Inducing Domain Specific Languages for Bayesian Program Learning

Anonymous Authors[1]

Abstract

This document provides a basic paper template and submission guidelines. Abstracts must be a single paragraph, ideally between 4–6 sentences long. Gross violations will trigger corrections at the camera-ready phase.

1. Introduction

Imagine an agent faced with a suite of new problems totally different from anything it has seen before. It has at its disposal a basic set of primitive actions it can compose to build solutions to these problems, but it is no idea what kinds of primitives are appropriate for which problems nor does it know the higher-level vocabulary in which solutions are best expressed. How can our agent get off the ground?

The AI and machine learning literature contains two broad takes on this problem. The first take is that the agent should come up with a better representation of the space of solutions, for example, by inventing new primitive actions: see *options* in reinforcement learning (Stolle & Precup, 2002), the EC algorithm in program synthesis (Dechter et al., 2013), or predicate invention in inductive logic programming (Muggleton et al., 2015). The second take is that the agent should learn a discriminative model mapping problems to a distribution over solutions: for example, policy gradient methods in reinforcement learning or neural models of program synthesis (Devlin et al., 2017; Balog et al., 2016). Our contribution is a general algorithm for fusing these two takes on the problem: we propose jointly inducing a representation language, called a *Domain Specific Language* (DSL), alongside a bottom-up discriminative model that regresses from problems to solutions. We evaluate our algorithm on four domains: building Boolean circuits; symbolic regression; FlashFill-style (Gulwani, 2011) string processing problems; and functions on lists. We show that HELMHOLTZHACKER can construct a set of basis primitives suitable for discovering solutions in each of these domains

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

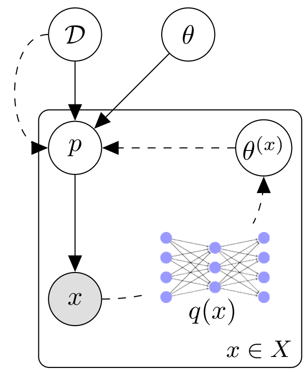


Figure 1: DSL $\mathcal{D}$ generates programs p by sampling DSL primitives with probabilities θ (Algorithm 1). We observe program outputs x. A neural network $q(\cdot)$ called the *recognition model* regresses from program outputs to a distribution over programs ($\theta^{(x)} = q(x)$). Solid arrows correspond to the top-down generative model. Dashed arrows correspond to the bottom-up recognition model.

We cast these problems as *Bayesian Program Learning* (BPL; see (Lake et al., 2013; Ellis et al., 2016; Liang et al., 2010)), where the goal is to infer from an observation x a posterior distribution over programs, $\mathbb{P}[p|x]$. A DSL $\mathcal{D}$ specifies the vocabulary in which programs p are written. We equip our DSLs with a *weight vector* θ; together, $(\mathcal{D}, \theta)$ define a probabilistic generative model over programs, $\mathbb{P}[p|\mathcal{D}, \theta]$. In this BPL setting, $\mathbb{P}[p|x] \propto \mathbb{P}[x|p]\mathbb{P}[p|\mathcal{D}, \theta]$, where the likelihood $\mathbb{P}[x|p]$ is domain-dependent. The solid lines in Fig. 1 the diagram this generative model. Alongside this generative model, we infer a bottom-up recognition model, $q(x)$, which is a neural network that regresses from observations to a distribution over programs.

Our key observation is that the generative and recognition models can bootstrap off of each other, greatly increasing the tractability of BPL.

2. The HELMHOLTZHACKER Algorithm

The goal of HELMHOLTZHACKER is to both induce a DSL and find good programs solving each of the tasks. Our strategy is to iterate through three steps: (1) searching for programs that solve the tasks, (2) learning a better neural recognition model – which we use to accelerate the search over programs – and (3) improving the DSL. The key observation here is that each of these three steps can bootstrap off of each other:

- Searching for programs: Program search uses a distribution determined by the neural recognition model – so the recognition model bootstraps the search process.
- Learning a recognition model: The recognition model is trained both on samples from the DSL and on programs found by the search procedure. As the DSL improves and we find more programs, the recognition model gets both more data to train on and better data.
- Improving the DSL: We induce a DSL from the programs we have found so far which solve the tasks; as we solve more tasks, we can hone in on richer DSLs that more closely match the domain.

Section 2.1 frames this 3-step procedure as a means of maximizing a lower bound on the posterior probability of the DSL given the tasks. Section 2.2 explains how we search for programs that solve the tasks; Section 2.3 explains how we train a neural network to accelerate the search over programs; and Section 2.4 explains how HELMHOLTZHACKER induces a DSL from programs.

2.1. Probabilistic Framing

HELMHOLTZHACKER 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 $\mathcal{D}$ that distills the commonalities across all of the programs that solve the tasks.

We frame this problem as maximum a posteriori (MAP) inference in the generative model diagrammed in Fig. 1. We wish to maximize the MAP probability of $\mathcal{D}$:

$$\mathbb{P}[\mathcal{D}|X] \propto \mathbb{P}[\mathcal{D}] \int \mathrm{d}\theta P(\theta|\mathcal{D}) \prod_{x\in X} \sum_p \mathbb{P}[x|p]\mathbb{P}[p|\mathcal{D},\theta]$$

In general this marginalization over θ is intractable, so we make an AIC-style approximation[1], $A \approx \log \mathbb{P}[\mathcal{D}|X]$:

$$A = \log \mathbb{P}[\mathcal{D}] + \arg\max_\theta \sum_{x\in X} \log \sum_p \mathbb{P}[x|p]\mathbb{P}[p|\mathcal{D},\theta] + \log P(\theta|\mathcal{D}) - ||\theta||_0 \qquad (1)$$

[1] Sec. 2.4 explains that $\mathcal{D}$ is a context-sensitive grammar. Conventional NLP approaches to using variational inference to lower bound the marginal over θ do not apply in our setting.

If we had a $(\mathcal{D},\theta)$ maximizing Eq. 1, then we could recover the most likely program for task x by maximizing $\mathbb{P}[x|p]\mathbb{P}[p|\mathcal{D},\theta]$. Through this lens we now take as our goal to maximize Eq. 1. But even *evaluating* Eq. 1 is intractable because it involves summing over the infinite set of all possible programs. In general, programs are hard-won: finding even a single program that explains a given observation presents a daunting combinatorial search problem. With this fact in mind, we will instead maximize the following tractable lower bound on Eq. 1, which we call J:

$$J = \log \mathbb{P}[\mathcal{D},\theta] + \sum_{x\in X} \log \sum_{p\in\mathcal{F}_x} \mathbb{P}[x|p]\mathbb{P}[p|\mathcal{D},\theta] \qquad (2)$$

This lower bound depends on sets of programs, $\{\mathcal{F}_x\}_{x\in X}$:

Definition. The *frontier of task* x, written $\mathcal{F}_x$, is a finite set of programs where $\mathbb{P}[x|p] > 0$ for all $p \in \mathcal{F}_x$.

We maximize J by alternatingly maximizing it w.r.t. the DSL and the frontiers:

Program Search: Maxing J w.r.t. the frontiers. Here we want to find new programs to add to the frontiers so that J increases the most. Adding new programs to the frontiers means searching for new programs p for task x where $\mathbb{P}[x,p|\mathcal{D},\theta]$ is large.

DSL Induction: Maxing J w.r.t. the DSL. Here $\{\mathcal{F}_x\}_{x\in X}$ is held fixed and so we can evaluate J. Now the problem is that of searching the discrete space of DSLs and finding one maximizing J.

Searching for programs is extremely difficult because of how large the search space is. We ease the difficulty of the search by learning a neural recognition model:

Neural recognition model: tractably maxing J 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. With q in hand we can find programs for frontier $\mathcal{F}_x$ by searching for programs maximizing $q(p|x)$. The network q exploits the structure of the DSL $\mathcal{D}$: rather than directly predicting a distribution over p conditioned on x, it predicts a weight vector, $\theta^{(x)}$, and we define $q(p|x) \triangleq \mathbb{P}[p|\mathcal{D},\theta = q(x)]$. This approach implements an amortized inference scheme (Ritchie et al., 2016) for the generative model in Fig. 1.

2.2. 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 include p in the frontier $\mathcal{F}_x$.

To make this concrete we need to define what programs

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 λvar. 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 $e \in \mathcal{E}$
Let $e : \alpha_1 \to \alpha_2 \to \cdots \to \alpha_K \to \beta$. Unify τ with β.
for $k = 1$ **to** K **do**
  $a_k \sim$ sample($\mathcal{D}, \theta, \mathcal{E}, \alpha_k$)
end for
return $e\ a_1\ a_2\ \cdots\ a_K$

function yield(τ):
Input: type τ
Output: the return type of τ
if $\tau = \alpha \to \beta$ **then return** yield(β) **else return** τ

actually are and what form $\mathbb{P}[p|\mathcal{D}, \theta]$ takes. In this work, we represent programs as *polymorphicly-typed λ-calculus expressions*. λ-calculus is a formalism for expressing functional programs. It includes variables, function application, and the ability to create new functions using ...

TODO: summarize lambda calculus and types in one paragraph

Definition: $\mathcal{D}$**.** A DSL $\mathcal{D}$ is a set of typed λ-calculus expressions.

Definition: θ**.** A weight vector θ for a DSL $\mathcal{D}$ is a vector of $|\mathcal{D}| + 1$ real numbers: one number for each DSL primitive $e : \tau \in \mathcal{D}$, written θ_e, and a weight controlling the probability of a variable occurring in a program, written θ_{var}.

Algorithm 1 is a procedure for drawing samples from $\mathbb{P}[p|\mathcal{D}, \theta]$. In practice, we enumerate programs rather than sampling them. Enumeration proceeds by a depth-first search over the random choices made by Algorithm 1; we wrap the depth-first search in iterative deepening to (approximately) 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? (Solar Lezama, 2008; Schkufza et al., 2013; Feser et al., 2015; Osera & Zdancewic, 2015; Polozov & Gulwani, 2015). We have two reasons:

- 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.
- 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 of programs.

A main drawback of an enumerative search algorithm is that we have no efficient means of solving for arbitrary constants that might occur in the program. In Section 3.2, we will show how to find programs with real-valued constants by automatically differentiating through the program and setting the constants using gradient descent. In Section 3.3 we will show that the bottom-up neural recognition model can learn which discrete constants should be included in a program.

2.3. Learning a Neural Recognition Model

The purpose of the recognition model is to accelerate the search over programs. It does this by learning to predict which programs both assign high likelihood to a task, and at the same time have high prior probability under the prior $\mathbb{P}[\cdot|\mathcal{D}, \theta]$.

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 can leverage the structure of the DSL, and is *not* responsible for generating programs wholesale. We will show that this lets us get away with a simple, low-capacity neural network.

We want a recognition model that closely approximates the true posteriors over programs, and so aim to minimize the following KL-divergence:

$$\mathbb{E}\left[\text{KL}\left(\mathbb{P}[p|x, \mathcal{D}, \theta] || q(p|x)\right)\right]$$

which is equivalent to 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 the observations, like how an autoencoder is trained (**?**); or, one could take this expectation over samples from the generative model, like how a Helmholtz machine is trained (?). We found it useful to maximize both an autoencoder-style objective (written $\mathcal{L}_{\text{AE}}$) and a Helmholtz-style objective ($\mathcal{L}_{\text{HM}}$), giving the HELMHOLTZHACKER objective for a recognition model, $\mathcal{L}_{\text{RM}}$:

$$\mathcal{L}_{\text{RM}} = \mathcal{L}_{\text{AE}} + \mathcal{L}_{\text{HM}} \tag{3}$$
$$\mathcal{L}_{\text{HM}} = \mathbb{E}_{p\sim(\mathcal{D},\theta)}\left[\log q(p|x)\right],\ p \text{ evaluates to } x$$
$$\mathcal{L}_{\text{AE}} = \mathbb{E}_{x\sim X}\left[\sum_{p\in\mathcal{F}_x} \frac{\mathbb{P}\left[x,p|\mathcal{D},\theta\right]}{\sum_{p'\in\mathcal{F}_x}\mathbb{P}\left[x,p|\mathcal{D},\theta\right]}\log q(p|x)\right]$$

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 outputs to the program. If these programs map inputs to outputs, then we need some way of sampling these inputs as well. Our solution to this problem is to sample the inputs from the empirical observed distribution of inputs in X.

2.4. Inducing the DSL from the Frontiers

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 HELMHOLTZHACKER algorithm we infer the DSL from a collection of frontiers. Intuitively, we want the algorithm to look at the programs in the frontiers and generalize beyond them; not only so the DSL can better express the current solutions, but also so that the DSL might expose new abstractions which will later used to discover even better programs.

Exact maximization of J (Eq.2) w.r.t. $(\mathcal{D},\theta)$ is intractable, so we take a more heuristic approach. The strategy is to search locally through the space of DSLs, proposing small local changes to $\mathcal{D}$ until J 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 frontier. These extracted subexpressions are fragments of the original programs, and can introduce new variables (Figure 2), which then become new functions in the DSL.

Closely related to Fragment Grammars (?) and Tree-Substitution Grammars (?), but context-sensitive

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:

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

Algorithm 2 DSL Induction Algorithm

Input: Set of frontiers $\{\mathcal{F}_x\}$
Hyperparameters: Pseudocounts α, regularization parameter λ
Output: DSL $\mathcal{D}$, weight vector θ
Define $L(\mathcal{D},\theta) = \prod_x \sum_{z\in\mathcal{F}_x}\mathbb{P}[z|\mathcal{D},\theta]$
Define $\theta^*(\mathcal{D}) = \arg\max_\theta \text{Dir}(\theta|\alpha)L(\mathcal{D},\theta)$
Define $\text{score}(\mathcal{D}) = \log\mathbb{P}[\mathcal{D}] + L(\mathcal{D},\theta^*) - ||\theta||_0$
$\mathcal{D} \leftarrow$ every primitive in $\{\mathcal{F}_x\}$
while true **do**
  N $\leftarrow \{\mathcal{D}\cup\{s\}|x\in X, z\in\mathcal{F}_x, s$ subexpression of $z\}$
  $\mathcal{D}' \leftarrow \arg\max_{\mathcal{D}'\in N}\text{score}(\mathcal{D}')$
  if $\text{score}(\mathcal{D}') < \text{score}(\mathcal{D})$ **return** $\mathcal{D},\theta^*(\mathcal{D})$
  $\mathcal{D} \leftarrow \mathcal{D}'$
end while

Algorithm 3 The HELMHOLTZHACKER Algorithm

Input: Initial DSL $\mathcal{D}$, set of tasks X, iterations I
Hyperparameters: Maximum frontier size F
Output: DSL $\mathcal{D}$, weight vector θ, bottom-up recognition model $q(\cdot)$
Initialize $\mathcal{D}_0 \leftarrow \mathcal{D}$, $\theta_0 \leftarrow$ uniform, $q_0(\cdot) = \theta_0$
for $i = 1$ **to** I **do**
  for $x : \tau \in X$ **do**
    $\mathcal{F}_x \leftarrow \{z|z \in \text{enumerate}(\mathcal{D}_{i-1}, q_{i-1}(x), F) \cup \text{enumerate}(\mathcal{D}_{i-1},\theta_{i-1},F)$ if $\mathbb{P}[x|z] > 0\}$
  end for
  $\mathcal{D}_i,\theta_i \leftarrow$induceDSL$(\{\mathcal{F}_x\}_{x\in X})$
  $q_i \leftarrow$ train recognition model to maximize $\mathcal{L}_{\text{RM}}$
end for
return $\mathcal{D}^I,\theta^I,q^I$

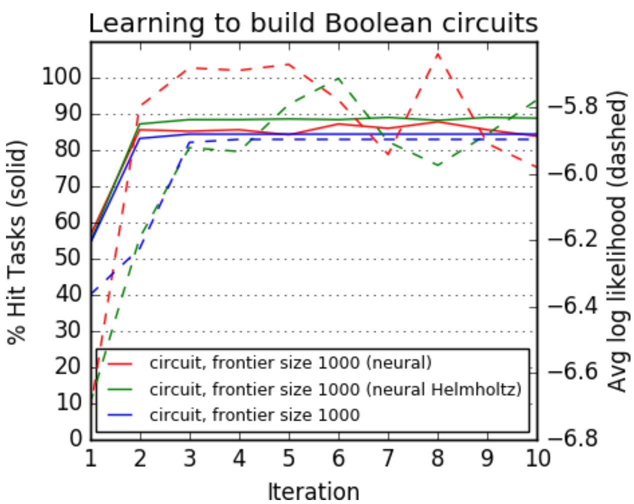
where $\text{size}(p)$ measures the size of the syntax tree of program p, λ is a hyperparameter that acts as a regularizer on the size of the DSL, and α is a concentration parameter controlling the smoothness of the prior over θ. Algorithm 2 specifies the DSL induction algorithm.

Additionally, for each proposed $\mathcal{D}$ we have to reestimate θ. Although this problem may seem very similar to estimating the parameters of a probabilistic context free grammar (PCFG: see ()), for which we have effective approaches like the Inside/Outside algorithm (?), $\mathcal{D}$ is actually context-sensitive due to the presence of variables in the programs and also due to the polymorphic typing system. In the Appendix we derive a tractable MAP estimator for θ.

Putting it all together. Algorithm 3 describes how we combine the program search, recognition model training, and DSL induction.

Domain	Example programs in frontiers	Example proposed subexpression
Boolean circuits	`(lambda x (nand x x))` `(lambda x (lambda y (nand x (nand y y))`	`(nand z z)`
String editing	`(lambda s (+ ',' (index 0 (split ',' s))))` `(lambda s (index 0 (split ' ' s)))`	`(index 0 (split z s))`

Figure 2: The DSL induction algorithm works by proposing subexpressions of programs to add to the DSL. These subexpressions are taken from programs in the frontiers (middle column), and can introduce new variables (z in the right column).



3. Experiments

3.1. Boolean circuits

pedagogical example; easy domain

3.2. Symbolic Regression

We show how to use HELMHOLTZHACKER to infer programs containing both discrete structure and continuous parameters. The high-level idea is to synthesize programs with unspecified-real-valued parameters, and to fit those parameters using gradient descent. Concretely, we ask the algorithm to solve a set of 1000 symbolic regression problems, each a polynomial of degree 0, 1, or 2, where our observations x take the form of N input/output examples, which we write as $x = \{(i_n, o_n)\}_{n\leq N}$. For example, one task is to infer a program calculating $3x + 2$, and the observations are the input-output examples $\{(-1, -1), (0, 2), (1, 5)\}$.

We initially equip our DSL learner with addition and multiplication, 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 integrate over the real-valued parameters, which we collectively write as $\vec{\mathcal{R}}$:

$$\log \mathbb{P}\left[\{(i_n, o_n)\}|p\right] = \log \int \mathrm{d}\vec{\mathcal{R}}\; P_{\vec{\mathcal{R}}}(\vec{\mathcal{R}}) \prod_{n\leq N} \mathcal{N}(p(i_n, \vec{\mathcal{R}})|o_n)$$

where $\mathcal{N}(\cdot|\cdot)$ is the normal density and $P_{\vec{\mathcal{R}}}(\cdot)$ is a prior over $\vec{\mathcal{R}}$. We approximate this marginal using the BIC (Bishop, 2006):

$$\log \mathbb{P}[x|p] \approx \sum_{n\leq N} \log \mathcal{N}(p(i_n, \vec{\mathcal{R}}^*)|o_n) - \frac{D \log N}{2}$$

where $\vec{\mathcal{R}}^*$ is an assignment to $\vec{\mathcal{R}}$ found by performing gradient ascent on the likelihood of the observations w.r.t. $\vec{\mathcal{R}}$.

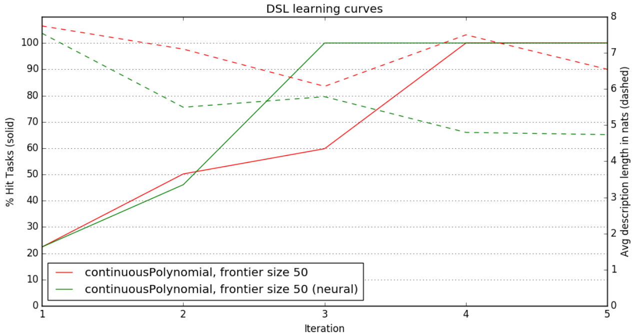
What DSL does HELMHOLTZHACKER learn? The learned DSL contains templates for quadratic and linear functions, which lets the algorithm quickly hone in on the kinds of functions that are most appropriate to this domain. Examining the programs themselves, one finds that the algorithm discovers representations for each of the polynomials that minimizes the number of continuous degrees of freedom: for example, it represents the polynomial $8x^2 + 8x$

Primitives	$+, \times : \mathbb{R} \to \mathbb{R} \to \mathbb{R}$ $\mathcal{R} : \mathbb{R}$ (real valued parameter)	
Observation x	N input/output examples: $\{(i_n, o_n)\}_{n\leq N}$	
Likelihood $\mathbb{P}[x\|p]$	$\propto \exp\left(-D \log N\right) \prod_{n\leq N} \mathcal{N}(p(i_n)\|o_n)$	
Subset of Learned DSL	$\lambda x.\mathcal{R} \times x + \mathcal{R}$ $\lambda x.\mathcal{R} + x$ $\lambda x.x \times$ (linear x) λx.increment (quadratic$_0$ x)	linear increment quadratic$_0$ quadratic

3.3. String editing

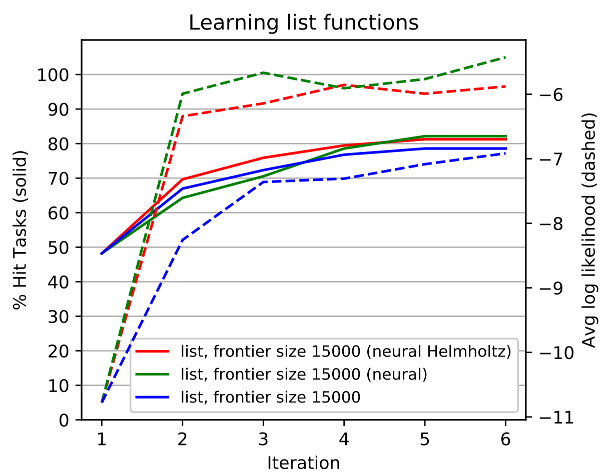
3.4. List functions

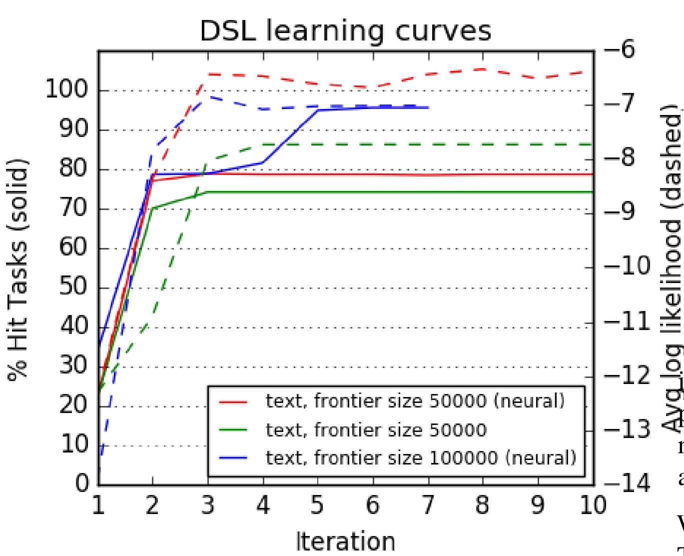
Our list function domain consists of tasks which are solved by functions that take as input an integer or a list of integers, and have as output either a Boolean, an integer, or a list of integers. Examples of these functions are in Table 1. For each function, we create a task x by generating 15



name	*input*	*output*
add-3	[1 2 3 4]	[4 5 6 7]
append-4	[7 0 2]	[7 0 2 4]
len	[3 5 12 1]	4
range	3	[1 2 3]
has-2	[4 5 7 4]	`false`
has-4	[4 5 7 4]	`true`
repeat-2	[7 0]	[7 0 7 0]
drop-3	[0 3 8 6 4]	[6 4]

Table 1: Examples from the domain of list functions.





input/output examples used for testing whether a program produces the correct output. Supplying many examples reduces ambiguity in the task's function, ensuring solutions achieve the desired concept.

We supply HELMHOLTZHACKER with the DSL outlined in Table 2.

We found that using a less sophisticated but equally-capable DSL made common patterns, such as summation, unlikely and unlearnable in a small enumeration bound.

4. Model

name	*type*
empty	`[`α`]`
singleton	α $\rightarrow$ `[`α`]`
concat	`[`α`]` $\rightarrow$ `[`α`]` $\rightarrow$ `[`α`]`
map	`(`α $\rightarrow$ β`)` $\rightarrow$ `[`α`]` $\rightarrow$ `[`β`]`
reduce	`(`β $\rightarrow$ α $\rightarrow$ β`)` $\rightarrow$ β $\rightarrow$ `[`α`]` $\rightarrow$ β
true	`bool`
not	`bool` $\rightarrow$ `bool`
and	`bool` $\rightarrow$ `bool` $\rightarrow$ `bool`
or	`bool` $\rightarrow$ `bool` $\rightarrow$ `bool`
0, . . ., 9	`int`
+	`int` $\rightarrow$ `int` $\rightarrow$ `int`
*	`int` $\rightarrow$ `int` $\rightarrow$ `int`
negate	`int` $\rightarrow$ `int`
mod	`int` $\rightarrow$ `int` $\rightarrow$ `int`
eq?	`int` $\rightarrow$ `int` $\rightarrow$ `bool`
gt?	`int` $\rightarrow$ `int` $\rightarrow$ `bool`
is-prime	`int` $\rightarrow$ `bool`
is-square	`int` $\rightarrow$ `bool`
range	`int` $\rightarrow$ `[int]`
sort	`[int]` $\rightarrow$ `[int]`
sum	`[int]` $\rightarrow$ `int`
reverse	`[`α`]` $\rightarrow$ `[`α`]`
all	`(`α $\rightarrow$ `bool)` $\rightarrow$ `[`α`]` $\rightarrow$ `bool`
any	`(`α $\rightarrow$ `bool)` $\rightarrow$ `[`α`]` $\rightarrow$ `bool`
index	`int` $\rightarrow$ `[`α`]` $\rightarrow$ α
filter	`(`α $\rightarrow$ `bool)` $\rightarrow$ `[`α`]` $\rightarrow$ `[`α`]`
slice	`int` $\rightarrow$ `int` $\rightarrow$ `[`α`]` $\rightarrow$ `[`α`]`

Table 2: DSL for the domain of list function.

5. Program Representation

We choose to represent programs using λ-calculus (Pierce, 2002). A λ-calculus expression is either:
A *primitive*, like the number 5 or the function `sum`.
A *variable*, like x, y, z
A λ*-abstraction*, which creates a new function. λ-abstractions have a variable and a body. The body is a λ-calculus expression. Abstractions are written as λvar.body.
An *application* of a function to an argument. Both the function and the argument are λ-calculus expressions. The application of the function f to the argument x is written as $f\ x$.

For example, the function which squares the logarithm of a number is $\lambda x.\texttt{square}(\texttt{log}\ x)$, and the identity function $f(x) = x$ is $\lambda x.x$. The λ-calculus serves as a spartan but expressive Turing complete program representation, and distills the essential features of functional languages like Lisp.

However, many λ-calculus expressions correspond to ill-typed programs, such as the program that takes the logarithm of the Boolean `true` (i.e., `log true`) or which applies the number five to the identity function (i.e., $5\ (\lambda x.x)$). We use a well-established typing system for λ-calculus called *Hindley-Milner typing* (Pierce, 2002), which is used in programming languages like OCaml. The purpose of the typing system is to ensure that our programs never call a function with a type it is not expecting (like trying to take the logarithm of `true`). Hindley-Milner has two important features: Feature 1: It supports *parametric polymorphism*: meaning that types can have variables in them, called *type variables*. Lowercase Greek letters are conventionally used for type variables. For example, the type of the identity function is $\alpha \rightarrow \alpha$, meaning it takes something of type α and return something of type α. A function that returns the first element of a list has the type $\texttt{list}(\alpha) \rightarrow \alpha$. Type variables are not the same has variables introduced by λ-abstractions. Feature 2: Remarkably, there is a simple algorithm for automatically inferring the polymorphic Hindley-Milner type of a λ-calculus expression (Damas & Milner, 1982). A detailed exposition of Hindley-Milner is beyond the scope of this work.

6. Estimating θ

We write $c(e, p)$ to mean the number of times that primitive e was used in program p; $R(p)$ to mean the sequence of types input to sample in Alg.1. Jensen's inequality gives an intuitive lower bound on the likelihood of a program p:

$$\log \mathbb{P}[p|\theta] \stackrel{\pm}{=} \sum_{e\in\mathcal{D}} c(e,p)\log\theta_e - \sum_{\tau\in R(p)} \log \sum_{\substack{e:\tau'\in\mathcal{D}\\ \text{unify}(\tau,\tau')}} \theta_e$$

$$\stackrel{+}{\geq} \sum_{e\in\mathcal{D}} c(e,p)\log\theta_e - c(p)\log \sum_{\tau\in R(p)} \sum_{\substack{e:\tau'\in\mathcal{D}\\ \text{unify}(\tau,\tau')}} \theta_e$$

$$= \sum_{e\in\mathcal{D}} c(e,p)\log\theta_e - c(p)\log\sum_{e\in\mathcal{D}} r(e,p)\theta_e$$

where $c(p) = \sum_{e\in\mathcal{D}} c(e,p)$ and $r(e : \tau', p) = \sum_{\tau\in R(p)} \mathbb{1}[\text{canUnify}(\tau,\tau')]$.

Differentiate with respect to θ_e and set to zero

$$\frac{c(x)}{\theta^{(x)}} = N\frac{a(x)}{\sum_y a(y)\theta_y} \tag{4}$$

This equality holds if $\theta^{(x)} = c(x)/a(x)$:

$$\frac{c(x)}{\theta_x} = a(x). \tag{5}$$

$$N\frac{a(x)}{\sum_y a(y)\theta_y} = N\frac{a(x)}{\sum_y c(y)} = N\frac{a(x)}{N} = a(x). \tag{6}$$

If this equality holds then $\theta_x \propto c(x)/a(x)$:

$$\theta_x = \frac{c(x)}{a(x)} \times \underbrace{\frac{\sum_y a(y)\theta_y}{N}}_{\text{Independent of } x}. \tag{7}$$

Now what we are actually after is the parameters that maximize the joint log probability of the data+parameters, which I will write J:

$$J = L + \log \mathrm{D}(\theta|\alpha) \tag{8}$$

$$\stackrel{+}{\geq} \sum_x c(x)\log\theta_x - N\log\sum_x a(x)\theta_x + \sum_x (\alpha_x - 1)\log\theta_x \tag{9}$$

$$= \sum_x (c(x)+\alpha_x - 1)\log\theta_x - N\log\sum_x a(x)\theta_x \tag{10}$$

So you add the pseudocounts to the *counts* ($c(x)$), but not to the *possible counts* ($a(x)$).

References

Balog, Matej, Gaunt, Alexander L, Brockschmidt, Marc, Nowozin, Sebastian, and Tarlow, Daniel. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

Bishop, Christopher M. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 0387310738.

Damas, Luis and Milner, Robin. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 207–212. ACM, 1982.

Dechter, Eyal, Malmaud, Jon, Adams, Ryan P., and Tenenbaum, Joshua B. Bootstrap learning via modular concept discovery. In *IJCAI*, pp. 1302–1309. AAAI Press, 2013. ISBN 978-1-57735-633-2. URL http://dl.acm.org/citation.cfm?id=2540128.2540316.

Devlin, Jacob, Uesato, Jonathan, Bhupatiraju, Surya, Singh, Rishabh, Mohamed, Abdel-rahman, and Kohli, Pushmeet. Robustfill: Neural program learning under noisy i/o. *arXiv preprint arXiv:1703.07469*, 2017.

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

Feser, John K, Chaudhuri, Swarat, and Dillig, Isil. Synthesizing data structure transformations from input-output examples. In *ACM SIGPLAN Notices*, volume 50, pp. 229–239. ACM, 2015.

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

Lake, Brenden M, Salakhutdinov, Ruslan R, and Tenenbaum, Josh. One-shot learning by inverting a compositional causal process. In *Advances in neural information processing systems*, pp. 2526–2534, 2013.

Liang, Percy, Jordan, Michael I., and Klein, Dan. Learning programs: A hierarchical bayesian approach. In Fürnkranz, Johannes and Joachims, Thorsten (eds.), *ICML*, pp. 639–646. Omnipress, 2010. ISBN 978-1-60558-907-7.

Muggleton, Stephen H, Lin, Dianhuan, and Tamaddoni-Nezhad, Alireza. Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. *Machine Learning*, 100(1):49–73, 2015.

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

Pierce, Benjamin C. *Types and programming languages*. MIT Press, 2002. ISBN 978-0-262-16209-8.

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

Ritchie, Daniel, Horsfall, Paul, and Goodman, Noah D. Deep amortized inference for probabilistic programs. *arXiv preprint arXiv:1610.05735*, 2016.

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

Solar Lezama, Armando. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-177.html.

Stolle, Martin and Precup, Doina. Learning options in reinforcement learning. In *International Symposium on abstraction, reformulation, and approximation*, pp. 212–223. Springer, 2002.