

# LEARNING TO INFER GRAPHICS PROGRAMS FROM HAND-DRAWN IMAGES

Anonymous authors

Paper under double-blind review

## 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 such as 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 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 1a).

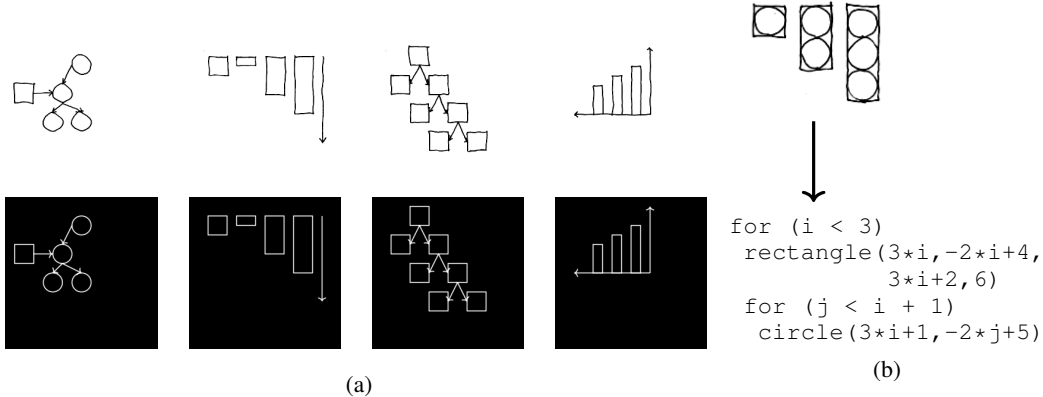


Figure 1: (a): Model learns to convert hand drawings (top) into  $\text{\LaTeX}$  (bottom). (b) Synthesizes high-level *graphics program* from hand drawing.

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

We present a hybrid architecture for inferring graphics programs that is structured around these two steps. For the first step, our approach uses a deep network to infer a set of primitive shape-drawing

commands. We refer to this set as an *execution trace*, since it corresponds to the commands that a program would have issued but lacks the high-level structure that determines how the program decided to issue them. For added robustness, we train the network to produce a proposal distribution of the most likely traces and use stochastic search to find the trace that best matches the input. The network is trained from an automatically-generated corpus of synthetic image-generating programs.

The second step involves generating a high-level program capable of producing the program trace identified by the first phase. This paper argues that this stage is best achieved by *constraint-based program synthesis* (1). The program synthesizer can search the space of possible programs for one capable of producing the desired trace – inducing structures like symmetries, repetition, or conditional branches. Although these program synthesizers do not need any training data, we show how to learn a *search policy* in order to synthesize programs an order of magnitude faster than constraint-based synthesis techniques alone.

Images are high-dimensional and unstructured while programs are high-level and symbolic. We resolve this mismatch by advancing a new model of collaboration between these two representations, which we call The Trace Hypothesis:

**The Trace Hypothesis.** The set of commands issued by a program is the correct liaison between high-dimensional unstructured perceptual input and high-level structured symbolic representations.

The roadmap of our paper is structured around the trace hypothesis, as outlined in Figure 2.

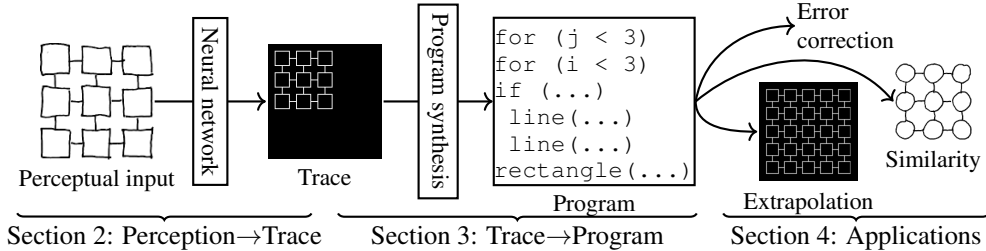


Figure 2: Paper roadmap. Both the paper and the system pipeline are structured around the *trace hypothesis*

Prior work addresses converting hand drawings to vectors (3; 4) and converting synthetic images to symbolic parses (5; 6). Our new contributions are:

- A model that converts sketches to high-level programs
- An algorithm for learning a policy for efficiently searching for programs
- The trace hypothesis: a framework for going from perception to programs

## 2 NEURAL ARCHITECTURE FOR INFERRING DRAWING DRAWING COMMANDS

We developed a deep network architecture for efficiently inferring an 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 3 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; see Table 1 for the drawing commands we currently model. We predict the drawing command token-by-token,<sup>1</sup> conditioning each token both on the image features and on the

<sup>1</sup>A token is an atomic symbol – so the model is *not* predicting lines of code character-by-character, but instead knows about the syntactic structure of primitive commands

previously generated tokens. We also use a differentiable attention mechanism (Spatial Transformer Networks: (7)) to let the model attend to different regions of the image while predicting each token. 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. For our main experiments we constrain coordinates to lie on a discrete  $16 \times 16$  grid, and later explain how to remove this restriction. See supplement for the full details of the architecture, which we implemented in Tensorflow (8).

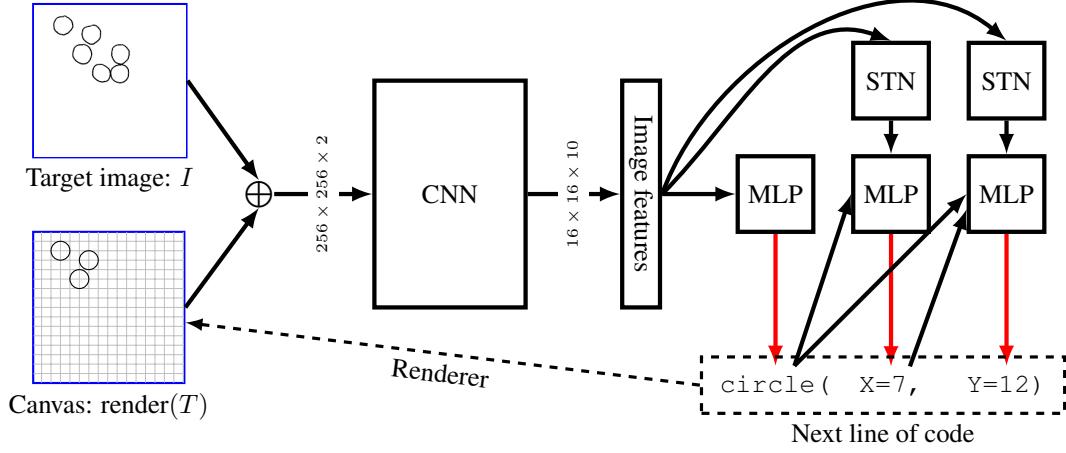


Figure 3: Our neural architecture for inferring the execution trace of a graphics program from its output. **Blue**: network inputs. **Black**: network operations. **Red**: samples from a multinomial. Typewriter font: network outputs. Renders snapped to a  $16 \times 16$  grid, illustrated in gray. STN (spatial transformer network) is a differentiable attention mechanism (7).

<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>     <math>\text{arrow} \in \{0, 1\}, \text{dashed} \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: Primitive drawing commands currently supported by our model.

For the model in Fig. 3, the distribution over the next drawing command factorizes as:

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

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,  $f_\theta(\cdot, \cdot)$  is the image feature extractor (convolutional network), and  $a_\theta(\cdot | \cdot)$  is an attention mechanism. 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  $T$ , the subscripts on  $T$  index drawing commands within the trace (so  $T_n$  is a sequence of tokens:  $t_1 t_2 \cdots t_K$ ), and the `STOP` token is emitted by the network to signal that the trace explains the image.

We train the network by sampling execution traces  $T$  and target images  $I$  for randomly generated scenes and maximizing Eq. 2 with respect to  $\theta$  by gradient ascent. We trained the network on  $10^5$  scenes, which takes a little less than a day on an Nvidia TitanX GPU.

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 (9), but critically, *without a learned recurrent memory state*, e.g. an RNN. We can do without an RNN because we have access to ground truth (image, trace) pairs. This allows handling scenes with a very large number of objects without having to worry about the conditioning of gradients as they propagate over long sequences, and also makes training more straightforward (it is just maximum likelihood estimation of the model parameters).

Our network suffices to “derender” synthetic images like those shown in Figure 4. We can perform a beam search decoding to recover what the network thinks is the most likely execution trace for images like these, recovering traces maximizing  $\mathbb{P}_\theta[T|I]$ . But, if the network makes a mistake (predicts an incorrect line of code), it has no way of recovering from the error. In order to derender an image with  $n$  objects, it must correctly predict  $n$  drawing commands – so its probability of success will decrease exponentially in  $n$ , assuming it has any nonzero chance of making a mistake. For added robustness as  $n$  becomes large, we treat the neural network outputs as proposals for a Sequential 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  $L(\cdot|\cdot) : \text{image}^2 \rightarrow \mathbb{R}$  uses the distance between two images as a proxy for a likelihood. Unconventionally, the target distribution of the SMC sampler includes the likelihood under the proposal distribution. Intuitively, both the proposal distribution and the distance function offer complementary signals for whether a drawing command is correct. Empirically, accuracy is much better when we combine these signals.

Figure 5 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  $\leq 12$  objects, it successfully parses scenes with many more objects than the training data. We also compare with a natural baseline that models the problem as a kind of image captioning (“LSTM” in Figure 5). Given the target image, this baseline produces the program trace in one shot by using a CNN to extract features of the input which are passed to an LSTM which finally predicts the trace token-by-token. This architecture is used in several successful neural models of image captioning (e.g., (11)), but, for this domain, does not suffice to parse cluttered scenes with many objects.

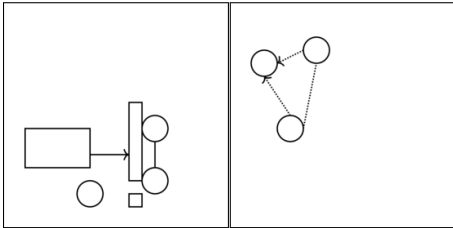


Figure 4: Network is trained to infer execution traces for randomly generated scenes like the two shown above. See supplement for details of the training data generation.

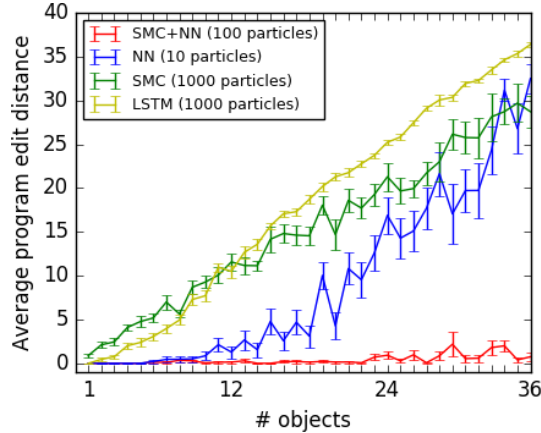


Figure 5: Parsing  $\text{\LaTeX}$  output after training on diagrams with  $\leq 12$  objects. It generalizes to scenes with many more objects. Neither the stochastic search nor the neural network are sufficient on their own. # particles varies by model: we compare the models *with equal runtime* ( $\approx 1$  sec/object)

## 2.1 GENERALIZING TO HAND DRAWINGS

A practical application of our neural network is the automatic conversion of hand drawings into a subset of  $\text{\LaTeX}$ . 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 6; 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 2). 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:

$$-\log L_{\text{learned}}(\text{render}(T_1)|\text{render}(T_2)) \approx \frac{|T_1 - T_2| + |T_2 - T_1|}{2} \quad (3)$$

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. Because our model assumes that objects are snapped to a  $16 \times 16$  grid, we made the drawings on graph paper.

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 63% 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. 8). Because our current model sometimes makes mistakes on hand drawings, 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.

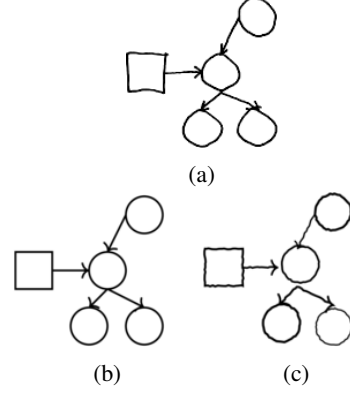
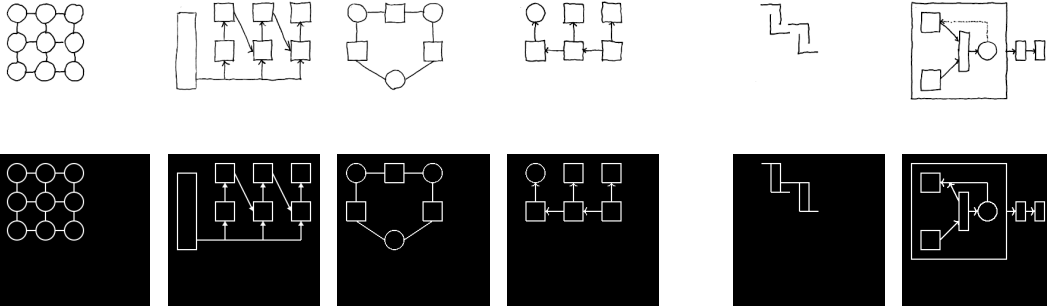


Figure 6: (a): a hand drawing. (b): Rendering of the trace our model infers for (a). We can generalize to hand drawings like these because we train the model on images corrupted by a noise process designed to resemble the kind of noise introduced by hand drawings - see (c) for a noisy rendering of (b).



(a) Left to right: Ising model, recurrent network architecture, a figure from a deep learning textbook, graphical model

(b) Near misses. Rightmost: illusory contours (note: no SMC)

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)

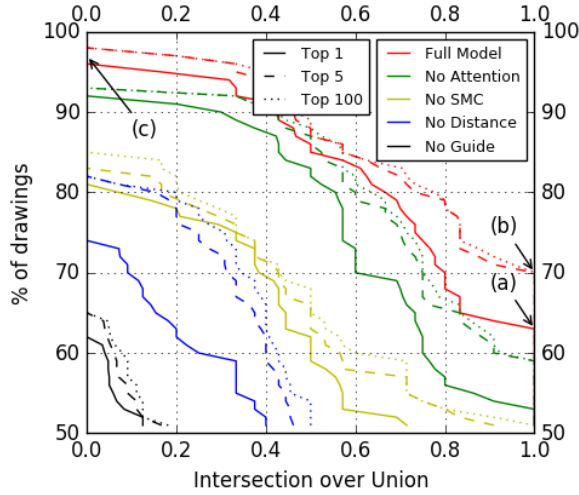


Figure 8: How close are the model’s outputs to the ground truth on hand drawings, as we consider larger sets of samples (1,10,100)? Distance to ground truth trace measured by the intersection over union of predicted vs. ground truth traces. (a) for 63% of drawings the model’s top prediction is exactly correct; (b) for 70% of drawings the ground truth is in the top 10 model predictions; (c) for 4% of drawings all of the model outputs have no overlap with the ground truth trace.

### 3 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 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) (12). Our DSL (Table 2) encodes prior knowledge of what graphics programs tend to look like.

---

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

---

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

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  $T$ :

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

We define the cost of a program to be the number of Statement’s it contains (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 such program synthesis problems. We use the state-of-the-art Sketch tool (1). Describing Sketch’s program synthesis algorithm is beyond the scope of this paper; see (1). 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 having no guarantee on the time required to find a minimum cost solution, it comes with the guarantee that it will always find a globally optimal program. In practice, synthesis times vary from minutes to hours, with 16% of the diagrams timing out the synthesizer after 3 hours. See Figure 10 for examples of the kinds of programs our system recovers. A main impediment to our use of these general techniques is the prohibitively high cost of searching for programs. We next describe how to learn to synthesize programs much faster (Section 3.1), timing out on 2% of the drawings and solving most problems in under a minute.

### 3.1 LEARNING A SEARCH POLICY FOR SYNTHESIZING PROGRAMS

We want to leverage powerful, domain-general techniques from the program synthesis community, but make them much faster by learning a domain-specific *search policy*. A search policy proposes search problems like those in Equation 4, but can also offer additional constraints on the structure of the program (Table 3). For example, a policy might decide to first try searching over small programs before searching over large programs, or decide to prioritize searching over programs that have loops.

A search policy  $\pi_\theta(\sigma|T)$  takes as input a trace  $T$  and predicts a distribution over synthesis problems, each of which is written  $\sigma$  and corresponds to a set of possible programs to search over (so  $\sigma \subset \text{DSL}$ ). Good policies will prefer tractable program spaces, so that the search procedure will terminate early. Good policies will also prefer program spaces that are likely to contain programs that can explain the data. In general these two desiderata are in tension: tractable synthesis problems involve searching over smaller spaces, but smaller spaces are less likely to contain a minimum cost program for explaining the trace. Our goal now is to find the parameters of the policy, written  $\theta$ , which best navigates this trade-off.

Parameter	Description	Range
Loops?	Is the program allowed to loop?	{True, False}
Reflects?	Is the program allowed to have reflections?	{True, False}
Incremental?	Solve the problem piece-by-piece or all at once?	{True, False}
Maximum depth	Bound on the depth of the program syntax tree	{1, 2, 3}

Table 3: Parameterization of different ways of posing the program synthesis problem. The policy learns to choose parameters likely to quickly yield a minimal cost program. In our notation, we write  $\sigma$  to mean an assignment to each of these parameters, so  $\sigma$  can assume one of 24 different values.

Given a search policy, what is the best way of using it to quickly find minimum cost programs? We use a *bias-optimal search algorithm* (see Schmidhuber 2004: (2)):

**Definition: Bias optimality.** A search algorithm is *n-bias optimal* with respect to a distribution  $\mathbb{P}_{\text{bias}}[\cdot]$  if it is guaranteed to find a solution  $\sigma$  after searching for at least time  $n \times \frac{t(\sigma)}{\mathbb{P}_{\text{bias}}[\sigma]}$ , where  $t(\sigma)$  is the time it takes to verify that  $\sigma$  is a solution to the search problem.

An example of a 1-bias optimal search algorithm is a time-sharing system that allocates  $\mathbb{P}_{\text{bias}}[\sigma]$  of its time to trying  $\sigma$ . We construct a 1-bias optimal search algorithm for our domain by identifying  $\mathbb{P}_{\text{bias}}[\sigma] = \pi_\theta(\sigma|T)$  and  $t(\sigma)$  with how long the program synthesizer takes to search through the programs in  $\sigma$ .

In theory any  $\pi_\theta(\cdot|\cdot)$  is a bias-optimal searcher. But the real world runtime of the algorithm depends strongly upon the bias  $\mathbb{P}_{\text{bias}}[\cdot]$ . Our approach is to learn  $\mathbb{P}_{\text{bias}}[\cdot]$  by picking the policy that minimizes the expected bias-optimal time to solve a training corpus of graphics program synthesis problems,

which is:

$$\text{Loss}(\theta; \text{training corpus}) = \mathbb{E}_{T \in \text{training corpus}} \left[ \min_{\sigma \in \text{BEST}(T)} \frac{t(\sigma|T)}{\pi_{\theta}(\sigma|T)} \right] \quad (5)$$

where  $\sigma \in \text{BEST}(T)$  if a minimum cost program for  $T$  is in  $\sigma$ .

To generate a training corpus for learning a policy which minimizes this loss, we synthesized minimum cost programs for each trace of our hand drawings and for each  $\sigma$ . Solving these 2400<sup>2</sup> synthesis problems takes a little over a day on a 20-CPU machine. With the training set in hand we locally minimize this loss using gradient descent. See supplement for details.

Because we want to learn a policy from only 100 hand-drawn sketches, we chose a simple low-capacity, bilinear model for a policy:

$$\pi_{\theta}(\sigma|T) \propto \exp(\phi_{\text{params}}(\sigma)^{\top} \theta \phi_{\text{trace}}(T)) \quad (6)$$

where  $\phi_{\text{params}}(\sigma)$  is a one-hot encoding of the parameter settings of  $\sigma$  (see Table 3) and  $\phi_{\text{trace}}(T)$  extracts a few simple features of the trace  $T$ ; see supplement for details.

We contrast our learned search policy with two alternatives (see Figure 9): (1) *Sketch*, which poses the entire problem wholesale to the Sketch program synthesizer; and (2) an *Oracle*, a policy which always picks the quickest to search  $\sigma$  also containing a minimum cost program. Our approach improves upon Sketch by itself, and comes close to the Oracle’s performance. One could never construct this Oracle, because the agent does not know ahead of time which  $\sigma$ ’s contain minimum cost programs nor does it know how long each  $\sigma$  will take to search. The Oracle is an upper bound on the performance of any search policy. With this learned policy in hand we can now synthesize most programs in under a minute.

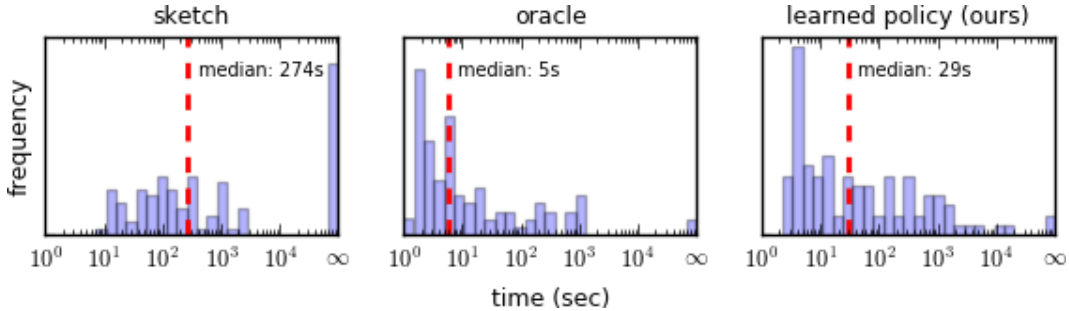


Figure 9: How long does it typically take to synthesize a minimum cost program? Sketch: out-of-the-box performance of the Sketch (1) program synthesizer. Oracle: baseline that always picks the easiest synthesis problem containing a minimum cost program. Learned policy: a bias-optimal learned search policy running on an ideal timesharing machine. Oracle is impossible to implement and is an upper bound on the performance of any search policy.  $\infty$  = timeout. Red dashed line is median time.

<sup>2</sup>There are 100 the drawings and  $2 \times 2 \times 2 \times 3 = 24$  different values of  $\sigma$




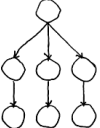
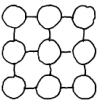
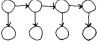

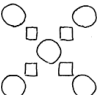
Drawing	Trace	Program	Compression factor
	<pre> Line(2,15, 4,15) Line(4,9, 4,13) Line(3,11, 3,14) Line(2,13, 2,15) Line(3,14, 6,14) Line(4,13, 8,13) </pre>	<pre> for(i&lt;3)   line(i,-1*i+6,         2*i+2,-1*i+6)   line(i,-2*i+4,i,-1*i+6) </pre>	$\frac{6}{3} = 2x$
	<pre> Line(5,13, 2,10),   arrow = True) Circle(5,9) Circle(8,5) Line(2,8, 2,6),   arrow = True) Circle(2,5) Circle(5,14) Circle(2,9) </pre> <p>... etc. ...; 13 lines</p>	<pre> circle(4,10) for(i&lt;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) </pre>	$\frac{13}{6} = 2.2x$
	<pre> Circle(5,8) Circle(2,8) Circle(8,11) Line(2,9, 2,10) Circle(8,8) Line(3,8, 4,8) Line(3,11, 4,11) Line(8,9, 8,10) Circle(5,14) </pre> <p>... etc. ...; 21 lines</p>	<pre> for(i&lt;3)   for(j&lt;3)     if(j&gt;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) </pre>	$\frac{21}{6} = 3.5x$
	<pre> Line(11,14, 13,14),   arrow = True) Circle(10,10) Line(10,13, 10,11),   arrow = True) Line(7,14, 9,14),   arrow = True) Circle(6,10) Circle(2,10) </pre> <p>... etc. ...; 15 lines</p>	<pre> for(i&lt;4)   line(-4*i+13,4,-4*i+13,2,         arrow=True) for(j&lt;3)   if(j&gt;0)     circle(-4*i+13,4*j+-3)     line(-4*j+10,5,-4*j+12,5,           arrow=True) </pre>	$\frac{15}{6} = 2.5x$
	<pre> Line(5,11,5,13) Circle(5,9) Circle(5,14) Circle(2,9) Circle(2,14) Circle(8,14) Circle(8,9) Line(2,10,2,13) Line(8,12,8,13) </pre>	<pre> for (i&lt;3)   circle(-3*i+7,1)   circle(-3*i+7,6)   line(-3*i+7,-1*i+4,-3*i+7,5) </pre>	$\frac{9}{4} = 2.25x$
	<pre> Circle(2,8) Rectangle(6,9, 7,10) Circle(8,8) Rectangle(6,12, 7,13) Rectangle(3,9, 4,10) Circle(2,14) Circle(8,14) Circle(5,11) Rectangle(3,12, 4,13) </pre>	<pre> reflect(y=8) for(i&lt;3)   if(i&gt;0)     rectangle(3*i-1,2,3*i,3)     circle(3*i+1,3*i+1) </pre>	$\frac{9}{5} = 1.8x$

Figure 10: Example drawings (left), their ground truth traces (middle left), and programs synthesized from these traces (middle right). Compared to the trace the programs are more compressive (right: programs have fewer lines than traces) and automatically group together related drawing commands. Note the nested loops in the Ising model, special case conditionals for the HMM, combination of symmetry and iteration in the bottom figure, and affine transformations in the top figure and second figure to bottom.

## 4 APPLICATIONS OF GRAPHICS PROGRAM SYNTHESIS

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 correct errors from the execution trace proposal network by favoring traces which lead to more concise or general programs. For example, figures with perfectly aligned objects are preferable to figures whose parts are slightly misaligned, and precise alignment lends itself to short programs. Concretely, we estimated a prior over programs. Then, given the few most likely traces output by the neurally guided sampler, we reranked them according to the prior probability of their programs. Our sampler could only do better on 7/100 drawings by looking at the Top-100 sampled traces (see Fig. 8), precluding a statistically significant analysis of how much learning a prior over programs could help correct errors. But, learning this prior does sometimes help correct mistakes made by the neural network, and also occasionally introduces mistakes of its own; see Fig. 11 for a representative example of the kinds of corrections that it makes. See supplement for details.

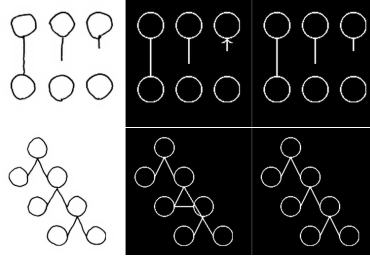


Figure 11: Left: hand drawings. Center: interpretations favored by the deep network. Right: interpretations favored after learning a prior over programs. Our learned prior favors shorter, simpler programs, thus (top example) continuing the pattern of not having an arrow is preferred, or (bottom example) continuing the “binary search tree” is preferred.

### 4.2 MODELING SIMILARITY BETWEEN DRAWINGS

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

We measure the similarity between two drawings by extracting features of the best programs that describe them. Our features are counts of the number of times that different components in the DSL were used (Table 2). We project these features down to a 2-dimensional subspace using primary component analysis (PCA); see Fig. 12. One could use many alternative similarity metrics between drawings which would capture pixel-level similarities while missing high-level geometric similarities. We used our learned distance metric between execution traces,  $L_{\text{learned}}(\cdot|\cdot)$ , and projected to a 2-dimensional subspace using multidimensional scaling (MDS: (13)). This reveals similarities between the objects in the drawings, while missing similarities at the level of the program.

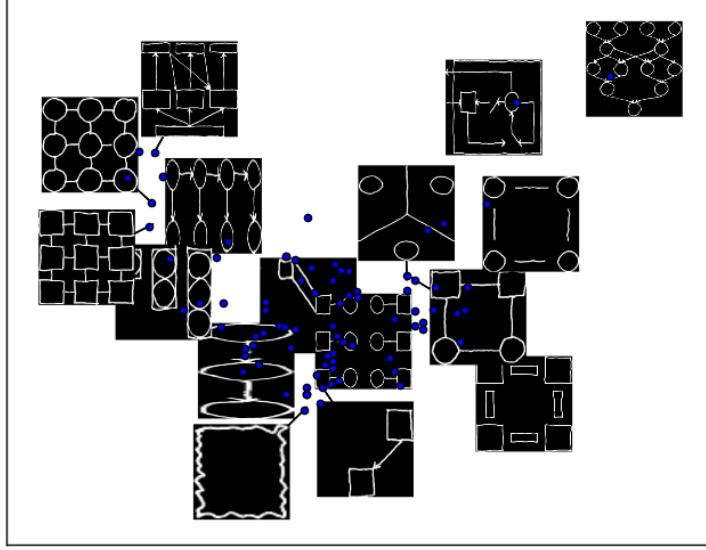


Figure 12: PCA on features of the programs that were synthesized for each drawing. Symmetric figures cluster to the right; “loopy” figures cluster to the left; complicated programs are at the top and simple programs are at the bottom.

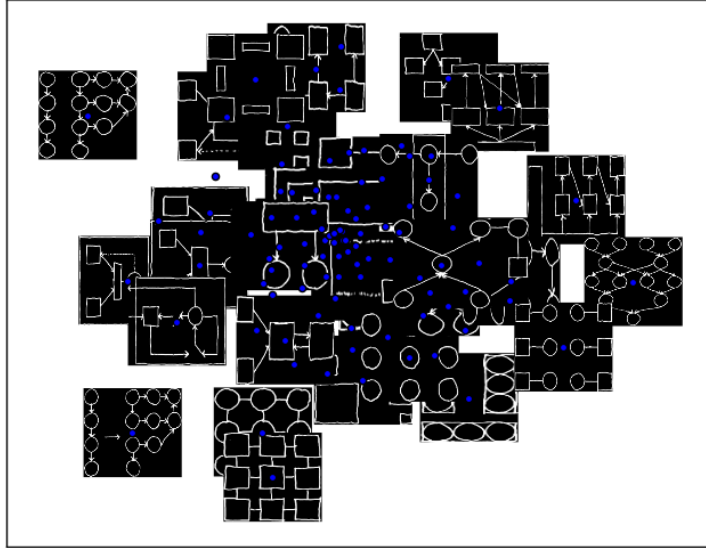


Figure 13: MDS on drawings using the learned distance metric,  $L_{\text{learned}}(\cdot|\cdot)$ . Drawings with similar looking parts in similar locations are clustered together.

#### 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. 14 shows extrapolations of programs synthesized from ground truth traces; see supplement for our full set of extrapolations.

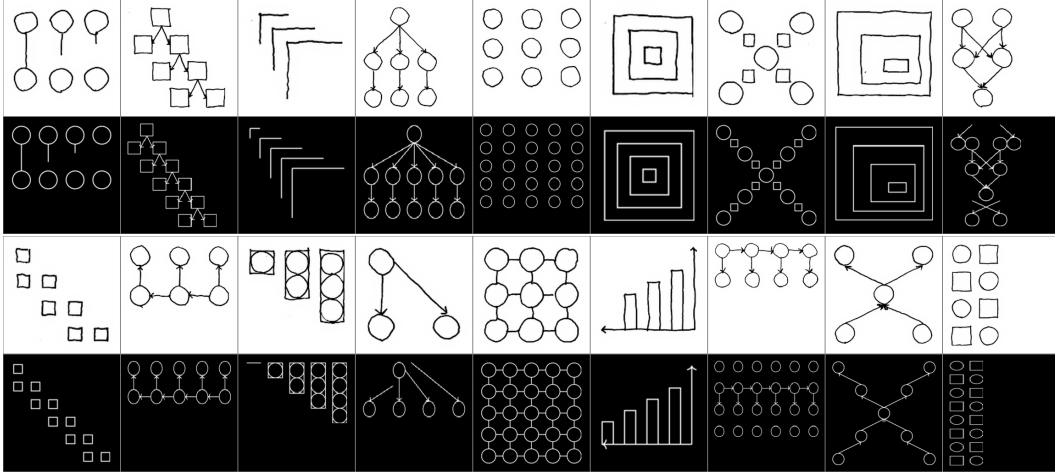


Figure 14: Top, white: hand drawings. Bottom, black: extrapolations produced by running loops for extra iterations.

## 5 DISCUSSION

## 6 RELATED WORK

Our approach to learning to search for programs draws theoretical underpinnings from Levin search (14; 15) and Schmidhuber’s OOPS model (2). DeepCoder (16) and (17) are recent models which, like ours, learn to predict which program components are likely to be useful given a program synthesis problem. Our work is different from their approaches because we treat the problem as a variety of *metareasoning*, identifying and modeling the trade-off between tractable search and probability of successful search. TerpreT (18) was the first work to systematically compare constraint-based program synthesis techniques against gradient-based search techniques (e.g., a DNC (19)). The TerpreT experiments help motivate our use of constraint-based techniques.

Our work bears resemblance to the Attend-Infer-Repeat (AIR) system, which learns to decompose an image into its constituent objects (9). 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 hand-drawn sketches.

Our image-to-execution-trace parsing architecture builds on prior work on controlling procedural graphics programs (20). 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 (3). Another uses convolutional networks to support sketch-based instantiation of procedural primitives within an interactive architectural modeling system (4). 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. Sketch-n-Sketch is a bi-directional editing system in which direct manipulations to a program’s output automatically propagate to the program source code (21). 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 (22) system also aims to have a high-level understanding of hand-drawn figures. Their primary goal is cognitive modeling (they apply their system to solving IQ-test style visual reasoning problems), whereas we are interested in building an automated AI application (e.g. in our system the user need not annotate which strokes correspond to which shapes; our neural network produces something equivalent to the annotations). Unsupervised Program Synthesis (23) is a related framework which was also applied to geometric reasoning problems. The goals of (23) were cognitive modeling, and they applied their technique to synthetic scenes used in human behavioral studies.

The idea that an execution trace could assist in program learning goes back to the 1970’s (24) and has also been applied in neural models of program induction, like Neural Program Interpreters (25). Our contribution to this idea is the trace hypothesis: that execution traces can be inferred from perceptual data, and that the trace is the correct bridge between perception and symbolic representation. Our work is the first to articulate and explore this hypothesis by demonstrating how a trace could be inferred and how it can be used to synthesize a high-level program.

## 6.1 FUTURE WORK

There are many directions for future work. The proposal network currently handles only a very small subset of  $\text{\LaTeX}$  drawing commands, though there is no reason that it could not be extended to handle more with a higher-capacity network. 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 (26).

In the parsing phase, the proposal network currently samples positional variables on a discrete  $16 \times 16$  grid. More general types of drawings could be supported by instead sampling from continuous distributions, e.g. using Mixture Density Networks (MDN) (27). Implementing a variant of our model that predicts continuous coordinates using Mixture Density Networks is straightforward; we have prototyped such a system and show examples of it both working and failing in the supplement. But, for our collection of hand drawings, discrete positional variables suffice, and currently work much better in practice.

## 6.2 CONTRIBUTIONS

We have presented a system for inferring graphics programs which generate  $\text{\LaTeX}$ -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.

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.

## ACKNOWLEDGMENTS

We are grateful for advice from Will Grathwohl on the design of the neural architecture.

# SUPPLEMENTAL INFORMATION.

The supplement may be found at: <http://web.mit.edu/ellisk/www/graphicsProgramSupplement.pdf>

# REFERENCES

- [1] Armando Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.
- [2] Jürgen Schmidhuber. Optimal ordered problem solver. *Machine Learning*, 54(3):211–254, 2004.
- [3] 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.
- [4] 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.
- [5] Jiajun Wu, Joshua B Tenenbaum, and Pushmeet Kohli. Neural scene de-rendering. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.
- [6] Tony Beltramelli. pix2code: Generating code from a graphical user interface screenshot. *CoRR*, abs/1705.07962, 2017.
- [7] Max Jaderberg, Karen Simonyan, Andrew Zisserman, et al. Spatial transformer networks. In *Advances in Neural Information Processing Systems*, pages 2017–2025, 2015.
- [8] 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.
- [9] 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>.
- [10] Arnaud Doucet, Nando De Freitas, and Neil Gordon, editors. *Sequential Monte Carlo Methods in Practice*. Springer, 2001.
- [11] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3156–3164, 2015.
- [12] Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. *ACM SIGPLAN Notices*, 50(10):107–126, 2015.
- [13] Michael AA Cox and Trevor F Cox. Multidimensional scaling. *Handbook of data visualization*, pages 315–347, 2008.
- [14] Leonid Anatolevich Levin. Universal sequential search problems. *Problemy Peredachi Informatsii*, 9(3):115–116, 1973.
- [15] Raymond J Solomonoff. Optimum sequential search. 1984.
- [16] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. DeepCoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, November 2016.
- [17] Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. A machine learning framework for programming by example. In *ICML*, pages 187–195, 2013.
- [18] Alexander L Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. Terpret: A probabilistic programming language for program induction. *arXiv preprint arXiv:1608.04428*, 2016.

- [19] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.
- [20] 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.
- [21] 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.
- [22] 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.
- [23] Kevin Ellis, Armando Solar-Lezama, and Josh Tenenbaum. Unsupervised learning by program synthesis. In *Advances in Neural Information Processing Systems*, pages 973–981, 2015.
- [24] Phillip D. Summers. A methodology for lisp program construction from examples. *J. ACM*, 24(1):161–175, January 1977.
- [25] Scott Reed and Nando de Freitas. Neural programmer-interpreters. *CoRR*, abs/1511.06279, 2015.
- [26] Vu Le and Sumit Gulwani. Flashextract: a framework for data extraction by examples. In *ACM SIGPLAN Notices*, volume 49, pages 542–553. ACM, 2014.
- [27] Christopher M. Bishop. Mixture Density Networks. Technical report, 1994.