Thesis Proposal: Automatically Learning and Growing Libraries of Code

Arushi Somani

September 23, 2022

1 Introduction

DreamCoder: Growing generalizable, interpretable knowledge with wake-sleep Bayesian program learning (Ellis K et al 2019) introduces the idea of ”library learning”. It posits that, given some primitives and the ability to combine and test the use of such primitives, machines can learn to develop higher-level routines, which can then further be combined. This form of curriculum learning would allow machines to develop deep and complex libraries from simple start-points, which can then be used as automated programmers.

My thesis work attempts to build upon the contributions of this paper in two ways:

1. Instead of manually crafting a DSL of primitives for the model to build upon, then iteratively training it on I/O tests as training set, we develop a data-driven grammar induction technique. We instead use open-source code available to us via Github and Kaggle to scalably learn **idiomatic patterns** in our generated DSLs.

2. We also mine the internet for problems solved by our DSL. This can be found as docstrings or of code that can then be executed to generate appropriate I/O examples. We train our model to fantasize on these tests. This allows the fantasizing phase to equate popular problems as more ”interesting”, thus biasing the model to solving them. We believe this a strict improvement over the random model from the Dreamcoder paper.

2 Project Breakdown

As highlighted in the figure in the previous section, the project can be broken down into phases and its sub-parts.

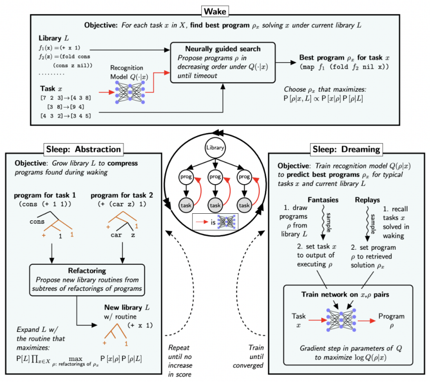
Wake
Objective: For each task x in X, find best program ρx solving x under current library L
Library L
Task x
Recognition Model Q(·|x)
Neurally guided search
Propose programs ρ in decreasing order under Q(·|x) until timeout
Best program ρx for task x
Choose ρx that maximizes:
Sleep: Abstraction
Objective: Grow library L to compress programs found during waking
program for task 1
program for task 2
Refactoring
Propose new library routines from subtrees of refactorings of programs
New library L w/ routine
Expand L w/ the routine that maximizes:
Repeat until no increase in score
Library
prog
task
Train until converged
Sleep: Dreaming
Objective: Train recognition model Q(ρ|x) to predict best programs ρx for typical tasks x and current library L
Fantasies
Replays
Train network on x,ρ pairs
Task x
Program ρ
Gradient step in parameters of Q to maximize log Q(ρ|x)

Figure 1: Dreamcoder Implementation

2.1 Wake Phase

The wake phase of the tool is the part where queries are given to the synthesis engine as pairs of input/output examples (PBE). The synthesizer accepts this query, a library of functions available to it, and a neural recognition model that guides it through the problem space. It returns the best program that implements the given task query, and also internally saves this program.

2.1.1 Bottom-up Enumerator

The core of the wake phase is a bottom-up enumerator, which walks through all possible combinations of productions/rules of a grammar. What this implies is that, given infinite time, we would find a program that solves a given task if it exists (completeness).

The design of the enumerator is such that, at every step, it returns programs that satisfy the given productions. Each production recursively calls more enumeration within it, with some terminals to stop the recursive process.

2.1.2 Library

The enumerator accepts its grammar as a library of functions L. These functions can be thought of as primitives that can be used in the production process.

2.1.3 Recognition Model

The recognition model can be thought of as a ranking of what patterns of enumeration are most likely, and helps the enumerator yield candidates in an order of likelihood instead of in a brute-force fashion.

2.2 Sleep: Abstraction

This phase adds to the library. It abstracts away finer details of the differences between functions found in the waking phase, and adds a higher-level routine to the library.

2.2.1 Function Compressor

This section looks at the various functions found in the waking step, and pulls out similar structures in the abstract syntax tree of these found functions. It passes these higher abstracted functions away to the library adder.

2.2.2 Library Adder

This section accepts functions and adds them to the library — either as a production or as a primitive (or both!). This library is then used in the waking phase, and the newly added functions as now top-level citizens of this new library.

2.3 Sleep: Dreaming

In this phase, we train our recognition model to be stronger than it is, and to better enumerate programs in the waking phase.

2.3.1 Recognition Model Update

This section replays the programs found in the waking phase, and updates the neural recognition model by these programs. Fantasies Update This section comes up with theoretical programs that are "interesting". The criteria for interesting is not defined by the paper so we pick a wide one for our given domain— they accept or return an example of the object defined in the domain.

3 Core Algorithms

In the above-defined phases, we execute various algorithms. The most salient few are listed here.

3.1 Bottom-Up Enumerator Synthesis Engine

This is used in the wake phase to enumerate through a grammar using the recognition model.

3.2 Expression Enumeration

Initially, we designed a depth-first enumeration model.

However, in practical testing, as library size grows — this algorithm becomes infeasible, since for every depth, the number of possible iterations are

Algorithm 1 Synthesis Engine

```
Task $T$, Grammar $g$, Recognition Model $Q$
Program $\rho : \rho \in E(g)$ where $E(g)$ = every possible program in $g$, or $\rho = \bot$
$sol \leftarrow None$
$context \leftarrow \{\}$
$step \leftarrow 0$
$programs \leftarrow$ GetExpressions$(g, Q, context, step)$
while programs $\neq \emptyset$ do $\rho \leftarrow next(programs)$
    if Satisfies(p, T) then $return\ \rho$
    $return\ \bot$
```

Algorithm 2 GetExpressions Depth

```
Grammar $g$, Recognition Model $Q$, context $C$
Programs $\rho : \rho \in E(g)$ where $E(g)$ = every program in $g$ ordered by $Q$
while program not found do $depth \leftarrow 0$
$order \leftarrow Q.order(depth,$
[Variable, Function, Application]$)$

    for $rule \in order$ do
        $yield\ from$ GetEnums$(rule, depth, C)$
$depth \leftarrow depth + 1$
```

len(primitives)!. Instead, we move to a different algorithm that does not rely so heavily on depth, which is as follows:

Algorithm 3 GetExpressions Step

```
Grammar $g$, Recognition Model $Q$, context $C$, step $s$
Programs $\rho : \rho \in E(g)$ where $E(g)$ = every program in $g$ ordered by $Q$
while program not found do
$order \leftarrow Q.order(step,$
[Variable, Function, Application]$)$

    for $rule \in order$ do
        $yield\ from$ GetEnums$(rule, step + 1, C)$
```

We consider non-terminals in the grammar to be a variable name, a function, or an application. This is encoded as defined in GetEnums.

Algorithm 4 GetEnums

```
Rule $r$, Recognition Model $Q$, context $C$ | grammar $g$
Programs $\rho : \rho \in E(g)$ where $E(r)$ = every program in $r$ ordered by $Q$
if $r$ = Variable then
    for $var \in C$ do yield $var$

        if $r$ = Function then
$arg \leftarrow$ UnusedVarName
$C' \leftarrow C + arg$
$bodies \leftarrow$ GetExpressions$(g, Q, C', step + 1)$
            for $body \in bodies$ do
yield lambda arg:body

            if $r$ = Application then
$expr1 \leftarrow$ GetExpressions$(g, Q, C, step + 1)$
$expr2 \leftarrow$ GetExpressions$(g, Q, C, step + 1)$
                for $e1, e2 \in expr1 \cdot expr2$ do  yield e1(e2)
```

4 Related Work

4.1 Criticisms of Dreamcoder

This section elaborates upon the limitations of the original Dreamcoder project. This section is deemed important because of the need to understand prior literature, and the weaknesses we are attempting to improve upon.

4.1.1 Synthesis Duration Explosion

As found during experimentation, and explained in the original paper, the neural model takes a long time to ramp up and start learning intelligible patterns. Leaving the model running for 36+ hours on 1 CPU only had very minimal learning of a small library of basic functions. While part of this is admittedly because of our poor optimization, the paper admits that an untrained recognition model would take very long to ramp up to being able to guide the enumeration — the training phase was probably substantially longer than their mentioned testing phase of 1-24 hours per task.

4.1.2 Need to Learn from Failures

The core problem with the learning model of this setup is that learning happens only through the correct solutions. Correct solutions, however, are very hard to come by, and incorrect candidates still have learnings that can be pulled from. For example, using them in the fantasies phase if they passed a substantial number (but not all) of the tests, a heuristic that they perform a different but no less relevant operation on the inputs.

4.1.3 Lack of Metric for "Interestingness" in Fantasies

To DreamCoder, all fantasies are interesting fantasies, so long as they are not buggy code. As such, the model does come up with extremely interesting functions, it also comes up with noisy and uninteresting functions. However, all of these are added to the model and the library. This causes the model to be more imprecise than it should be, and causes the library to bloat.

4.1.4 Explainability and Readability

The output of the system doesn't have any. This tool is clearly not meant to be used by an individual. If it ever were to be incorporated as a real life tool to be used, the details of its functioning would necessarily have to be abstracted away from the user.

4.2 Comparative Analysis

This section elaborates upon the differences between our project and the canonical DreamCoder. Needless to say, Dreamcoder has a lot more bells and whistles, and a lot more tooling to help it learn and operate at higher benchmarks. This section attempts to explain some of these shortcomings of Dreamer.

4.2.1 Wake Phase: Enumerator

In Dreamcoder, the enumerator is not a lambda calculus enumerator. Instead, it is a grammar enumerator, that stores a complex grammar that includes lambda calculus principles, and at every level of the recursive enumeration, emits these rules in order. Our enumerator only has three simple productions, which means we have to explore at greater depth to find solutions.

4.2.2 Library

As mentioned previously, we store new functions as variables that our enumerator can use in any of its productions. Dreamcoder stores new functions as productions themselves,.

4.2.3 Recognition Model

Our recognition model is a simple one-layer neural network. The dreamcoder recognition model is a "Contextual Grammar Network". They have developed a lot of featurizing tools over their programs, which helps their network train faster and yield better results. It is also worth mentioning that the recognition model in dreamcoder has much more complex feature extraction than our tool.

4.2.4 Focus on Creative Tasks

Unlike our project, which tries to develop DreamCoder for PBE tasks like string edits or list manipulation. However, DreamCoder is more focused on creative tasks, like drawing in LOGO or with bricks. As such, the metric for success is much less complex, as something that looks interesting is a successful development. Curriculum learning is made easier, whereas in our domain, it is much harder to do so.

5 Future Work

As elaborated in the introduction, there is future work to be done in terms of adding robustness to the grammar induction techniques, as well as further mining and development of the continuous fantast testing suite. I look forward to continuing this project beyond this thesis.