
Learning to Infer Graphics Programs from Hand-Drawn Images

Anonymous Author(s)

Affiliation

Address

email

Abstract

1 We introduce a model that learns to convert simple hand drawings into graphics
2 programs written in a subset of \LaTeX . The model combines techniques from deep
3 learning and program synthesis. We learn a convolutional neural network that
4 proposes plausible drawing primitives that explain an image. From these deep
5 net proposals we can use program synthesis techniques to recover a graphics
6 program with constructs like variable bindings, iterative loops, or simple kinds of
7 conditionals. [Can you (concisely) communicate the idea of treating the deep net
8 proposal as an ‘execution trace?’ It’s a really neat intellectual framing that deserves
9 to be foregrounded.] With a graphics program in hand, we can correct errors made
10 by the deep network, extrapolate drawings, and [what else: ‘visual analogies’ is
11 vague. ‘Cluster drawings by use of similar high-level program structures?’] Taken
12 together these results are a step towards agents that induce useful, human-readable
13 programs from perceptual input.

14 1 Introduction

15 How can an agent convert noisy, high-dimensional perceptual input to a symbolic, abstract object, such
16 as a computer program? Here we consider this problem within a graphics program synthesis domain.
17 We develop an approach for converting natural images, such as hand drawings, into executable source
18 code for drawing the original image. The graphics programs in our domain draw simple figures like
19 those found in machine learning papers (see Figure 1).

20 High dimensional perceptual input may seem ill matched to the abstract semantics of a programming
21 language. But programs with constructs such as recursion or iteration produce a simpler *execution*
22 *trace* of primitive actions; for our domain, the primitive actions are drawing commands. Our
23 hypothesis is that the execution trace of the program is better aligned with the perceptual input, and
24 that the trace can act as a kind of bridge between perception and programs. We test this hypothesis
25 by developing a model that learns to map from an image to the execution trace of the graphics
26 program that drew it. With the execution trace in hand, we can bring to bear techniques from the
27 program synthesis community to recover the latent graphics program. This family of techniques,
28 called *constraint-based program synthesis* [1], work by modeling a set of possible programs inside of
29 a constraint solver, such as a SAT or SMT solver [2]. These techniques excel at uncovering high-level
30 symbolic structure, but are not well equipped to deal with real-valued perceptual inputs.

31 We develop a hybrid architecture for inferring graphics programs. Our approach uses a deep neural
32 network infer an execution trace from an image; this network recovers primitive drawing operations
33 such as lines, circles, or arrows, along with their parameters. For added robustness, we use the
34 deep network as a proposal distribution for a stochastic search over execution traces. Finally, we
35 use program synthesis to recover the program from its trace. The program synthesizer discovers
36 constructs such as loops and geometric operations such as reflections and affine transformations.

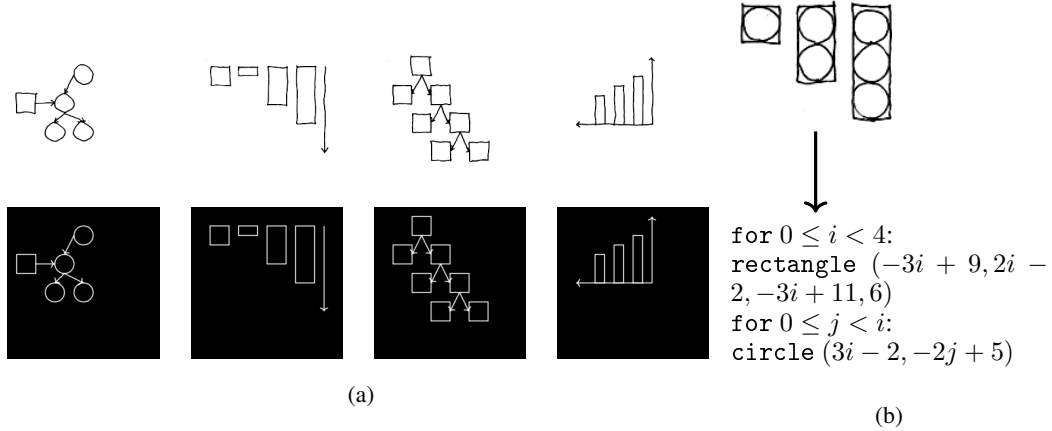


Figure 1: (a): Model learns to convert hand drawings (top) into L\^AT\^EX (bottom). (b) Synthesizes high-level *graphics program* from hand drawing. [Any objection to putting this on page bottom?]

Each of these three components – the deep network, the stochastic search, the program synthesizer – confers its own advantages. From the deep network, we get a fast system that can recover plausible execution traces in about a minute [A minute seems slow to me, for deep net inference...]. From the stochastic search, we get added robustness: essentially, the stochastic search can correct mistakes made by the deep network’s proposals. From the program synthesizer, we get abstraction: our system recovers coordinate transformations, for loops, and subroutines, which are useful for downstream tasks and can help correct some mistakes of the earlier stages. [Tie these claims into the paper results: state what the ‘downstream’ tasks are that you actually do, and refer to the sections where they occur]

2 Related work

The CogSketch [?] system also aims to have a high-level understanding of hand-drawn figures. Their primary goal is cognitive modeling, whereas we are interested in building an automated AI application.

Our work bears resemblance to the Attend-Infer-Repeat (AIR) system, which learns to decompose an image into its constituent objects [3]. AIR learns an iterative inference scheme which infers objects one by one and also decides when to stop inference; this is similar to our approach’s first stage, which parses images into program execution traces. Our approach further produces interpretable, symbolic programs which generate those execution traces. The two approaches also differ in their architectures and training regimes: AIR learns a recurrent auto-encoding model via variational inference, whereas our parsing stage learns an autoregressive-style model from randomly-generated (execution trace, image) pairs. Finally, while AIR was evaluated on multi-MNIST images and synthetic 3D scenes, we focus on parsing and interpreting hand-drawn sketches.

Our image-to-execution-trace parsing architecture builds on prior work on controlling procedural graphics programs [4]. Given a program which generates random 2D recursive structures such as vines, that system learns a structurally-identical “guide program” whose output can be directed, via neural networks, to resemble a given target image. We adapt this method to a different visual domain (figures composed of multiple objects), using a broad prior over possible scenes as the initial program and viewing the execution trace through the guide program as a symbolic parse of the target image. We then show how to synthesize higher-level programs from these execution traces.

In the computer graphics literature, there have been other systems which convert sketches into procedural representations. One uses a convolutional network to match a sketch to the output of a parametric 3D modeling system [5]. Another uses convolutional networks to support sketch-based instantiation of procedural primitives within an interactive architectural modeling system [6]. Both systems focus on inferring fixed-dimensional parameter vectors. In contrast, we seek to automatically infer a structured, programmatic representation of a sketch which captures higher-level visual patterns.

Prior work has also applied sketch-based program synthesis to authoring graphics programs. In particular, Sketch-n-Sketch presents a bi-directional editing system in which direct manipulations to a program’s output automatically propagate to the program source code [7]. We see this work as complementary to our own: programs produced by our method could be provided to a Sketch-n-Sketch-like system as a starting point for further editing.

[Do you also want to cite your own work on “Unsupervised Learning by Program Synthesis” here?]

3 Neural architecture for inferring drawing execution traces

We developed a deep network architecture for efficiently inferring a execution trace, T , from an image, I . Our model constructs the trace one drawing command at a time. When predicting the next drawing command, the network takes as input the target image I as well as the rendered output of previous drawing commands. Intuitively, the network looks at the image it wants to explain, as well as what it has already drawn. It then decides either to stop drawing or proposes another drawing command to add to the execution trace; if it decides to continue drawing, the predicted primitive is rendered to its “canvas” and the process repeats.

Figure 2 illustrates this architecture. We first pass a 256×256 target image and a rendering of the trace so far (encoded as a two-channel image) to a convolutional network. Given the features extracted by the convnet, a multilayer perceptron then predicts a distribution over the next drawing command to add to the trace. We predict the drawing command token-by-token, conditioning each token both on the image features and on the previously generated tokens. For example, the network first decides to emit the `circle` token conditioned on the image features, then it emits the x coordinate of the circle conditioned on the image features and the `circle` token, and finally it predicts the y coordinate of the circle conditioned on the image features, the `circle` token, and the x coordinate. [There are some more details that are important to provide about this architecture in the supplement: the functional form(s) of the probability distributions over tokens, the network layer sizes, which MLPs share parameters, etc.]

The distribution over the next drawing command factorizes as:

$$\mathbb{P}_\theta[t_1 t_2 \cdots t_K | I, T] = \prod_{k=1}^K \mathbb{P}_\theta[t_k | f_\theta(I, \text{render}(T)), \{t_j\}_{j=1}^{k-1}] \quad (1)$$

where $t_1 t_2 \cdots t_K$ are the tokens in the drawing command, I is the target image, T is an execution trace, θ are the parameters of the neural network, and $f_\theta(\cdot, \cdot)$ is the image feature extractor (convolutional network). The distribution over execution traces factorizes as:

$$\mathbb{P}_\theta[T | I] = \prod_{n=1}^{|T|} \mathbb{P}_\theta[T_n | I, T_{1:(n-1)}] \times \mathbb{P}_\theta[\text{STOP} | I, T] \quad (2)$$

where $|T|$ is the length of execution trace T , and the STOP token is emitted by the network to signal that the execution trace explains the image. [Make explicit that a T_n in Equation 2 is a concise way of referring to a sequence of tokens from Equation 1?]

We train the network by sampling execution traces T and target images I for randomly generated scenes [this process ought to be explained, perhaps in supplement if it is at all detailed], and maximizing (2) with respect to θ by gradient ascent. Training does not require backpropagation across the entire sequence of drawing commands: drawing to the canvas ‘blocks’ the gradients, effectively offloading memory to an external visual store. In a sense, this model is like an autoregressive variant of AIR [3] without attention. [Somewhere (not necessarily here) you should probably cite the deep learning toolkit you used.]

This network suffices to “derender” synthetic images like those shown in Figure 3. We can perform a beam search decoding to recover what the network thinks is the most likely execution trace for images like these, recovering traces maximizing $\mathbb{P}_\theta[T | I]$. But, if the network makes a mistake (predicts an incorrect line of code), it has no way of recovering from the error. In order to derender an image with n objects, it must correctly predict n drawing commands – so its probability of success will decrease exponentially in n , assuming it has any nonzero chance of making a mistake. For added robustness as n becomes large, we treat the neural network outputs as proposals for a Sequential

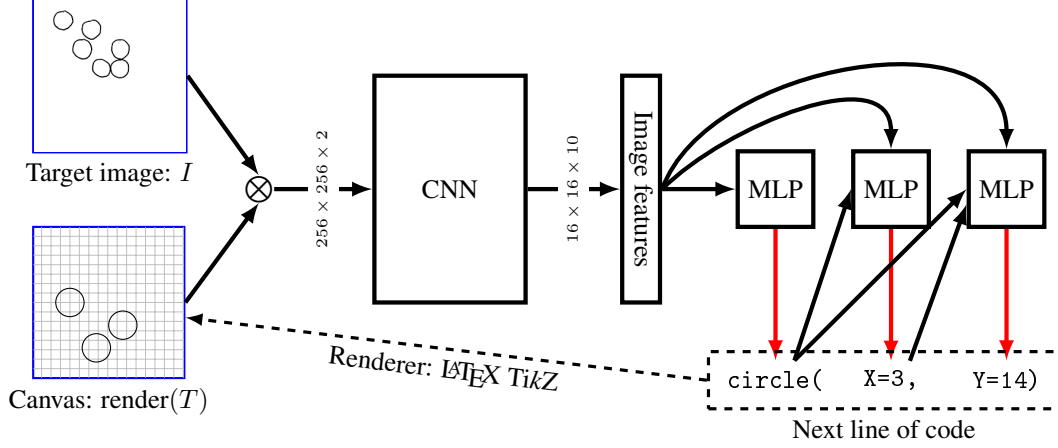


Figure 2: Our neural architecture for inferring the execution trace of a graphics program from its output. **Blue**: network inputs. **Black**: network operations. **Red**: samples from a multinomial. Typewriter font: network outputs. Renderers snapped to a 16×16 grid, illustrated in gray. [Thoughts on improving this figure: (1) Convnet diagrams typically show the sequence of layers, if possible (space might not permit it here, but those thin arrows just aren't doing it for me).]

<code>circle(x, y)</code>	Circle at (x, y)
<code>rectangle(x_1, y_1, x_2, y_2)</code>	Rectangle with corners at (x_1, y_1) & (x_2, y_2)
<code>line(x_1, y_1, x_2, y_2,</code> <code>arrow $\in \{0, 1\}$, dashed $\in \{0, 1\}$)</code>	Line from (x_1, y_1) to (x_2, y_2) , optionally with an arrow and/or dashed
<code>STOP</code>	Finishes execution trace inference

Table 1: The deep network in (2) predicts drawing commands, shown above.

117 Monte Carlo (SMC) sampling scheme [?]. For the SMC sampler, we use pixel-wise distance as
 118 a surrogate for a likelihood function. The SMC sampler is designed to produce samples from the
 119 distribution $\propto L(I|\text{render}(T))\mathbb{P}_\theta[T|I]$, where $L(\cdot|\cdot) : \text{image}^2 \rightarrow \mathcal{R}$ uses the distance between two
 120 images as a proxy for a likelihood.

121 Figure 4 compares the neural network with SMC against the neural network by itself or SMC by itself.
 122 Only the combination of the two passes a critical test of generalization: when trained on images with
 123 ≤ 8 objects, it successfully parses scenes with many more objects than the training data.

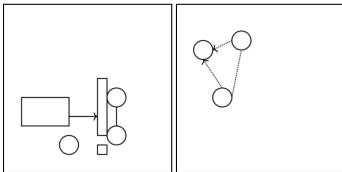


Figure 3: Network is trained to infer execution traces for randomly generated figures like the two shown above.

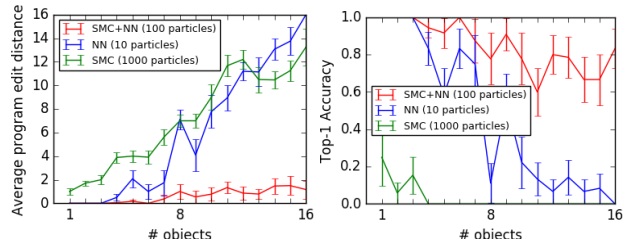


Figure 4: Using the model to parse latex output. The model is trained on diagrams with up to 8 objects. As shown above it generalizes to scenes with many more objects. Neither the stochastic search nor the neural network are sufficient on their own. # particles varies by model: we compare the models *with equal runtime* (≈ 1 sec/object)

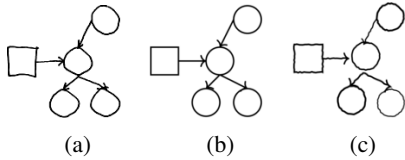


Figure 5: (a): a hand drawing. (b): Rendering of the trace our model infers for (a). We can generalize to hand drawings like these because we train the model on images corrupted by a noise process designed to resemble the kind of noise introduced by hand drawings - see (c) for a noisy rendering of (b).

7

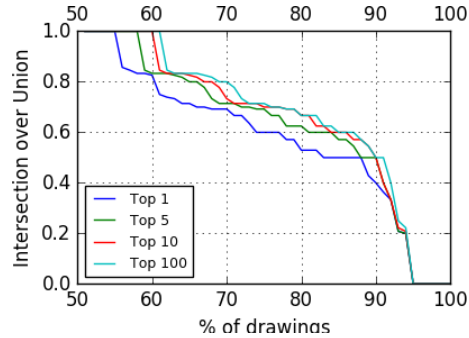


Figure 6: How close are the model’s outputs to the ground truth on hand drawings, as we consider larger sets of samples (1, 5, 10, 100 samples)? Distance to the ground truth trace is measured by the intersection over union of predicted vs. ground truth traces (sets of drawing commands).

124 3.1 Generalizing to hand drawings

125 A practical application of our neural network is the automatic conversion of hand drawings into a
 126 subset of \LaTeX . We train the model to generalize to hand drawings by introducing noise into the
 127 renderings of the training target images. We designed this noise process to introduce the kinds of
 128 variations found in hand drawings (Figure 7). [Process should be described (in supplement)] Our
 129 neurally-guided SMC procedure used pixel-wise distance as a surrogate for a likelihood function
 130 ($L(\cdot|\cdot)$ in section 3). But pixel-wise distance fares poorly on hand drawings, which never exactly
 131 match the model’s renders. So, for hand drawings, we *learn* a surrogate likelihood function. Our
 132 learned $L(\cdot|\cdot)$ is a convolutional network that we train to predict the distance between two traces
 133 conditioned upon their renderings [The two L ’s should have some kind of identifier e.g. a subscript
 134 to distinguish them]. Formally we train our likelihood surrogate to approximate:

$$-\log L(\text{render}(T_1)|\text{render}(T_2)) \approx |T_1 - T_2| + |T_2 - T_1| \quad (3)$$

135 [What distance metric is this? Some kind of edit distance?] We drew 100 figures by hand; see
 136 figure ???. These were drawn reasonably carefully but not perfectly. Because our model assumes that
 137 objects are snapped to a 16×16 grid, we made the drawings on graph paper.

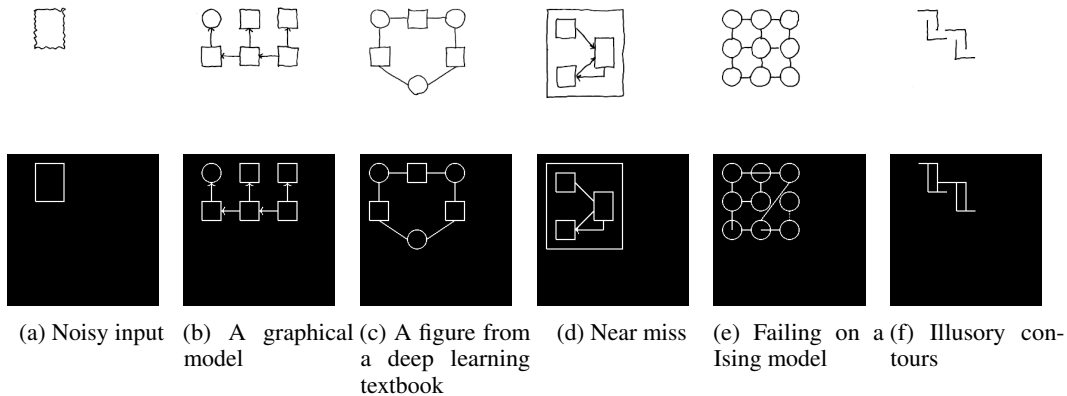


Figure 7: Example drawings above model outputs. See also Fig. 1

[Showing ‘failure cases’ (the last three above) out-of-context seems not great. Perhaps remind people in the caption that stochastic search can help correct some of these issues?]

138 For each drawing we annotated a ground truth trace, and evaluated the model by asking it to sample
 139 many candidate traces for each drawing. For 55% of the drawings the Top-1 most likely sample

140 exactly matches the ground truth; as we consider more samples the model encounters traces that are
 141 closer to the ground truth annotation (Fig. 6). Because our current model sometimes makes mistakes
 142 on hand drawings, we envision the current system working as follows: a user sketches a diagram, and
 143 the system responds by proposing a few candidate interpretations. The user could then select the one
 144 closest to their intention and edit it if necessary.

145 4 Synthesizing graphics programs from execution traces

146 Although the execution trace of a graphics program describes the parts of a scene, it fails to encode
 147 higher-level features of the image, such as repeated motifs or symmetries. A *graphics program* better
 148 describe structures like these, and we now take as our goal to synthesize simple graphics programs
 149 from their execution traces.

150 We constrain the space of allowed programs by writing down a context free grammar over a space of
 151 programs. Although it might be desirable to synthesize programs in a Turing-complete language such
 152 as Lisp or Python, a more tractable approach is to specify what in the program languages community
 153 is called a Domain Specific Language (DSL) [citation?]. Our DSL (Table 2) encodes prior knowledge
 154 of what graphics programs tend to look like.

Program	→	Command; ...; Command
Command	→	circle(Expression, Expression)
Command	→	rectangle(Expression, Expression, Expression, Expression)
Command	→	line(Expression, Expression, Expression, Expression, Boolean, Boolean)
Command	→	for($0 \leq \text{Var} < \text{Expression}$) { if ($\text{Var} > 0$) { Program }; Program }
Command	→	reflect(Axis) { Program }
Expression	→	$Z * \text{Var} + Z$
Var	→	A free (unused) variable
Z	→	an integer
Axis	→	$X = Z$
Axis	→	$Y = Z$

Table 2: Grammar over graphics programs. We allow loops (for) with conditionals (if), vertical/horizontal reflections (reflect), variables (Var) and affine transformations ($Z * \text{Var} + Z$).

155 Given the DSL and a trace T , we want to recover a program that both evaluates to T and, at the same
 156 time, is the “best” explanation of T . For example, we might prefer more general programs or, in the
 157 spirit of Occam’s razor, prefer shorter programs. We wrap these intuitions up into a cost function
 158 over programs, and seek the minimum cost program consistent with T :

$$\text{program}(T) = \arg \min_{\substack{p \in \text{DSL} \\ p \text{ evaluates to } T}} \text{cost}(p) \quad (4)$$

159 We define the cost of a program to be the number of statements it contains, where a statement is a
 160 “Command” in Table 2. We also penalize using more distant constants [awkward], which encourages
 161 reuse of coefficients. [More detail on this in supplement]

162 The constrained optimization problem in equation 4 is intractable in general, but there exist efficient-
 163 in-practice tools for finding exact solutions to program synthesis problems like these. We use the
 164 state-of-the-art Sketch tool [1]. Describing Sketch’s program synthesis algorithm is beyond the scope
 165 of this paper; see supplement. At a high level, Sketch takes as input a space of programs, along with
 166 a specification of the program’s behavior and optionally a cost function. It translates the synthesis
 167 problem into a constraint satisfaction problem, and then uses a SAT solver to find a minimum cost
 168 program satisfying the specification. In exchange for not having any guarantees on how long it will
 169 take to find a minimum cost solution, it comes with the guarantee that it will always find a globally
 170 optimal program.

171 Why synthesize a graphics program, if the execution trace already suffices to recover the objects in an
 172 image? Within our domain of hand-drawn figures, graphics program synthesis has several uses: [I’m
 173 of two minds about how these subsections should be ordered. The current ordering leads with the
 174 coolest results, which is nice. But it’s a bit...deflating?...to start with such cool results and then end

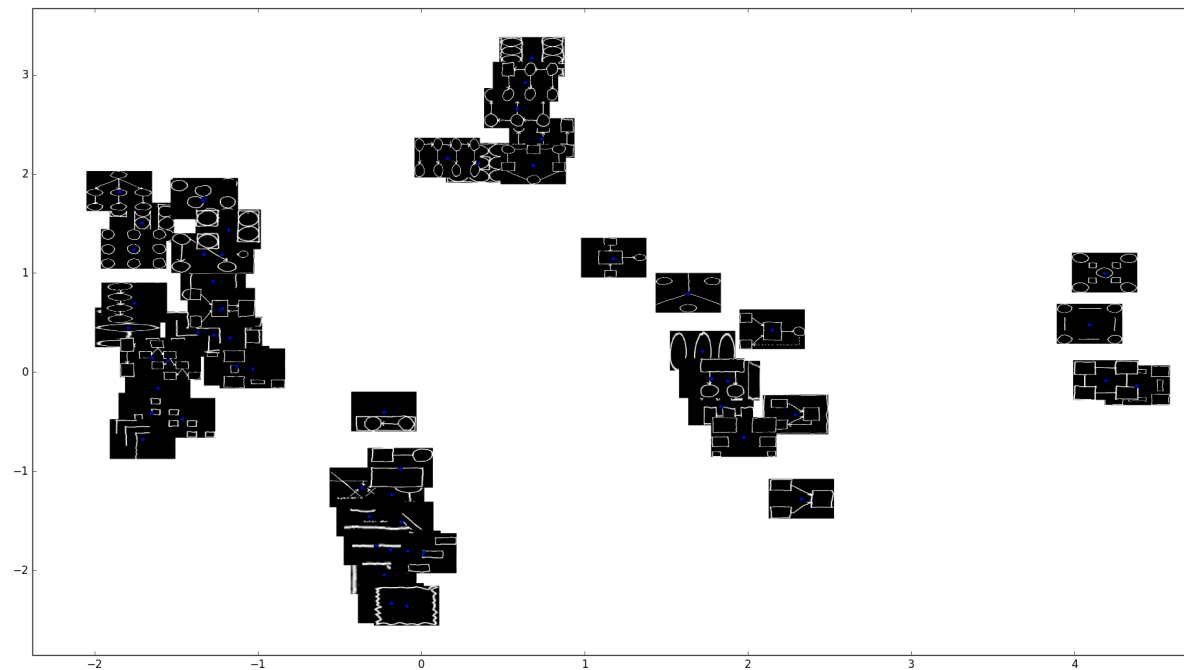
175 on the somewhat technical point of how the synthesizer can help correct parse errors. An alternative
176 would be to flip the ordering, starting with the technical point and building toward progressively
177 cooler results. This requires a bit more patience on the part of the reader, but the overall narrative
178 flow/payoff feels better. In any case, whatever ordering you pick should be the same order as these
179 applications are mentioned in the abstract.]

180 4.1 Extrapolating figures

181 Having access to the source code of a graphics program facilitates coherent, high-level edits to the
182 figure generated by that program. For example, we can change all of the circles to squares or make all
183 of the lines be dashed. We can also **extrapolate** figures by increasing the number of times that loops
184 are executed. [Pick a few of these to show off and put the rest in supplement?] Extrapolating repetitive
185 visuals patterns comes naturally to humans, and building this ability into an application is practical:
186 imagine hand drawing a repetitive graphical model structure and having the model automatically
187 induce and extend the pattern. Fig. 8 shows extrapolations of programs synthesized from ground
188 truth traces; see supplement for our full set of extrapolations.

189 4.2 Modeling similarity between figures

190 Similarity metric in program space:



191

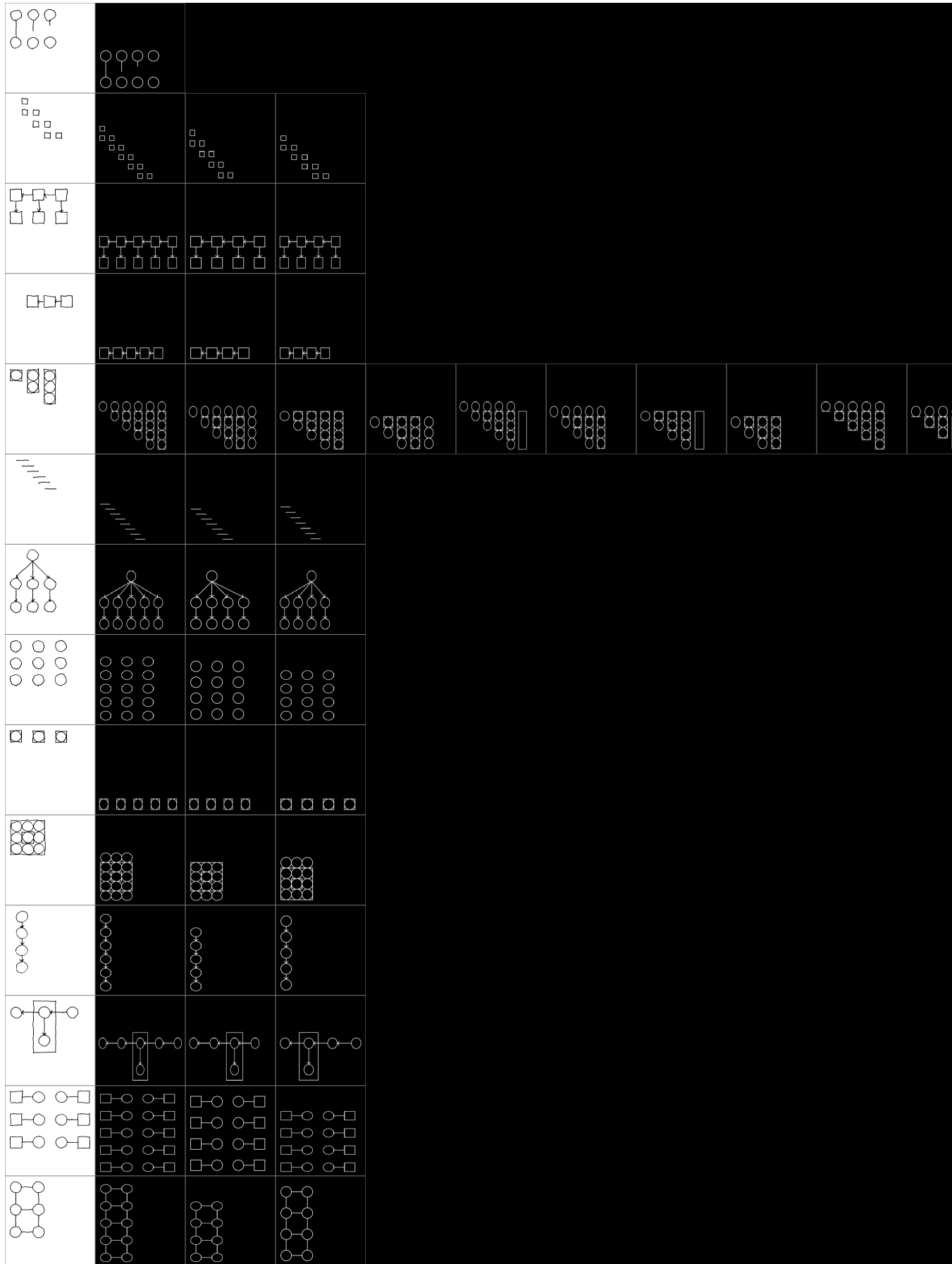
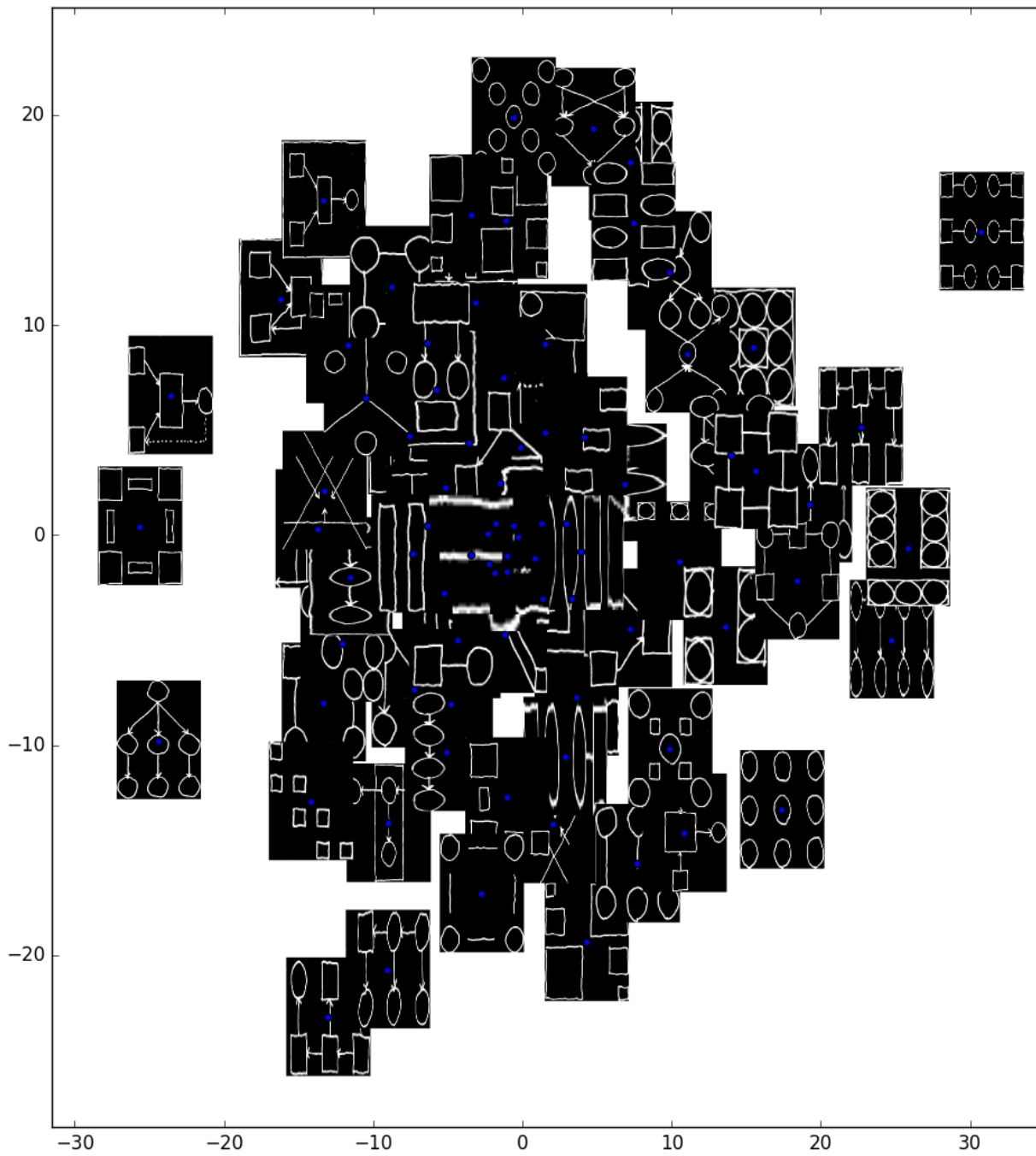


Figure 8: Left: hand drawings. Right: extrapolations produced by running different parts of different loops either forward or backward an extra iteration.



4.3 Correcting errors made by the parsing network

The program synthesizer can help correct errors from the parse proposal network by favoring execution traces which lead to more concise or general programs. For example, one generally prefers figures with perfectly aligned objects over figures whose parts are slightly misaligned – and precise alignment lends itself to short programs. Similarly, figures often have repeated parts, which the program synthesizer might be able to model as a loop or reflectional symmetry. So, in considering several candidate traces proposed by the neural network, we might prefer traces whose best programs have desirable features such as being short or having iterated structures.

Concretely, we implemented the following scheme: the neurally guided sampling scheme of section 3 for image I samples candidate traces $\mathcal{F}(I)$. Instead of predicting the most likely trace in $\mathcal{F}(I)$ according to the neural network, we can take into account the programs that best explain the traces. Writing $\hat{T}(I)$ for the trace the model predicts for image I ,

$$\hat{T}(I) = \arg \max_{T \in \mathcal{F}(I)} L(I|\text{render}(T)) \times \mathbb{P}_\beta[\text{program}(T)] \quad (5)$$

where $\mathbb{P}_\beta[\cdot]$ is a prior probability distribution over programs parameterized by β . This is equivalent to doing MAP inference in a generative model where the program is first drawn from $\mathbb{P}_\beta[\cdot]$, then the program is executed deterministically, and then we observe a noisy version of the program’s output, where L is the noise model.

Given a corpus of graphics program synthesis problems with annotated ground truth traces (i.e. (I, T) pairs), we find a maximum likelihood estimate of β :

$$\beta^* = \arg \max_{\beta} \mathbb{E} \left[\log \frac{\mathbb{P}_\beta[\text{program}(T)] \times L(I|\text{render}(T))}{\sum_{T' \in \mathcal{F}(I)} \mathbb{P}_\beta[\text{program}(T')] \times L(I|\text{render}(T'))} \right] \quad (6)$$

where the expectation is taken both over the model predictions and the (I, T) pairs in the training corpus. We define $\mathbb{P}_\beta[\cdot]$ to be a log linear distribution $\propto \exp(\beta \cdot \phi(\text{program}))$, where $\phi(\cdot)$ is a feature extractor for programs. We extract a few basic features of a program, such as its size and how many loops it has, and use these features to help predict whether a trace is the correct explanation for an image.

References

- [1] Armando Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.
- [2] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
- [3] SM Eslami, N Heess, and T Weber. Attend, infer, repeat: Fast scene understanding with generative models. arxiv preprint arxiv:1603.08575, 2016. URL <http://arxiv.org/abs/1603.08575>.
- [4] Daniel Ritchie, Anna Thomas, Pat Hanrahan, and Noah Goodman. Neurally-guided procedural models: Amortized inference for procedural graphics programs using neural networks. In *Advances In Neural Information Processing Systems*, pages 622–630, 2016.
- [5] Haibin Huang, Evangelos Kalogerakis, Ersin Yumer, and Radomir Mech. Shape synthesis from sketches via procedural models and convolutional networks. *IEEE transactions on visualization and computer graphics*, 2017.
- [6] Gen Nishida, Ignacio Garcia-Dorado, Daniel G. Aliaga, Bedrich Benes, and Adrien Bousseau. Interactive sketching of urban procedural models. *ACM Trans. Graph.*, 35(4), 2016.
- [7] Brian Hempel and Ravi Chugh. Semi-automated svg programming via direct manipulation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, UIST ’16, pages 379–390, New York, NY, USA, 2016. ACM.