**)
  $\mathcal{D}, \theta \leftarrow$induceDSL$(\{\mathcal{F}_x^\theta \cup \mathcal{F}_x^q\}_{x \in X})$ (**Compress**)
end for
return $\mathcal{D}, \theta, q$

2.5 Implementing ECC

Alg. 2 describes how we combine program search, recognition model training, and DSL induction. We note the following implementation details: (1) We perform an exploration cycle before each compression and compilation cycle. (2) On the first iteration, we do *not* train the recognition model on samples from the generative model because a generative model has not yet been learned – we instead train the network to only maximize $\mathcal{L}_{\text{AE}}$. (3) During both DSL induction and neural net training, we calculate Eq. 3 and $\mathcal{L}_{\text{RM}}$ by only summing over the top K programs in $\mathcal{F}_x$ as measured by $\mathbb{P}[x, p|\mathcal{D}, \theta]$ – we found that $K = 2$ sufficed. (4) For added robustness, we enumerate programs from both the generative model $(\mathcal{D}, \theta)$ and the recognition model.

3 Experiments

3.1 Symbolic Regression

We show how to use ECC to infer programs containing both discrete structure and continuous parameters. The high-level idea is to write programs with unspecified real-valued parameters, and to fit those parameters by differentiating through the program and optimizing using gradient descent. We task our model with solving 200 symbolic regression problems, each either a polynomial of degree 1 to 4 or a rational function. The recognition model is a convolutional network that observes an image of the target function's graph (Fig. 4) – 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.

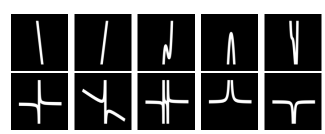

Figure 4: Recognition model input for symbolic regression. While the DSL learns subroutines for rational functions & polynomials, the recognition model jointly learns to look at a graph of the function (above) and predict which of those subroutines is appropriate for explaining the observation.

We initially give our system addition, multiplication, and division, along with the possibility of introducing real-valued parameters, which we write as $\mathcal{R}$. We define the likelihood of an observation x by assuming a Gaussian noise model for the input/output examples, and penalize the use of continuous parameters using the BIC [21].

ECC learns a DSL containing templates for polynomials of different orders, which lets the algorithm quickly find the functions that are most appropriate to this domain (Tbl. 1). We also found that the algorithm discovers programs that minimize the number of continuous degrees of freedom. For example, it represents the linear function $-x + 2$ with the program `(* (+ x`

f_0	= `(+ ℛ)`
f_1(`x`)	= `(* (* (+ x ℛ) ℛ) x)`
f_2(`x`)	= `(+ ℛ (`f_1` x))`
f_3(`x`)	= `(* x (`f_2` x))`
f_4(`x`)	= `(`f_0` (`f_3` x))`
f_5(`x`)	= `(* x (`f_4` x))`
f_6(`x`)	= `(`f_0` (`f_4` x))` (*4th order polynomial*)

Table 1: Some learned DSL primitives for symbolic regression. The system starts with addition, multiplication, and real numbers, and learns to build polynomials up to 4th order.

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 (a x) (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*)

Table 2: Some learned DSL primitives for string editing

ℛ) ℛ) which has two continuous degrees of freedom, and represents quartic functions using the invented DSL primitive f_6 in Tbl. 1 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.

3.2 Programs that manipulate sequences

We apply ECC to text editing (Section 3.2.1) and list processing (Section 3.2.2). For both these domains we initially provide the system with a generic minimal set of programming primitives, and use a bidirectional GRU [**?**] for the recognition model. The initial set of primitives includes routines commonly found in Lisp or Scheme interpreters: `fold`, `unfold`, `if`, `map`, `length`, `=`, `+`, `-`, `0`, `1`, `cons`, `car`, `cdr`, `nil`, and `is-nil`.

3.2.1 String Editing

Synthesizing programs that manipulate strings is a classic problem in the programming languages and AI literatures [22, 23], and algorithms that learn string editing programs ship in Microsoft Excel [2]. However, 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. We automatically generated 600 string editing tasks with 4 input/output examples each (Fig. 5). At first, ECC cannot find any correct programs for most of the tasks. It assembles a DSL (Tbl. 2) that lets it rapidly explore the space of programs and find solutions to all of the tasks.

3.2.2 List Functions

Synthesizing programs that manipulate data structures is a widely studied problem in the programming languages community [14]. We consider this problem within the context of learning functions that

Input	Output	Input 1	Input 2	Output
Temple Annalisa Haven 185	TAH1	Launa	Withers	Launa Withers
Lara Gregori Bradford	LGB	Rudolf	Akiyama	Rudolf Akiyama
f(`s`) = `(`f_2` s)`		f(`a,b`) = `(`f_0` a (cons " " b))`		

Figure 5: 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. 2.

Name	Input	Output
append-4	[7 0 2]	[7 0 2 4]
len	[3 5 12 1]	4
has-2	[4 5 7 4]	`false`
repeat-2	[7 0]	[7 0 7 0]
drop-3	[0 3 8 6 4]	[6 4]
count-head-in-tail	[1 2 1 1 3]	2
rotate-2	[8 14 1 9]	[1 9 8 14]
pow-3	[10 4 13]	[1000 64 2197]
keep-mod-5	[5 9 14 6 3 0]	[5 0]

Table 3: Sample tasks from our list function domain

$f_0(\mathtt{i}, \ell)$ = `(singleton (index i` ℓ`))`
(f_0: *Put the i-th index into a list*)
$f_1(\mathtt{i}, \ell)$ = `(++` ℓ `(`f_0 `i` ℓ`))`
(f_1: *Append the i-th index*)
$f_2(\mathtt{n}, \ell)$ = `(any (lambda (x) (= n x))` ℓ`)`
(f_1: *Whether n appears in the list*)
$f_3(\mathtt{i}, \ell)$ = `(index (negate i) (sort` ℓ`))`
(f_3: *Get the i-th largest number*)
$f_4(\mathtt{n}, \ell)$ = `(mapi (lambda (i x) (mod x n))` ℓ`)`
$f_5(\mathtt{k}, \ell)$ = `(mapi (lambda (i x) (+ x k))` ℓ`)`
$f_6(\mathtt{k}, \mathtt{n}, \ell)$ = `(`f_4 `n (`f_5 `k` ℓ`)`
(f_6: *Caesar shift of k in integers modulo n*)

Table 4: Some learned DSL primitives for list functions

manipulate lists. We created 225 Lisp-style list manipulation tasks, each with 15 input/output examples (Tbl. 3). 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 225 tasks.

We evaluate ECC starting from two different initial DSLs, *base* and *rich*. The base DSL starts with only low-level primitives such as `mapi`, `reducei`, and `if`, whereas the rich DSL includes some common structure that can be built from these such as `all`, `filter`, `slice`, and `index`. Both are capable of solving all tasks supplied by our dataset. See Supplement for details on these DSLs.

We found this domain difficult to tackle without starting from the rich DSL. Our algorithm, like human learners, requires a spectrum of problems ranging from easy to hard. When starting with the base DSL, we found that there were not enough steppingstones in the curriculum to get ECC off the ground.

3.3 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 our algorithm with $q(x) = \theta$ and $\lambda = \infty$, and is like learning the parameters of a PCFG while not learning any of the structure.
Enum, which does no learning and just enumerates a frontier. This is equivalent to our first wake cycle.

	Ours	Ours (no NN)	RF/DC	PCFG	Enum
		Boolean Circuits			
% solved	**84%**	76%	63%	62%	62%
Solve time	0.9s	1.1s	1.2s	1.3s	1.3s
		Symbolic Regression			
% solved	**98%**	94%	0.7%	0%	2%
Solve time	2.7s	2.8s	3.6s	–	2.1s
MDL (nats)	9.3	11.0	2.3	–	4.6
		String Editing			
% solved	**99%**	82%	9%	24%	18%
Solve time	0.9s	2.2s	2.7s	2.3s	2.4s
		List functions (base DSL)			
% solved	**52%**	**52%**	40%	44%	42%
Solve time	0.4s	0.7s	1.2s	0.9s	1.0s
		List functions (rich DSL)			
% solved	**86%**	84%	60%	74%	70%
Solve time	0.8s	0.7s	1.0s	1.0s	1.1s

Table 5: % 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. MDL: $-\mathbb{E}\left[\mathbb{P}[x|p]\right]$. For domains other than symbolic regression MDL is 0 nats. For symbolic regression MDL is a proxy for # of continuous parameters.

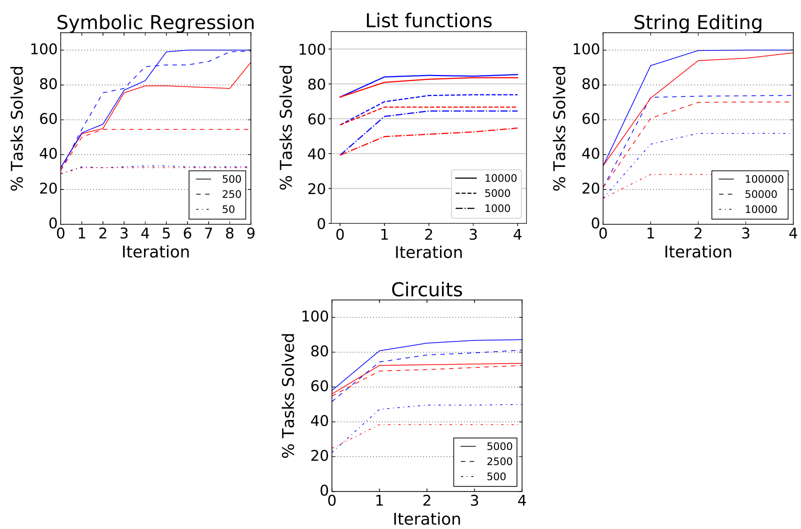

Figure 6: Learning curves for ECC both with (blue) and without (red) the recognition model as the frontier size is varied (solid/dashed/dotted lines).

For each domain, we are interested both in how many tasks the agent can solve and how quickly it can find those solutions. For symbolic regression, we also care about the quality of the solution, as measured by the likelihood model $\mathbb{P}[x|p]$, e.g. did the agent correctly explain a linear function using two numbers, or did it introduce extraneous parameters? Tbl. 5 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 wake/sleep cycles to reach a given number of tasks solved (Fig. 6). This supports the view of the recognition model as a way of accelerating the search over programs.

4 Related Work

Our work is far from the first for learning to learn programs, an idea that goes back to Solomonoff [24]:

Deep learning: Much recent work in the ML community has focused on creating neural networks that regress from input/output examples to programs [3, 25, 22, 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 [19]; and (2) we show how to combine these techniques with bottom-up neural recognition models. Other instances of this related idea are [26], Schmidhuber's OOPS model [27], and predicate invention in ILP [28].

Our work is an instance of Bayesian Program Learning (BPL; see [29, 8, 30, 26]). Previous BPL systems have largely assumed a fixed DSL (but see [26]), and our contribution here is a general way of doing BPL with less hand-engineering of the DSL.

5 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 [12], 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?

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] 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.

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

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

[9] Yadin Dudai, Avi Karni, and Jan Born. The consolidation and transformation of memory. *Neuron*, 88(1):20 – 32, 2015.

[10] Magdalena J Fosse, Roar Fosse, J Allan Hobson, and Robert J Stickgold. Dreaming and episodic memory: a functional dissociation? *Journal of cognitive neuroscience*, 15(1):1–9, 2003.

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

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

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

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

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

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

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

[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. *Journal of Machine Learning Research*, 11(Nov):3053–3096, 2010.

[21] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. 2006.

[22] 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.

[23] Tessa Lau. *Programming by demonstration: a machine learning approach*. PhD thesis, University of Washington, 2001.

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

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

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

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

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

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

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