

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. This set of drawing
 5 primitives is like an execution trace for a graphics program. From this trace we use
 6 program synthesis techniques to recover a graphics program with constructs such as
 7 variable bindings, iterative loops, or simple kinds of conditionals. With a graphics
 8 program in hand, we can correct errors made by the deep network, cluster drawings
 9 by use of similar high-level geometric structures, and extrapolate drawings. Taken
 10 together these results are a step towards agents that induce useful, human-readable
 11 programs from perceptual input.

1 Introduction

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

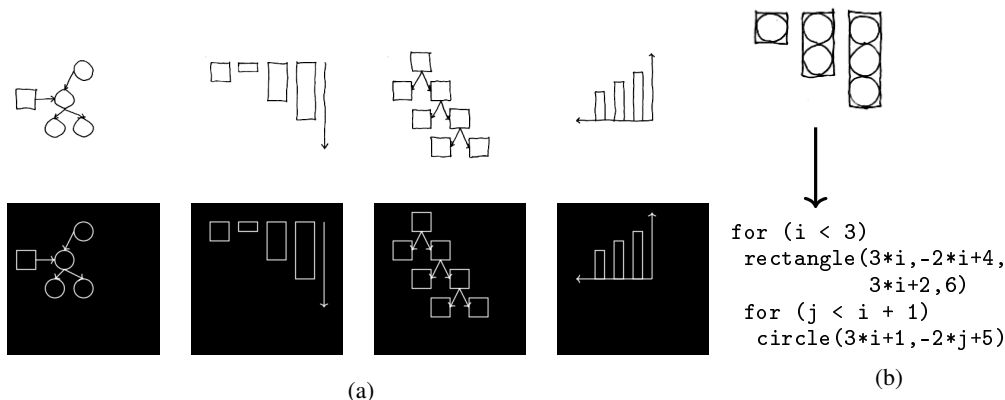


Figure 1: (a): Model learns to convert hand drawings (top) into \LaTeX (bottom). (b) Synthesizes high-level *graphics program* from hand drawing.

18 The key observation behind our work is that generating a programmatic representation from an image
19 of a diagram actually involves two distinct steps that require different technical approaches. The first
20 step involves identifying the components such as rectangles, lines and arrows that make up the image.
21 The second step involves identifying the high-level structure in how the components were drawn. In
22 Figure 1(b), it means identifying a pattern in how the circles and rectangles are being drawn that is
23 best described with two nested loops, and which can easily be extrapolated to a bigger diagram.

24 We present a hybrid architecture for inferring graphics programs that is structured around these
25 two steps. For the first step, our approach uses a deep network to infer a sequence of primitive
26 shape-drawing commands that generates an image similar to the observed image. We refer to this
27 sequence as an *execution trace*, since it corresponds to the sequence of primitive commands that
28 a program would have issued but lacks the high-level structure that determines how the program
29 decided to issue them. For added robustness, we train the network to produce a proposal distribution
30 of the most likely traces and use stochastic search to find the trace that best matches the input. The
31 network is trained from an automatically-generated corpus of synthetic image-generating programs.

32 The second step involves generating a high-level program capable of producing the program trace
33 identified by the first phase. This paper argues that this stage is best achieved by *constraint-based*
34 *program synthesis* [1]. Without the need for training, the program synthesizer can search the space
35 of possible programs for one capable of producing the desired trace – inducing structures like
36 symmetries, repetition, or conditional branches.

37 2 Related work

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

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

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

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

65 The CogSketch [8] system also aims to have a high-level understanding of hand-drawn figures. Their
66 primary goal is cognitive modeling (they apply their system to solving IQ-test style visual reasoning
67 problems), whereas we are interested in building an automated AI application (e.g. in our system
68 the user need not annotate which strokes correspond to which shapes; our neural network produces
69 something equivalent to the annotations). A key similarity however is that both CogSketch and our
70 system have as a goal to make it easier to produce nice-looking figures. Unsupervised Program

71 Synthesis [9] is a related framework which was also applied to geometric reasoning problems. The
 72 goals of [9] were cognitive modeling, and they applied their technique to synthetic scenes used in
 73 human behavioral studies.

74 3 Neural architecture for inferring drawing execution traces

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

82 Figure 2 illustrates this architecture. We first pass a 256×256 target image and a rendering of the
 83 trace so far (encoded as a two-channel image) to a convolutional network. Given the features extracted
 84 by the convnet, a multilayer perceptron then predicts a distribution over the next drawing command
 85 to add to the trace; see Table 1. We predict the drawing command token-by-token, conditioning
 86 each token both on the image features and on the previously generated tokens. For example, the
 87 network first decides to emit the `circle` token conditioned on the image features, then it emits the
 88 x coordinate of the circle conditioned on the image features and the `circle` token, and finally it
 89 predicts the y coordinate of the circle conditioned on the image features, the `circle` token, and
 90 the x coordinate. See supplement for the full details of the architecture, which we implemented in
 91 Tensorflow [10]. The distribution over the next drawing command factorizes as:

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

92 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,
 93 θ are the parameters of the neural network, and $f_\theta(\cdot, \cdot)$ is the image feature extractor (convolutional
 94 network). The distribution over execution traces factorizes as:

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

95 where $|T|$ is the length of execution trace T , the subscripts on T index drawing commands within the
 96 trace, and the `STOP` token is emitted by the network to signal that the trace explains the image.

97 We train the network by sampling execution traces T and target images I for randomly generated
 98 scenes and maximizing (2) with respect to θ by gradient ascent. Training does not require back-
 99 propagation across the entire sequence of drawing commands: drawing to the canvas ‘blocks’ the
 100 gradients, effectively offloading memory to an external visual store. In a sense, this model is like an
 101 autoregressive variant of AIR [3] without attention. We trained the network on 10^5 scenes, which
 102 takes a little less than a day on an Nvidia TitanX GPU.

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

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

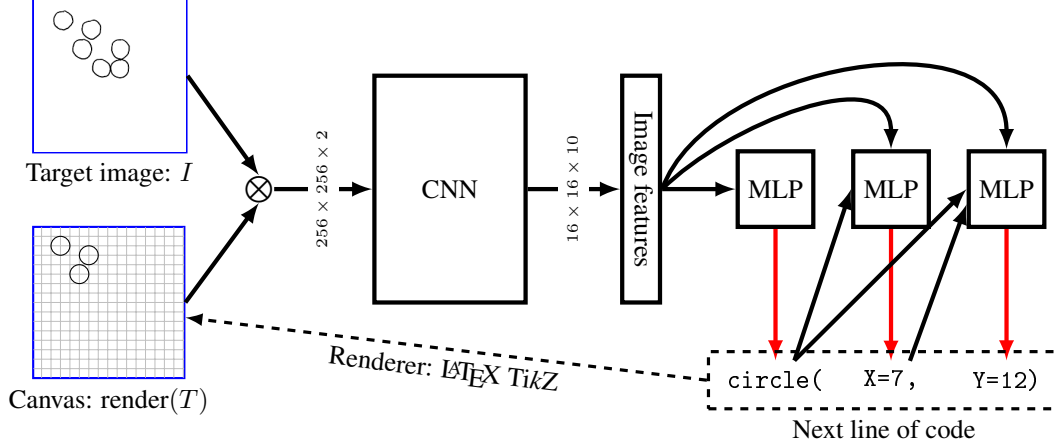


Figure 2: Our neural architecture for inferring the execution trace of a graphics program from its output. **Blue**: network inputs. **Black**: network operations. **Red**: samples from a multinomial. Typewriter font: network outputs. Renders snapped to a 16×16 grid, illustrated in gray.

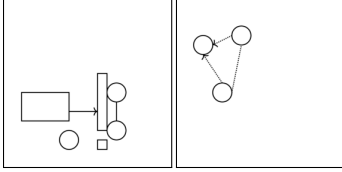


Figure 3: Network is trained to infer execution traces for randomly generated scenes like the two shown above. See supplement for details of the training data generation.

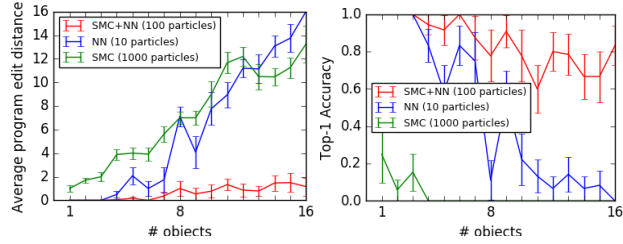


Figure 4: Using the model to parse latex output. The model is trained on diagrams with up to 8 objects. As shown above it generalizes to scenes with many more objects. Neither the stochastic search nor the neural network are sufficient on their own. # particles varies by model: we compare the models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

robustness as n becomes large, we treat the neural network outputs as proposals for a Sequential Monte Carlo (SMC) sampling scheme [11]. For the SMC sampler, we use pixel-wise distance as a surrogate for a likelihood function. The SMC sampler is designed to produce samples from the 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 images as a proxy for a likelihood. Figure 4 compares the neural network with SMC against the neural network by itself or SMC by itself. Only the combination of the two passes a critical test of generalization: when trained on images with ≤ 8 objects, it successfully parses scenes with many more objects than the training data.

3.1 Generalizing to hand drawings

A practical application of our neural network is the automatic conversion of hand drawings into a subset of \LaTeX . We train the model to generalize to hand drawings by introducing noise into the renderings of the training target images. We designed this noise process to introduce the kinds of variations found in hand drawings (Figure 5; see supplement for details). Our neurally-guided SMC procedure used pixel-wise distance as a surrogate for a likelihood function ($L(\cdot|\cdot)$ in section 3). But pixel-wise distance fares poorly on hand drawings, which never exactly match the model’s renders. So, for hand drawings, we *learn* a surrogate likelihood function, $L_{\text{learned}}(\cdot|\cdot)$. The density $L_{\text{learned}}(\cdot|\cdot)$ is predicted by a convolutional network that we train to predict the distance between two traces conditioned upon their renderings. We train our likelihood surrogate to approximate the symmetric difference, which is the number of drawing commands by which two traces differ:

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

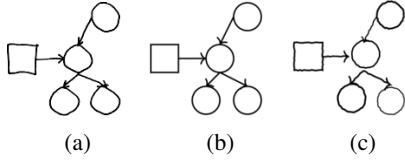


Figure 5: (a): a hand drawing. (b): Rendering of the trace our model infers for (a). We can generalize to hand drawings like these because we train the model on images corrupted by a noise process designed to resemble the kind of noise introduced by hand drawings - see (c) for a noisy rendering of (b).

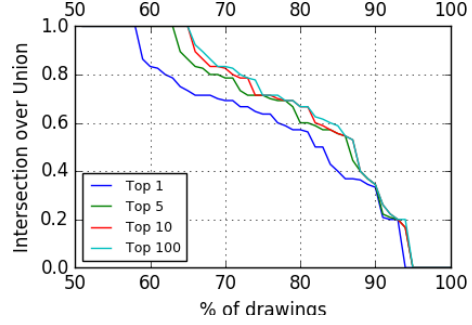


Figure 6: How close are the model’s outputs to the ground truth on hand drawings, as we consider larger sets of samples (1,5,10,100)? Distance to ground truth trace measured by the intersection over union of predicted vs. ground truth traces (sets of drawing commands).

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

We drew 100 figures by hand; see figure 7. These were drawn carefully but not perfectly. Our model assumes that objects are snapped to a 16×16 grid, so we made the drawings on graph paper.

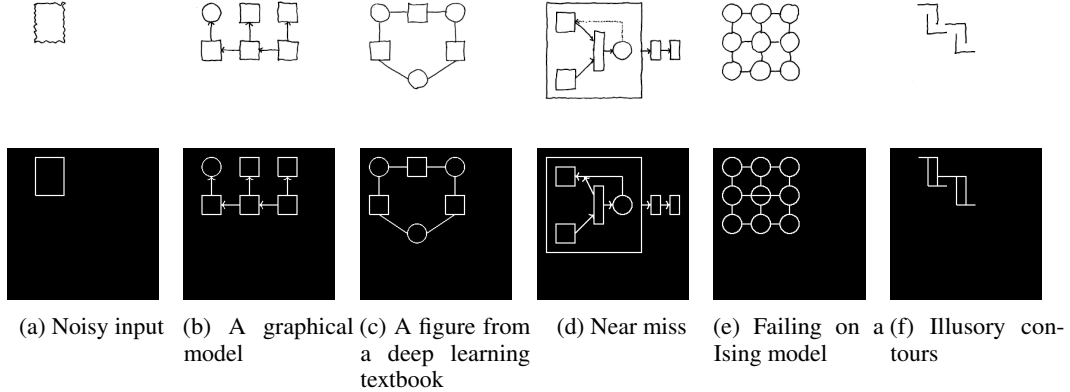


Figure 7: Example drawings above model outputs. See also Fig. 1. Stochastic search (SMC) can help correct for these errors, as can the program synthesizer (Section 4.1)

For each drawing we annotated a ground truth trace and asked the neurally guided SMC sampler to produce many candidate traces for each drawing. For 58% of the drawings, the Top-1 most likely sample exactly matches the ground truth; with more samples, the model finds traces that are closer to the ground truth annotation (Fig. 6). Because our current model sometimes makes mistakes, we envision it 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

While the execution trace of a graphics program describes the contents of a scene, it does not encode higher-level features of the image, such as repeated motifs or symmetries. A *graphics program* better describe such structures; we seek to synthesize simple graphics graphics from their execution traces.

145 We constrain the space of allowed programs by writing down a context free grammar over a space of
 146 programs. Although it might be desirable to synthesize programs in a Turing-complete language such
 147 as Lisp or Python, a more tractable approach is to specify what in the program languages community
 148 is called a Domain Specific Language (DSL) [12]. Our DSL (Table 2) encodes prior knowledge of
 149 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	→	$X = \mathcal{Z} \mid Y = \mathcal{Z}$

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

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

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

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

156 The constrained optimization problem in Equation 4 is intractable in general, but there exist efficient-
 157 in-practice tools for finding exact solutions to such program synthesis problems. We use the state-of-
 158 the-art Sketch tool [1]. Describing Sketch’s program synthesis algorithm is beyond the scope of this
 159 paper; see [1]. At a high level, Sketch takes as input a space of programs, along with a specification
 160 of the program’s behavior and optionally a cost function. It translates the synthesis problem into a
 161 constraint satisfaction problem and then uses a SAT solver to find a minimum-cost program satisfying
 162 the specification. In exchange for having no guarantee on the time required to find a minimum cost
 163 solution, it comes with the guarantee that it will always find a globally optimal program.

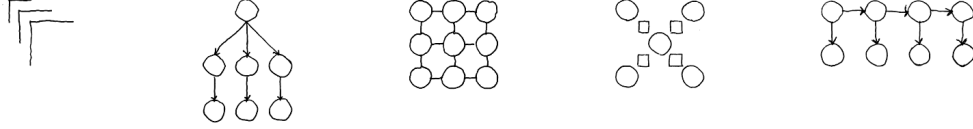
164 Why synthesize a graphics program, if the execution trace already suffices to recover the objects in
 165 an image? Within our domain of hand-drawn figures, graphics program synthesis has several uses:

166 4.1 Correcting errors made by the neural network

167 The program synthesizer can correct errors from the execution trace proposal network by favoring
 168 traces which lead to more concise or general programs. For example, figures with perfectly aligned
 169 objects are preferable to figures whose parts are slightly misaligned, and precise alignment lends itself
 170 to short programs. Concretely, we estimated a prior over programs. Then, given the few most likely
 171 traces output by the neurally guided sampler, we reranked them according to the prior probability of
 172 their programs. Our sampler could only do better on 7 drawings by looking at the Top-100 sampled
 173 traces (see Fig. 6), precluding a statistically significant analysis of how much learning a prior over
 174 programs could help correct errors. But, learning this prior does sometimes help correct mistakes
 175 made by the neural network, and also occasionally introduces mistakes of its own; see Fig. 9 for a
 176 representative example of the kinds of corrections that it makes. See supplement for details.

177 4.2 Modeling similarity between drawings

178 Modeling drawings using programs opens up new ways to measure similarity between them. For
 179 example, we might say that two drawings are similar if they both contain loops of length 4, or if they
 180 share a reflectional symmetry, or if they are both organized according to a grid-like structure.



```

for(i<3){
  line(i,-1*i+6,
2*i+2,-1*i+6)
  line(i,-2*i+4,
i,-1*i+6)
  circle(4,10)
  for(i<3){
    circle(-3*i+7,5)
    circle(-3*i+7,1)
    line(-3*i+7,4,
-3*i+7,2,
arrow=True)
    line(4,9,
-3*i+7,6,
arrow=True)
  }
  for(i<3)
    for(j<3)
      if(j>0)
        line(-3*j+8,
-3*i+7,-3*j+9,
-3*i+7)
        line(-3*i+7,-3*j+8,
-3*i+7,
-3*j+9)
        circle(-3*j+7,-3*i+7)
  reflect(y=8){
    for(i<3){
      for(i<3){
        if(i>0){
          rectangle(3*i-1,
2,3*i,3)}
          circle(3*i+1,
3*i+1)
        }
      }
    }
  }
  for(i<4){
    line(-4*i+13,4,
-4*i+13,2,
arrow=True)
    for(j<3){
      if(j>0){
        circle(-4*i+13,
4*j+-3)}
        line(-4*j+10,5,
-4*j+12,5,
arrow=True)
      }
    }
  }
}

```

Figure 8: Example drawings (top) and programs synthesized from their ground truth traces (bottom). Note the nested loops in the Ising model (middle), special case conditionals for the HMM (rightmost), combination of symmetry and iteration in middle left, and affine transformation in the leftmost figure.

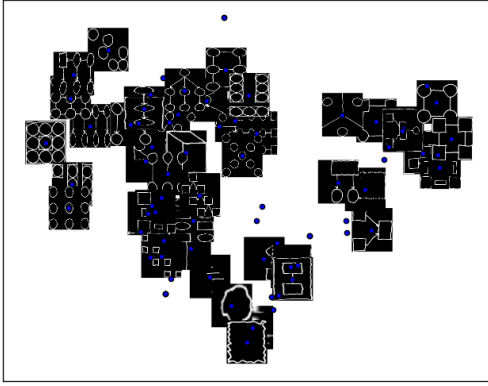


Figure 10: PCA on features of the programs that were synthesized for each drawing. Symmetric figures cluster to the right; “loopy” figures cluster to the left.

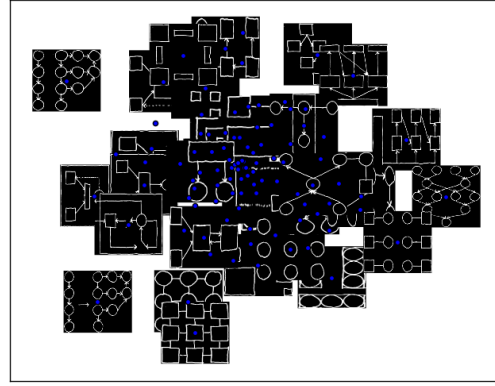


Figure 11: MDS on drawings using the learned distance metric, $L_{\text{learned}}(\cdot|\cdot)$. Drawings with similar looking parts in similar locations are clustered together.

181 We measure the similarity between two drawings by ex-
 182 tracting features of the best programs that describe them.
 183 Our features are counts of the number of times that differ-
 184 ent components in the DSL were used (Table 2). We project
 185 these features down to a 2-dimensional subspace using pri-
 186 mary component analysis (PCA); see Fig.10. One could
 187 use many alternative similarity metrics between drawings
 188 which would capture pixel-level while missing high-level
 189 geometric similarities. We used our learned distance metric
 190 between execution traces, $L_{\text{learned}}(\cdot|\cdot)$, and projected to a
 191 2-dimensional subspace using multidimensional scaling
 192 (MDS: [14]). This reveals similarities between the objects
 193 in the drawings, while missing similarities at the level of
 194 the program.

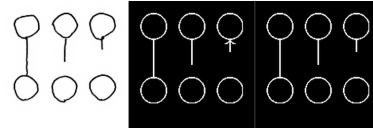


Figure 9: Left: hand drawing. Center: interpretation favored by the deep network. Right: interpretation favored after learning a prior over programs. Our learned prior favors shorter, simpler programs, thus continuing the pattern of not having an arrow is preferred.

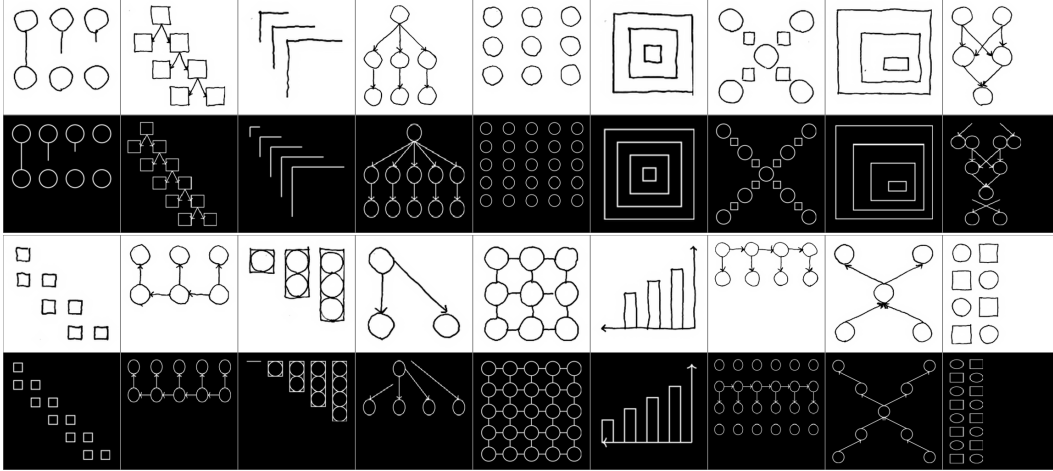


Figure 12: Top, white: hand drawings. Bottom, black: extrapolations produced by running loops for extra iterations.

4.3 Extrapolating figures

Having access to the source code of a graphics program facilitates coherent, high-level edits to the figure generated by that program. For example, we can change all of the circles to squares or make all of the lines be dashed. We can also **extrapolate** figures by increasing the number of times that loops are executed. Extrapolating repetitive visual patterns comes naturally to humans, and building this ability into an application is practical: imagine hand drawing a repetitive graphical model structure and having our system automatically induce and extend the pattern. Fig. 12 shows extrapolations of programs synthesized from ground truth traces; see supplement for our full set of extrapolations.

5 Conclusion

We have presented a system for inferring graphics programs which generate \LaTeX -style figures from hand-drawn images. The system uses a combination of deep neural networks and stochastic search to parse drawings into symbolic execution traces; it then feeds these traces to a general-purpose program synthesis engine to infer a structured graphics program. We evaluated our model’s performance at parsing novel images, and we demonstrated its ability to extrapolate from provided drawings and to organize them according to high-level geometric features.

There are many directions for future work. In the parsing phase, the proposal network currently samples positional variables on a discrete grid. More general types of drawings could be supported by instead sampling from continuous distributions, e.g. using Mixture Density Networks [15]. The proposal network also currently handles only a very small subset of \LaTeX drawing commands, though there is no reason that it could not be extended to handle more with a higher-capacity network. Exploring more sophisticated network architectures, including ones that utilize attention [16], could help correct some of the errors the network makes. In the synthesis phase, a more expressive DSL—including subroutines, recursion, and symmetry groups beyond reflections—would allow the system to effectively model a wider variety of graphical phenomena. The synthesizer itself could also be the subject of future work: the system currently uses the general-purpose Sketch synthesizer, which can take minutes to hours to run, whereas program synthesizers which are custom-built for special problem domains can run much faster or even interactively [17].

In the near future, we believe it will be possible to produce professional-looking figures just by drawing them and then letting an artificially-intelligent agent write the code. 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 Wetzel. 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] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.
- [11] Arnaud Doucet, Nando De Freitas, and Neil Gordon, editors. *Sequential Monte Carlo Methods in Practice*. Springer, 2001.
- [12] Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. *ACM SIGPLAN Notices*, 50(10):107–126, 2015.
- [13] Daniel D Lee and H Sebastian Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, 401(6755):788–791, 1999.
- [14] Michael AA Cox and Trevor F Cox. Multidimensional scaling. *Handbook of data visualization*, pages 315–347, 2008.
- [15] Christopher M. Bishop. Mixture Density Networks. Technical report, 1994.
- [16] Volodymyr Mnih, Nicolas Heess, Alex Graves, et al. Recurrent models of visual attention. In *Advances in neural information processing systems*, pages 2204–2212, 2014.
- [17] Vu Le and Sumit Gulwani. Flashextract: a framework for data extraction by examples. In *ACM SIGPLAN Notices*, volume 49, pages 542–553. ACM, 2014.