DREAMCODER: Bootstrapping Domain-Specific Languages for Neurally-Guided Bayesian Program Learning

Kevin Ellis [1] **Lucas Morales** [1] **Mathias Sablé Meyer** [1] **Armando Solar-Lezama** [1] **Joshua B. Tenenbaum** [1]

1. Introduction

Humans can induce programs flexibly across many different domains, often from just one or a few examples. If shown that a text-editing program should map "Jane Morris Goodall" to "J. M. Goodall", we can guess it maps "Richard Erskine Leakey" to "R. E. Leakey"; if the first input mapped to "Dr. Jane" or "Goodall, Jane", we might guess the latter mapped to "Dr. Richard" or "Leakey, Richard", respectively.

The FlashFill system (Gulwani, 2011) embedded in Microsoft Excel solves problems such as these and is probably the best known practical program-induction algorithm, but researchers have had many successes: handwriting recognition and generation (Lake et al., 2015); question answering (Johnson et al.); and robot motion planning (Devlin et al., 2017). These systems work in different ways, but most hinge upon a carefully engineered **Domain Specific Language (DSL)**. DSLs constrain the space of programs with strong prior knowledge in the form of a restricted set of primitives tuned to the domain: for text editing, these are operations like appending and splitting on characters.

In this work, we consider the problem of building agents that learn to solve program induction tasks, and also the problem of acquiring the prior knowledge necessary to quickly solve these tasks in a new domain; see Table 1. Our solution is an algorithm that grows or boostraps a DSL while jointly training a neural network to help write programs in the increasingly rich DSL. In contrast to computer assisted programming (Solar Lezama, 2008) or genetic programming (Koza), our goal is not to automate software engineering, or to synthesize large bodies of code starting from scratch. Ours is a basic AI goal: capturing the human ability to learn to think flexibly and efficiently in new domains — to learn what you need to know about a domain so you don't have to solve new problems starting from scratch.

[1]MIT. Correspondence to: Kevin Ellis <ellisk@mit.edu>.

Published at the ICML workshop Neural Abstract Machines & Program Induction v2 (NAMPI) — Extended Abstract, Stockholm, Sweden, 2018.

2. The DREAMCODER Algorithm

We are inspired by several ways that skilled human programmers have learned to code: Skilled coders build libraries of reusable subroutines shared across related programming tasks, that can be composed to generate increasingly complex subroutines. In text editing, a good library should support routines for splitting on characters, but also specialize these routines to split on frequent delimiters like spaces. Skilled coders also learn to recognize which programming idioms and library routines are useful for solving the task at hand, even if they cannot instantly work out the details.

Our algorithm is called DREAMCODER because it incorporates these insights into a novel kind of "wake-sleep" learning (c.f. (Hinton et al., 1995)), iterating between "wake" and "sleep" phases to achieve three goals: finding programs that solve tasks; growing a DSL by discovering and reusing domain-specific subroutines; and training a neural network that guides search for programs in the DSL. The learned DSL effectively encodes a prior on programs likely to solve tasks in the domain, while the neural net looks at a specific task and produces a "posterior" for programs likely to solve that specific task. The neural network thus functions as a **recognition model** supporting a form of approximate Bayesian program induction, jointly trained with a **generative model** for programs encoded in the DSL, in the spirit of the Helmholtz machine (Hinton et al., 1995). The recognition model ensures that searching for programs remains tractable even as the DSL expands.

Concretely, our algorithm iterates through three cycles:
Wake Cycle: Given a set of **tasks**, we search for compact programs that solve these tasks, taking the simple strategy of enumerating programs written in the current DSL in order of their probability according to the neural network.
Sleep-G Cycle: Here we grow the DSL (**G**enerative model), allowing the agent to more compactly write programs in the domain. We modify the DSL by discovering regularities across programs found during waking, compressing them to distill out common code fragments. The technical machinery behind this DSL learning comes from a formalism known as Fragment Grammars (O'Donnell, 2015).
Sleep-R Cycle: This trains a neural net (**R**ecognition model) to predict a distribution over programs in the cur-

	List Functions		Text Editing	Symbolic Regression	

Programs & Tasks	`[7 2 3]`→`[7 3]`
	`[1 2 3 4]`→`[3 4]`
	`[4 3 2 1]`→`[4 3]`
	$f(\ell)$ = (f_1 ℓ (λ (x) (> x 2)))
	
	`[2 7 8 1]`→`8`
	`[3 19 14]`→`19`
	$f(\ell)$ = (f_2 ℓ)
		`[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)))
			
			Temple Anna H →TAH
			Lara Gregori→LG
			f(s) = (f_2 s)
				
				f(x) = (f_1 x)
				
				
				f(x) = (f_4 x)
					
					f(x) = (f_6 x)
					
					
					f(x) = (f_3 x)

DSL	f_1(ℓ,p) = (foldr ℓ nil (λ (x a) (if (p x) (cons x a) a))) (f_1: *Higher-order filter function*) 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_3: *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_5 (f_0 x)) (f_6: *rational function*)	

Table 1: Top: Tasks from three domains we apply our algorithm to, each followed by the programs DREAMCODER discovers for them. Bottom: Several examples from learned DSL. Notice that learned DSL primitives can call each other, and that DREAMCODER rediscovers higher-order functions like `filter` (f_1 under List Functions)

rent DSL, in the spirit of "amortized" inference (Le et al., 2017). The network is trained, conditioned on a task, to assign high probability to programs that have high prior probability according to the DSL, while also assigning high likelihood to the task at hand, thus amortizing the cost of finding programs with high posterior probability.

3. Experiments

We apply DREAMCODER to list processing and text editing, using a recurrent network for the recognition model, initially providing the system with generic sequence manipulation primitives: `foldr`, `unfold`, `if`, `map`, `length`, `index`, `=`, `+`, `-`, `0`, `1`, `cons`, `car`, `cdr`, `nil`, and `is-nil`.

List Processing: Synthesizing programs that manipulate data structures is a widely studied problem in the programming languages community (Feser et al., 2015). We consider this problem within the context of learning functions that manipulate lists, creating 236 human-interpretable list manipulation tasks (Table 1, left). In solving these tasks, the system composed 38 new subroutines, and rediscovered the higher-order function `filter` (f_1 in Table 1, left).

Text Editing: Synthesizing programs that edit text is a classic problem in the programming languages and AI literatures (Gulwani, 2011; Lau, 2001). This prior work uses hand-engineered DSLs. Here, we instead start out with generic sequence manipulation primitives and recover many of the higher-level building blocks that have made these other systems successful.

We trained our system on 109 automatically-generated text editing tasks. After three wake/sleep cycles, it assembles a DSL containing a dozen new functions (Fig. 1, center) solving the training tasks. We also tested, but did not train, on the 108 text editing problems from the SyGuS (Alur et al., 2016) program synthesis competition. Before any learning, DREAMCODER 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 29 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.

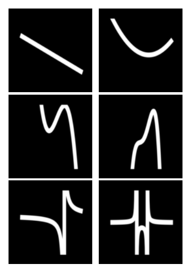

Figure 1: Recognition model input for symbolic regression. DSL learns subroutines for polynomials (top rows) and rational functions (bottom) while the recognition model jointly learns to look at a graph of the function (left) and predict which learned subroutines best explain the observation.

Symbolic Regression: Programs from visual input. We apply DREAMCODER 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 problems, each either a polynomial or rational function. The recognition model is a convnet that observes an image of the target function's graph (Fig. 1) — visually, different kinds of polynomials and rational functions produce different kinds of graphs, and so the convnet can look at a graph and predict what kind of function best explains it.

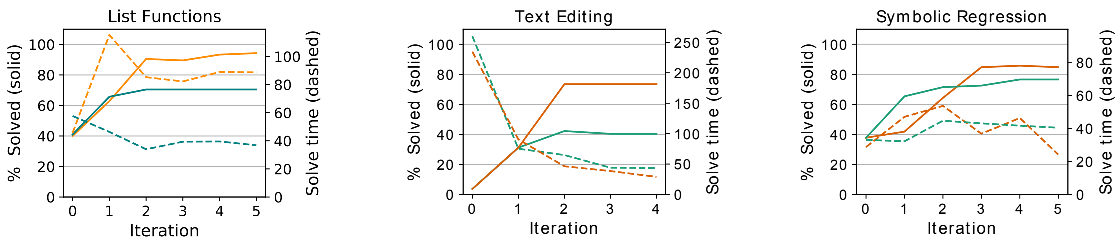

Figure 2: Learning curves for DREAMCODER both with (in orange) and without (in teal) the recognition model. Iterations: # wake/sleep cycles. Solid lines: % holdout testing tasks solved w/ 10m timeout. Dashed lines: Average solve time.

4. Discussion

We contribute an algorithm, DREAMCODER, 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 to efficiently deploy the DSL on new tasks. One future direction is DSL meta-learning: can we find a *single* universal primitive set that could bootstrap DSLs for new domains, including the domains considered here, but also many others?

References

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

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

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

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

Hinton, Geoffrey E, Dayan, Peter, Frey, Brendan J, and Neal, Radford M. The" wake-sleep" algorithm for unsupervised neural networks. *Science*, 268(5214):1158–1161, 1995.

Johnson, Justin, Hariharan, Bharath, van der Maaten, Laurens, Fei-Fei, Li, Zitnick, C Lawrence, and Girshick, Ross. Clevr: A diagnostic dataset for compositional language and elementary visual reasoning. In *CVPR*.

Koza, John R.

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

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

Le, Tuan Anh, Baydin, Atlm Gne, and Wood, Frank. Inference Compilation and Universal Probabilistic Programming. In *AISTATS*, 2017.

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

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