

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 \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).

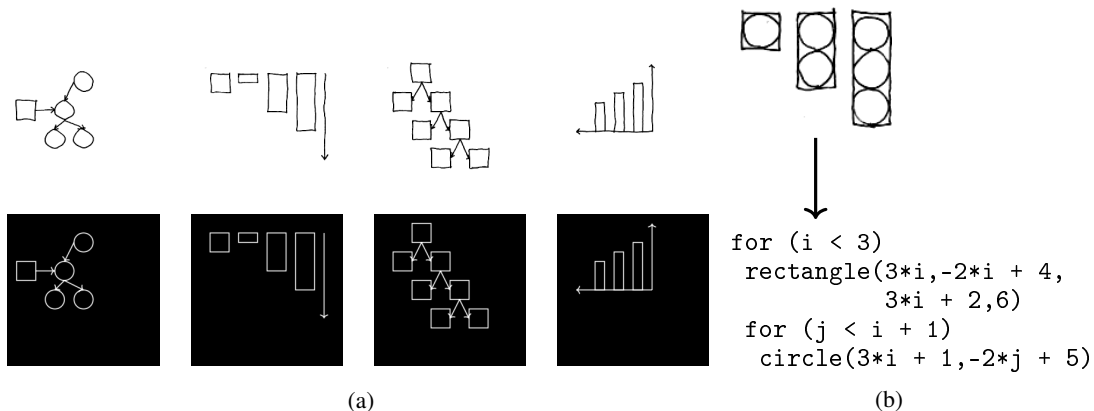


Figure 1: (a): Model learns to convert hand drawings (top) into \LaTeX (bottom). (b) Synthesizes high-level *graphics program* from hand drawing.

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 correcting errors made by the deep network, measuring similarity between hand drawings, and extrapolating figures.

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(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, 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.

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 256×256 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. See supplement for the full details of the architecture, which we implemented in Tensorflow.

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 and maximizing (2) with respect to θ by gradient ascent. Training does not require back-propagation 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.

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

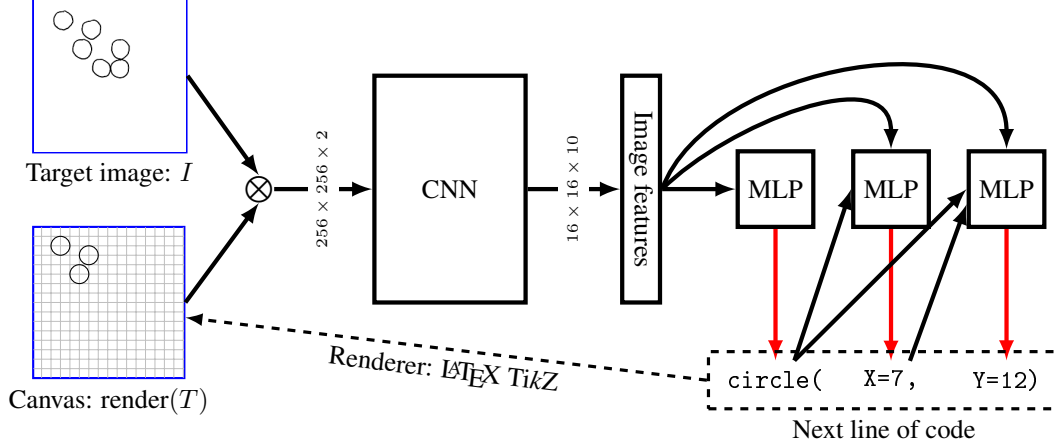


Figure 2: Our neural architecture for inferring the execution trace of a graphics program from its output. **Blue**: network inputs. **Black**: network operations. **Red**: samples from a multinomial. Typewriter font: network outputs. Renders snapped to a 16×16 grid, illustrated in gray.

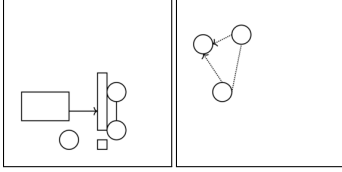


Figure 3: Network is trained to infer execution traces for randomly generated figures like the two shown above. See supplement for details of the training data generation.

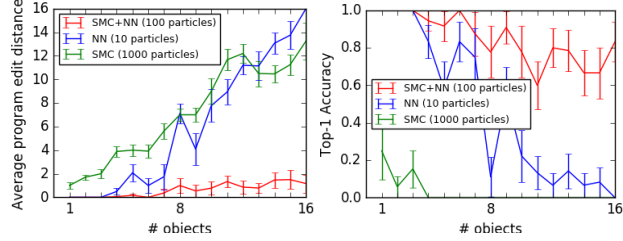


Figure 4: Using the model to parse latex output. The model is trained on diagrams with up to 8 objects. As shown above it generalizes to scenes with many more objects. Neither the stochastic search nor the neural network are sufficient on their own. # particles varies by model: we compare the models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

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 Monte Carlo (SMC) sampling scheme [10]. 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 on their renderings. We train our likelihood surrogate to approximate the symmetric

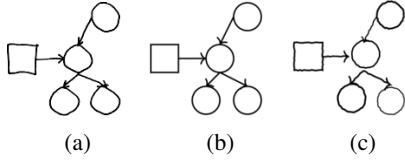


Figure 5: (a): a hand drawing. (b): Rendering of the trace our model infers for (a). We can generalize to hand drawings like these because we train the model on images corrupted by a noise process designed to resemble the kind of noise introduced by hand drawings - see (c) for a noisy rendering of (b).

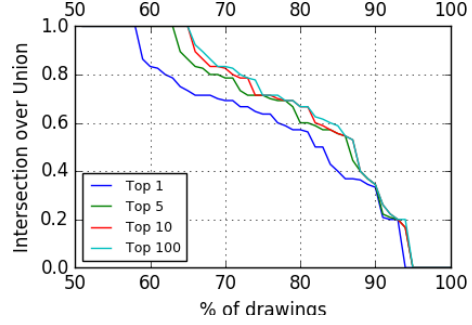


Figure 6: How close are the model’s outputs to the ground truth on hand drawings, as we consider larger sets of samples (1, 5, 10, 100 samples)? Distance to the ground truth trace is measured by the intersection over union of predicted vs. ground truth traces (sets of drawing commands).

129 difference, which is the number of drawing commands by which two traces differ:

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

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

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

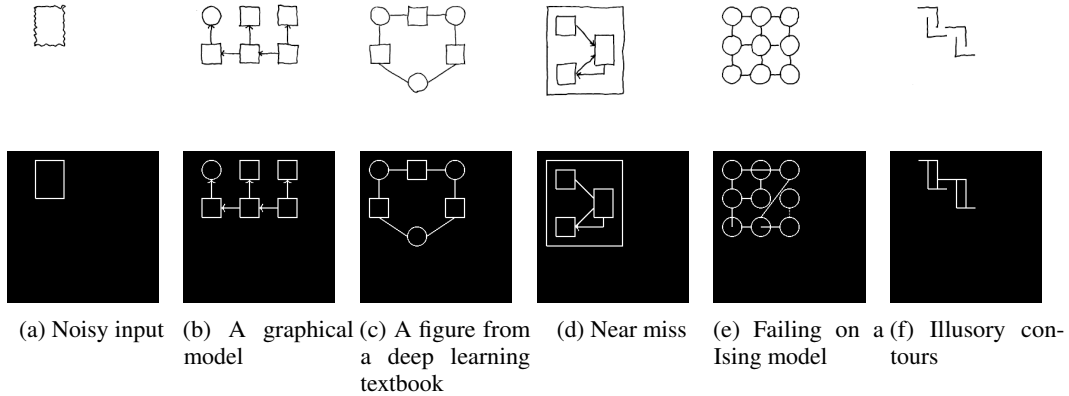


Figure 7: Example drawings above model outputs. See also Fig. 1. Stochastic search (SMC) can help correct for these errors, as can the program synthesizer (Section 4.1)

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 58% 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) [11]. Our DSL (Table 2) encodes prior knowledge of
 154 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 ($\mathcal{Z} * \text{Var} + \mathcal{Z}$).

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 many different numerical constants; see supplement.

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

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

172 4.1 Correcting errors made by the neural network

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

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

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

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

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

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

190 where the expectation is taken both over the model predictions and the (I, T) pairs in the training
 191 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
 192 extractor for programs. We extract a few basic features of a program, such as its size and how many
 193 loops it has, and use these features to help predict whether a trace is the correct explanation for an
 194 image.

195 We synthesized programs for the top five traces output by the deep network. Learning this prior over
 196 programs can help correct mistakes made by the neural network, and also occasionally introduces
 197 mistakes of its own; see Fig. 8 for a representative example of the kinds of corrections that it makes.
 198 On the whole modestly improves our Top-1 accuracy from 58% to 60%. Recall that from Fig. 6 that
 199 the best improvement in accuracy we could possibly get is 63% by looking at the top 5 traces.

200 4.2 Modeling similarity between drawings

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

207 We measure the similarity between two drawings by extract-
 208 ing features of the best programs that describe them. Here
 209 the features we use are just counts of the number of times
 210 that different components in the DSL were used (Table 2).
 211 We project these features down to a 2-dimensional sub-
 212 space using nonnegative matrix factorization (NMF: [12]);
 213 see Fig. 9. One could use many alternative similarity metrics between drawings which would capture
 214 pixel-level or object-level similarities while missing high-level geometric similarities. For example,
 215 we can use our learned distance metric between execution traces, $L_{\text{learned}}(\cdot | \cdot)$. Projecting these dis-
 216 tances to a 2-dimensional subspace using multidimensional scaling (MDS: [13]) reveals similarities
 217 between the objects in the drawings, while missing similarities at the level of the program.

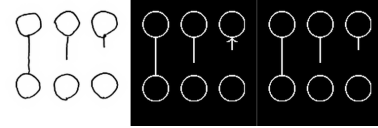


Figure 8: Left: hand drawing. Center: interpretation favored by the deep network. Right: interpretation favored after learning a prior over programs.

218 4.3 Extrapolating figures

219 Having access to the source code of a graphics program facilitates coherent, high-level edits to the
 220 figure generated by that program. For example, we can change all of the circles to squares or make all
 221 of the lines be dashed. We can also **extrapolate** figures by increasing the number of times that loops
 222 are executed. Extrapolating repetitive visual patterns comes naturally to humans, and building this
 223 ability into an application is practical: imagine hand drawing a repetitive graphical model structure
 224 and having our system automatically induce and extend the pattern. Fig. 11 shows extrapolations of
 225 programs synthesized from ground truth traces; see supplement for our full set of extrapolations.



Figure 9: 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.

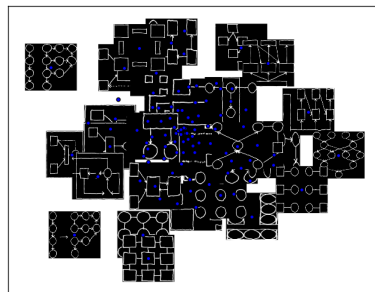


Figure 10: MDS on drawings using the learned distance metric, $L_{\text{learned}}(\cdot|\cdot)$. Drawings with similar looking parts in similar locations are clustered together.

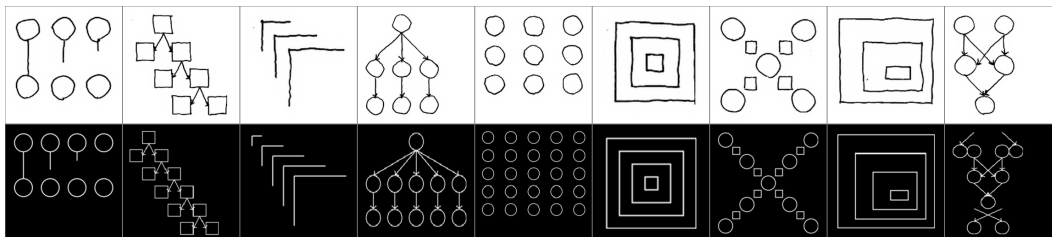


Figure 11: Top: hand drawings. Bottom: extrapolations produced by running loops for extra iterations. Rightmost pair is an illustrative failure case.

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 [14]. 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, could also 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 [15].

In the not-too-distant future, we believe it should be possible to produce professional-looking figures just by drawing them and then letting an artificially-intelligent agent write the corresponding 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] 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.