

# Learning to Infer Graphics Programs from Hand-Drawn Images

Anonymous Author(s)

Affiliation

Address

email

## Abstract

We introduce a model that learns to convert simple hand drawings into graphics programs written in a subset of  $\text{\LaTeX}$ . The model combines techniques from deep learning and program synthesis. We learn a convolutional neural network that proposes plausible drawing primitives that explain an image. This set of drawing primitives is like an execution trace for a graphics program. From this trace we use program synthesis techniques to recover a graphics program with constructs like variable bindings, iterative loops, or simple kinds of conditionals. With a graphics program in hand, we can correct errors made by the deep network, cluster drawings by use of similar high-level geometric structures, and extrapolate drawings. Taken together these results are a step towards agents that induce useful, human-readable programs from perceptual input.

## 1 Introduction

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

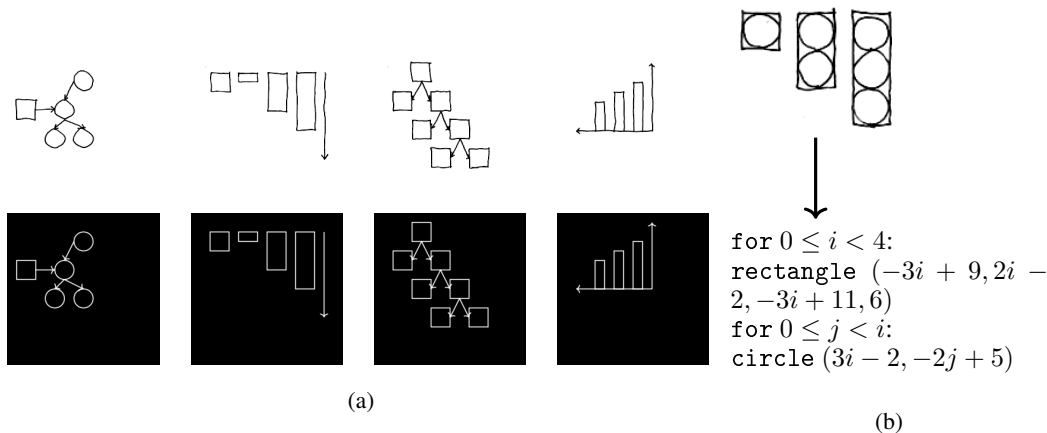


Figure 1: (a): Model learns to convert hand drawings (top) into  $\text{\LaTeX}$  (bottom). (b) Synthesizes high-level *graphics program* from hand drawing.

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

We develop a hybrid architecture for inferring graphics programs. Our approach uses a deep neural network infer an execution trace from an image; this network recovers primitive drawing operations such as lines, circles, or arrows, along with their parameters. For added robustness, we use the deep network as a proposal distribution for a stochastic search over execution traces. Section 3 describes this first stage of the architecture where we infer drawing commands from images, and explains how we handle noisy hand drawings. Finally, we use program synthesis techniques to recover the program from its trace. The program synthesizer discovers constructs such as loops and geometric operations such as reflections and affine transformations. Section 4 describes how the architecture synthesizes programs from execution traces, and how those programs are used for measuring similarity between hand drawings, extrapolating figures, and correcting errors made by the deep network.

## 2 Related work

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.

The CogSketch [8] system also aims to have a high-level understanding of hand-drawn figures. Their primary goal is cognitive modeling (eg, they apply their system to solving IQ-test style visual reasoning problems), whereas we are interested in building an automated AI application (eg, in our system the user need not annotate which strokes correspond to which shapes; our neural network produces something equivalent to the annotations). A key similarity however is that both CogSketch

<code>circle(<math>x, y</math>)</code>	Circle at $(x, y)$
<code>rectangle(<math>x_1, y_1, x_2, y_2</math>)</code>	Rectangle with corners at $(x_1, y_1)$ & $(x_2, y_2)$
<code>line(<math>x_1, y_1, x_2, y_2</math>, arrow <math>\in \{0, 1\}</math>, dashed <math>\in \{0, 1\}</math>)</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.

and our system have as a goal to make it easier to produce nice-looking figures. Unsupervised Program Synthesis [9] is a related framework which was also applied to geometric reasoning problems. The goals of [9] were cognitive modeling, and they applied their technique to synthetic scenes used in human behavioral studies.

### 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  $256 \times 256$  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,  $\theta$  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  $\theta$  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.]

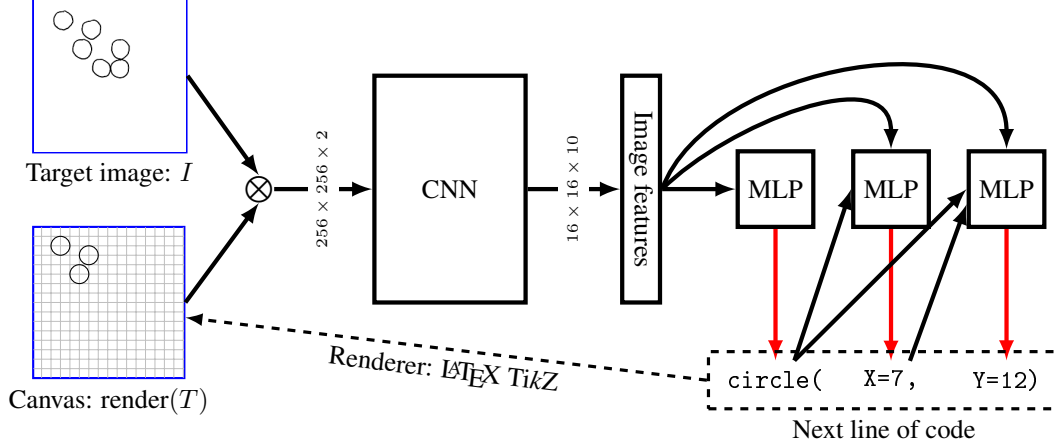


Figure 2: Our neural architecture for inferring the execution trace of a graphics program from its output. **Blue**: network inputs. **Black**: network operations. **Red**: samples from a multinomial. Typewriter font: network outputs. Renders snapped to a  $16 \times 16$  grid, illustrated in gray.

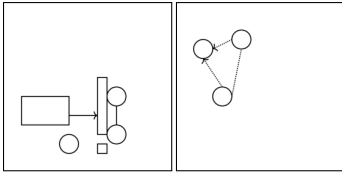


Figure 3: Network is trained to infer execution traces for randomly generated figures like the two shown above.

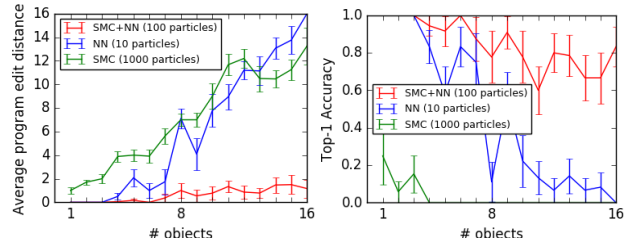


Figure 4: Using the model to parse latex output. The model is trained on diagrams with up to 8 objects. As shown above it generalizes to scenes with many more objects. Neither the stochastic search nor the neural network are sufficient on their own. # particles varies by model: we compare the models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

109 This network suffices to “derender” synthetic images like those shown in Figure 3. We can perform a  
 110 beam search decoding to recover what the network thinks is the most likely execution trace for images  
 111 like these, recovering traces maximizing  $\mathbb{P}_\theta[T|I]$ . But, if the network makes a mistake (predicts an  
 112 incorrect line of code), it has no way of recovering from the error. In order to derender an image  
 113 with  $n$  objects, it must correctly predict  $n$  drawing commands – so its probability of success will  
 114 decrease exponentially in  $n$ , assuming it has any nonzero chance of making a mistake. For added  
 115 robustness as  $n$  becomes large, we treat the neural network outputs as proposals for a Sequential  
 116 Monte Carlo (SMC) sampling scheme [10]. For the SMC sampler, we use pixel-wise distance as  
 117 a surrogate for a likelihood function. The SMC sampler is designed to produce samples from the  
 118 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  
 119 images as a proxy for a likelihood.

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

### 123 3.1 Generalizing to hand drawings

124 A practical application of our neural network is the automatic conversion of hand drawings into a  
 125 subset of  $\text{\LaTeX}$ . We train the model to generalize to hand drawings by introducing noise into the  
 126 renderings of the training target images. We designed this noise process to introduce the kinds of  
 127 variations found in hand drawings (Figure 5; see supplement for details). Our neurally-guided SMC  
 128 procedure used pixel-wise distance as a surrogate for a likelihood function ( $L(\cdot|\cdot)$  in section 3). But

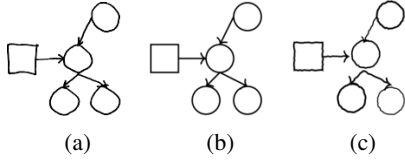


Figure 5: (a): a hand drawing. (b): Rendering of the trace our model infers for (a). We can generalize to hand drawings like these because we train the model on images corrupted by a noise process designed to resemble the kind of noise introduced by hand drawings - see (c) for a noisy rendering of (b).

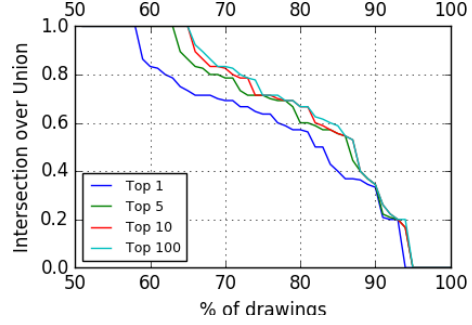


Figure 6: How close are the model’s outputs to the ground truth on hand drawings, as we consider larger sets of samples (1, 5, 10, 100 samples)? Distance to the ground truth trace is measured by the intersection over union of predicted vs. ground truth traces (sets of drawing commands).

129 pixel-wise distance fares poorly on hand drawings, which never exactly match the model’s renders.  
 130 So, for hand drawings, we *learn* a surrogate likelihood function,  $L_{\text{learned}}(\cdot|\cdot)$ . The density  $L_{\text{learned}}(\cdot|\cdot)$   
 131 is predicted by a convolutional network that we train to predict the distance between two traces  
 132 conditioned upon their renderings. We train our likelihood surrogate to approximate the symmetric  
 133 difference, which is the number of drawing commands by which two traces differ:

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

134 Intuitively, Eq. 3 says that  $L_{\text{learned}}(\cdot|\cdot)$  approximates the distance between the trace we want and the  
 135 trace we have so far. Pixel-wise distance metrics are sensitive to the fine details of how and exactly  
 136 where arrows, dashes, and corners are drawn – but we wish to be invariant to these details. So, we  
 137 learn a distance metric over images that approximates the distance metric in the search space over  
 138 traces.

139 We drew 100 figures by hand; see figure 7. These were drawn reasonably carefully but not perfectly.  
 140 Because our model assumes that objects are snapped to a  $16 \times 16$  grid, we made the drawings on  
 141 graph paper.

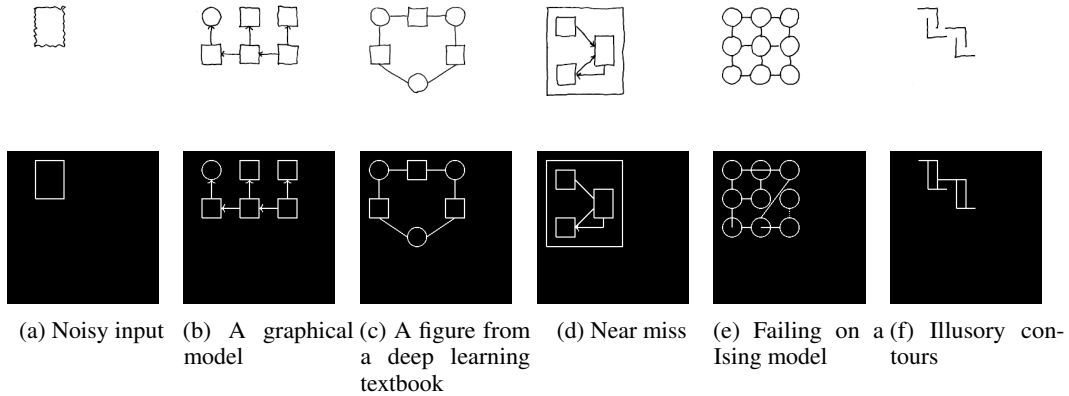


Figure 7: Example drawings above model outputs. See also Fig. 1. Stochastic search (SMC) can help correct for these errors, as can the program synthesizer (Section 4.1)

142 For each drawing we annotated a ground truth trace, and evaluated the model by asking it to sample  
 143 many candidate traces for each drawing. For 57% of the drawings the Top-1 most likely sample  
 144 exactly matches the ground truth; as we consider more samples the model encounters traces that are  
 145 closer to the ground truth annotation (Fig. 6). Because our current model sometimes makes mistakes

on hand drawings, we envision the current system working as follows: a user sketches a diagram, and the system responds by proposing a few candidate interpretations. The user could then select the one closest to their intention and edit it if necessary.

## 4 Synthesizing graphics programs from execution traces

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

We constrain the space of allowed programs by writing down a context free grammar over a space of programs. Although it might be desirable to synthesize programs in a Turing-complete language such as Lisp or Python, a more tractable approach is to specify what in the program languages community is called a Domain Specific Language (DSL) [11]. Our DSL (Table 2) encodes prior knowledge 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	→	$\mathcal{Z} * \text{Var} + \mathcal{Z}$
Var	→	A free (unused) variable
$\mathcal{Z}$	→	an integer
Axis	→	X = $\mathcal{Z}$
Axis	→	Y = $\mathcal{Z}$

---

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

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

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

We define the cost of a program to be the number of statements it contains, where a statement is a “Command” in Table 2. We also penalize using many different numerical constants; see supplement.

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

Why synthesize a graphics program, if the execution trace already suffices to recover the objects in an image? Within our domain of hand-drawn figures, graphics program synthesis has several uses:

### 4.1 Correcting errors made by the neural network

The program synthesizer can help correct errors from the execution trace proposal network by favoring execution traces which lead to more concise or general programs. For example, one generally prefers

179 figures with perfectly aligned objects over figures whose parts are slightly misaligned – and precise  
 180 alignment lends itself to short programs. Similarly, figures often have repeated parts, which the  
 181 program synthesizer might be able to model as a loop or reflectional symmetry. So, in considering  
 182 several candidate traces proposed by the neural network, we might prefer traces whose best programs  
 183 have desirable features such as being short or having iterated structures.

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

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

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

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

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

194 where the expectation is taken both over the model predictions and the  $(I, T)$  pairs in the training  
 195 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  
 196 extractor for programs. We extract a few basic features of a program, such as its size and how many  
 197 loops it has, and use these features to help predict whether a trace is the correct explanation for an  
 198 image.

## 199 4.2 Modeling similarity between drawings

200 Modeling an image using a program opens up new ways of measuring similarity between drawings.  
 201 For example, we might say that two drawings are similar if they both contain repetitions of length 4,  
 202 or if they share the same reflectional symmetry, or if they are both organized according to a grid-like  
 203 structure.

204 We measure the similarity between two drawings by extracting features of the best programs that  
 205 describe them. Here the features we use are just counts of the number of times that different  
 206 components in the DSL were used (Table 2). We project these features down to a 2-dimensional  
 207 subspace using nonnegative matrix factorization (NMF: [12]); see Fig.8. One could use many  
 208 alternative similarity metrics between drawings which would capture pixel-level or object-level  
 209 similarities while missing high-level geometric similarities. For example, we can use our learned  
 210 distance metric between execution traces,  $L_{\text{learned}}(\cdot|\cdot)$ . Projecting these distances to a 2-dimensional  
 211 subspace using multidimensional scaling (MDS: [13]) reveals similarities between the objects in the  
 212 drawings, while missing similarities at the level of the program.

## 213 4.3 Extrapolating figures

214 Having access to the source code of a graphics program facilitates coherent, high-level edits to the  
 215 figure generated by that program. For example, we can change all of the circles to squares or make all  
 216 of the lines be dashed. We can also **extrapolate** figures by increasing the number of times that loops  
 217 are executed. Extrapolating repetitive visual patterns comes naturally to humans, and building this  
 218 ability into an application is practical: imagine hand drawing a repetitive graphical model structure  
 219 and having our system automatically induce and extend the pattern. Fig. 10 shows extrapolations of  
 220 programs synthesized from ground truth traces; see supplement for our full set of extrapolations.

## 221 5 Conclusion

222 We have presented a system for inferring graphics programs which generate  $\text{\LaTeX}$ -style figures from  
 223 hand-drawn images. The system uses a combination of deep neural networks and stochastic search to

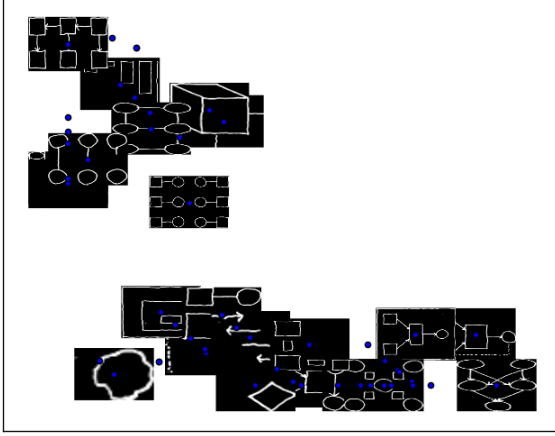


Figure 8: NMF on features of the programs that were synthesized for each image. Horizontal component roughly corresponds to “symmetry” while vertical component roughly corresponds to “loopyness”, with images on the diagonal having both of these.

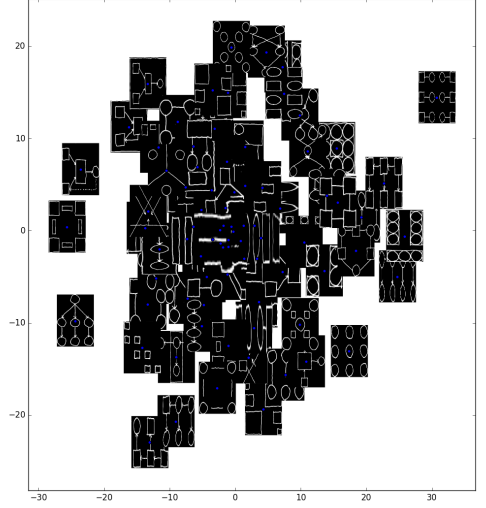


Figure 9: MDS on drawings using the learned distance metric,  $L_{\text{learned}}(\cdot|\cdot)$ . Drawings with similar looking parts in similar locations are clustered together.

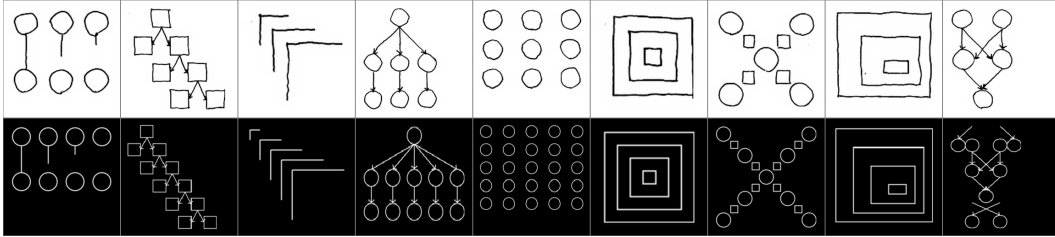


Figure 10: Top: hand drawings. Bottom: extrapolations produced by running loops for extra iterations. Rightmost pair is an illustrative failure case.

224 parse drawings into symbolic execution traces; it then feeds these traces to a general-purpose program  
 225 synthesis engine to infer a structured graphics program. We evaluated our model’s performance at  
 226 parsing novel images, and we demonstrated its ability to extrapolate from provided drawings and to  
 227 organize them according to high-level geometric features.

228 There are many directions for future work. In the parsing phase, the proposal network currently  
 229 samples positional variables on a discrete grid. More general types of drawings could be supported  
 230 by instead sampling from continuous distributions, e.g. using Mixture Density Networks [14]. The  
 231 proposal network also currently handles only a very small subset of  $\text{\LaTeX}$  drawing commands, though  
 232 there is no reason that it could not be extended to handle more with a higher-capacity network.  
 233 Exploring more sophisticated network architectures, including ones that utilize attention, could  
 234 also help correct some of the errors the network makes. In the synthesis phase, a more expressive  
 235 DSL—including subroutines, recursion, and symmetry groups beyond reflections—would allow the  
 236 system to effectively model a wider variety of graphical phenomena. The synthesizer itself could  
 237 also be the subject of future work: the system currently uses the general-purpose Sketch synthesizer,  
 238 which can take minutes to hours to run, whereas program synthesizers which are custom-built for  
 239 special problem domains can run much faster or even interactively [15].

240 In the not-too-distant future, we believe it should be possible to produce professional-looking figures  
 241 just by drawing them and then letting an artificially-intelligent agent write the corresponding code.  
 242 More generally, we believe that the two-phase system we have proposed—parsing into execution



traces, then searching for a low-cost symbolic program which generates those traces—may be a useful paradigm for other domains in which agents must programmatically reason about noisy perceptual input.

## 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:..., 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.
- [8] Kenneth Forbus, Jeffrey Usher, Andrew Lovett, Kate Lockwood, and Jon Wetzell. Cogsketch: Sketch understanding for cognitive science research and for education. *Topics in Cognitive Science*, 3(4):648–666, 2011.
- [9] Kevin Ellis, Armando Solar-Lezama, and Josh Tenenbaum. Unsupervised learning by program synthesis. In *Advances in Neural Information Processing Systems*, pages 973–981, 2015.
- [10] Arnaud Doucet, Nando De Freitas, and Neil Gordon, editors. *Sequential Monte Carlo Methods in Practice*. Springer, 2001.
- [11] Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. *ACM SIGPLAN Notices*, 50(10):107–126, 2015.
- [12] Daniel D Lee and H Sebastian Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, 401(6755):788–791, 1999.
- [13] Michael AA Cox and Trevor F Cox. Multidimensional scaling. *Handbook of data visualization*, pages 315–347, 2008.
- [14] Christopher M. Bishop. Mixture Density Networks. Technical report, 1994.
- [15] Vu Le and Sumit Gulwani. Flashextract: a framework for data extraction by examples. In *ACM SIGPLAN Notices*, volume 49, pages 542–553. ACM, 2014.