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

Anonymous Authors[1]

1. Introduction

Much of everyday human thinking and learning can be understood in terms of program induction: constructing a procedure that maps inputs to desired outputs, based on observing example input-output pairs. People can induce programs flexibly across many different domains, often from just one or a few examples. For instance, 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 instead the first input mapped to "Dr. Jane" or "Goodall, Jane", we might guess the latter should map 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 in programming languages and AI have had successes in many domains, such as handwriting recognition and generation (Lake et al., 2015), procedural graphics (Ellis et al., 2017), question answering (Johnson et al., 2017) and robot motion planning (Devlin et al., 2017a). These systems work in different ways, but most hinge upon a carefully engineered **Domain Specific Language (DSL)**. This is especially true for systems like FlashFill that induce a wide range of programs very quickly, in a few seconds or less. DSLs constrain the search over programs with strong prior knowledge in the form of a restricted set of programming primitives finely 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. Representative problems in three domains are shown in 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, 1993), 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.

2. The DREAMCODER Algorithm

We take inspiration from several ways that skilled human programmers have learned to code: Skilled coders build libraries of reusable subroutines that are shared across related programming tasks, and 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 such as spaces or commas. Skilled coders also learn to 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. In text editing, one might learn that if outputs are consistently shorter than inputs, removing characters is likely part of the solution.

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; creating a DSL by discovering and reusing domain-specific subroutines; and training a neural network that efficiently 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 the example input-output pairs for 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 (and hence the search space for programs) expands.

Concretely, our algorithm iterates through three cycles:
Wake Cycle: Here we take a given set of **tasks**, typically several hundred, and search for compact programs that solve these tasks. Our current implementation uses a simple and generic enumerative program synthesis algorithm, enumerating programs written in the current DSL in order of their probability according to the neural network.

	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] $f(\ell)$ = (f1 ℓ (λ (x) (> x 2))) [2 7 8 1]→8 [3 19 14]→19 $f(\ell)$ = (f2 ℓ) [7 3]→False [3]→False [9 0 0]→True [0]→True [0 7 3]→True $f(\ell)$ = (f3 ℓ 0)	+106 769-438→106.769.438 +83 973-831→83.973.831 f(s) = (f0 "." "-" (f0 "." " " (cdr s))) Temple Anna H →TAH Lara Gregori→LG f(s) = (f2 s)	 f(x) = (f1 x)  f(x) = (f6 x)  f(x) = (f4 x)  f(x) = (f3 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)

Sleep-G Cycle: Here we grow the the DSL (or **G**enerative model), which allows the agent to more compactly write programs in the domain. We modify the structure of the DSL by discovering regularities across programs found during waking, compressing them to distill out common code fragments across successful programs. The technical machinery behind this DSL learning comes from a formalism known as Fragment Grammars (O'Donnell, 2015), which casts DSL induction as grammar induction.

Sleep-R Cycle: Here we improve the search procedure by training a neural network (the **R**ecognition model) to predict a distribution over programs written in the current DSL, in the spirit of "amortized" or "compiled" inference (Le et al., 2017). The recognition model is trained, conditioned on a task, to assign high probability to programs that (1) have high prior probability according to the current DSL, while (2) also assigning high likelihood to the task at hand, thus amortizing the cost of finding programs with high posterior probability.

3. Experiments

3.1. Programs that manipulate sequences

We apply DREAMCODER to list processing and text editing, using a GRU (Cho et al., 2014) for the recognition model, and 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, and also perform arithmetic operations upon lists (Table **??**). We created 236 human-interpretable list manipulation tasks, each with 15 input/output examples. 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, with 4 input/output examples each. After three iterations, it assembles a DSL containing a dozen new functions (Fig. 1, center) solving all the training tasks. But, how well does the learned DSL generalized to real text-editing scenarios? We 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.

3.2. 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

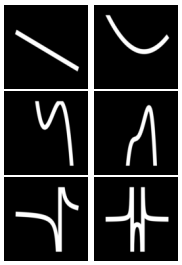

Figure 1: Recognition model input for symbolic regression. DSL learns subroutines for polynomials (top rows) and rational functions (bottom) while the recognition model jointly learns to look at a graph of the function (left) and predict which learned subroutines best explain the observation.

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. These programs can contain real numbers, which we set with gradient descent, and penalize continuous parameters by incorporating a BIC penalty into the likelihood model $\mathbb{P}[x|p]$. We learn a DSL containing 13 new functions, mainly templates for polynomials of different orders or ratios of polynomials. The model also learns to find programs minimizing the number of continuous parameters — learning to represent linear functions with `(* real (+ x real))`, which has two continuous parameters, and represents quartic functions using f_4 in the rightmost column of Fig. 1 which has five continuous parameters. This phenomenon arises from our Bayesian framing.

3.3. Quantitative Results on Held-Out Tasks

We evaluate on held-out testing tasks, measuring how many tasks are solved and how long it takes to solve them (Fig. 2). Prior to any learning, the system cannot find solutions for most of the tasks, and those it does solve take a long time; with more wake/sleep iterations, we converge upon DSLs and recognition models more closely matching the domain.

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. Two immediate future goals are to integrate more sophisticated neural recognition models (Devlin et al., 2017b) and program synthesizers (Solar Lezama, 2008), which may improve performance in some domains over the generic methods used here. Another 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.

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

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

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

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

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*, 2017b.

Ellis, Kevin, Ritchie, Daniel, Solar-Lezama, Armando, and Tenenbaum, Joshua B. Learning to infer graphics programs from hand-drawn images. *arXiv preprint arXiv:1707.09627*, 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 *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on*, pp. 1988–1997. IEEE, 2017.

Kalyan, Ashwin, Mohta, Abhishek, Polozov, Oleksandr, Batra, Dhruv, Jain, Prateek, and Gulwani, Sumit. Neural-guided deductive search for real-time program synthesis from examples. *ICLR*, 2018.

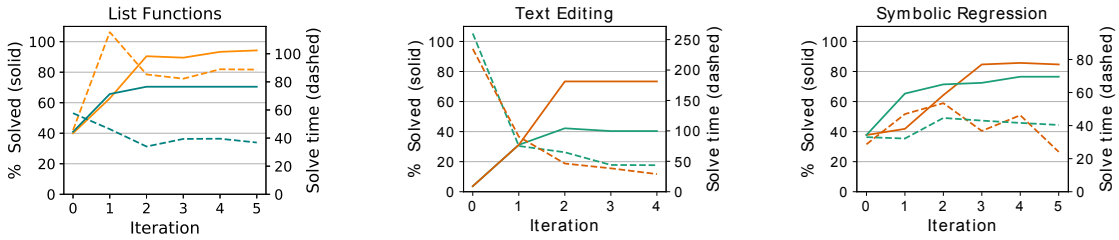

Figure 2: Learning curves for DREAMCODER both with (in orange) and without (in teal) the recognition model. Solid lines: % holdout testing tasks solved w/ 10m timeout. Dashed lines: Average solve time, averaged only over tasks that are solved.

Koza, John R. *Genetic programming - on the programming of computers by means of natural selection*. Complex adaptive systems. MIT Press, 1993. ISBN 978-0-262-11170-6.

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.

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.

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.