Learning to Infer Graphics Programs from Hand-Drawn Images

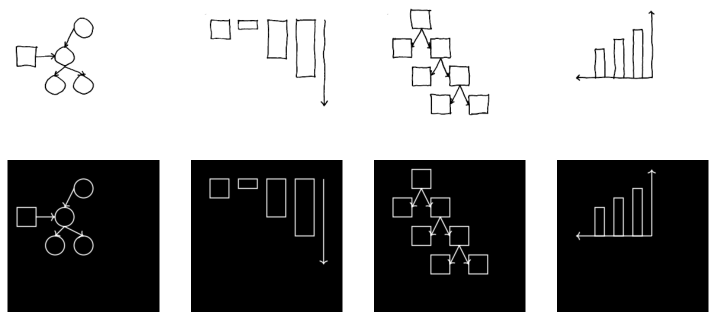
Anonymous authors
Paper under double-blind review

Abstract

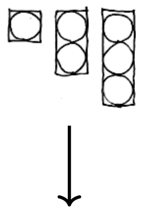
We introduce a model that learns to convert simple hand drawings into graphics programs written in a subset of LaTeX. The model combines techniques from deep learning and program synthesis. We learn a convolutional neural network that proposes plausible drawing primitives that explain an image. These drawing primitives are like a trace of the set of primitive commands issued by a graphics program. We learn a model that uses program synthesis techniques to recover a graphics program from that trace. These programs have 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, measure similarity between 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 Fig. 1a).



(a)



```
for (i < 3)
 rectangle(3*i,-2*i+4,
           3*i+2,6)
 for (j < i + 1)
  circle(3*i+1,-2*j+5)
```

(b)

Figure 1: (a): Model learns to convert hand drawings (top) into LaTeX (rendered below). (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 Fig. 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, a deep network to infers a set of primitive shape-drawing commands. We refer to this set as a *trace set* since it corresponds to the set of commands in a program's execution trace but lacks the high-level structure that determines how the program decided to issue them. The second step involves synthesizing a high-level program capable of producing the trace set identified by the first phase. We achieve this by *constraint-based program synthesis* (Solar Lezama, 2008). The program synthesizer searches the space of possible programs for one capable of producing the desired trace set – inducing structures like symmetries, loops, 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, which we call a *trace set*, 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 Fig. 2.

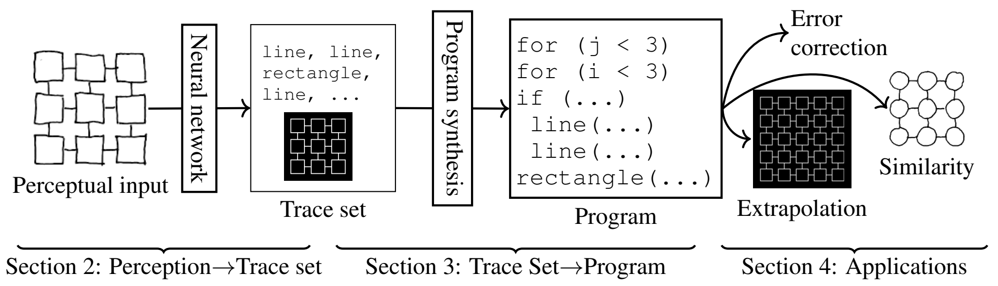


Figure 2: Both the paper and the system pipeline are structured around the *trace hypothesis*

The new contributions of this work are: (1) A model that converts sketches to high-level programs: in contrast to converting images to vectors or low-level parses (Huang et al., 2017; Nishida et al., 2016; Wu et al., 2017; Beltramelli, 2017; Deng et al., 2017). (2) An algorithm for learning a policy for efficiently searching for programs, building on Levin search (Levin, 1973) and generalizing recent work like DeepCoder (Balog et al., 2016). (3) The trace hypothesis: a framework for going from perception to programs, which connects this work to other models of program induction that use traces, like the Neural Program Interpreter (Reed & de Freitas, 2015).

2 Neural architecture for inferring trace sets

We developed a deep network architecture for efficiently inferring a trace set, T, from an image, I. Our model is a combination of ideas from Neurally-Guided Procedural Models (Ritchie et al., 2016) and Attend-Infer-Repeat (Eslami et al., 2016). The network constructs the trace set one drawing command at a time, conditioned on what it has drawn so far. Fig. 3 illustrates this architecture. We first pass a 256×256 target image and a rendering of the trace set 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 set; see Tbl. 1 for the drawing commands we currently model. We predict the drawing command token-by-token,conditioning each token both on the image features and on the previously generated tokens. We also use a differentiable attention mechanism (Spatial Transformer Networks: Jaderberg et al. (2015)) to let the model attend to different regions of the image while predicting each token. In Fig. 3 the network first decided to emit the `circle` token conditioned on the image features, then it emitted the x coordinate of the circle conditioned on the image features and the `circle` token, and finally it predicted the y coordinate of the circle conditioned on the image features, the `circle` token, and the x coordinate. We currently constrain coordinates to lie on a discrete 16×16 grid, but the grid could be made arbitrarily fine. See supplement for the full details of the architecture, which we implemented in Tensorflow (Abadi et al., 2015).

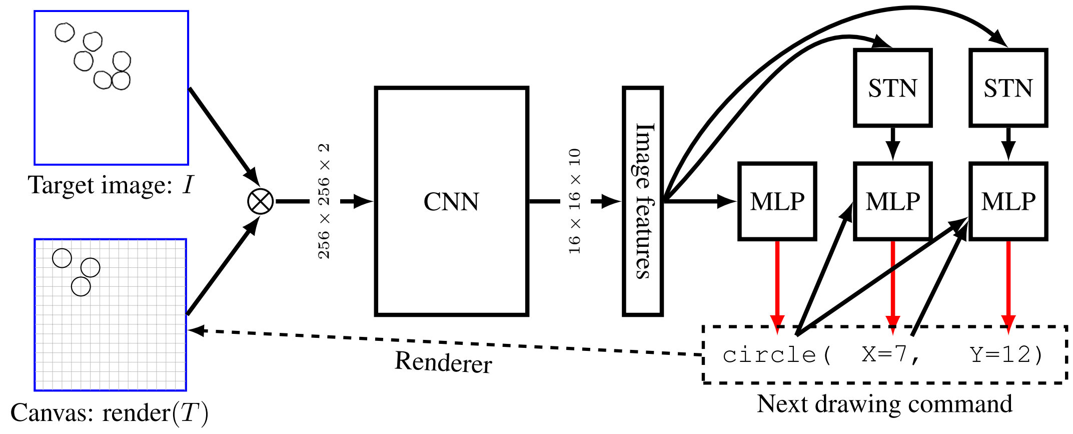


Figure 3: Our neural architecture for inferring the trace set of a graphics program from its output. Blue: network inputs. Black: network operations. Red: samples from a multinomial. `Typewriter font`: network outputs. Renders snapped to a 16×16 grid, illustrated in gray. STN (spatial transformer network) is a differentiable attention mechanism (Jaderberg et al., 2015).

Table 1: Primitive drawing commands currently supported by our model.

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

We train the network by sampling trace sets T and target images I for randomly generated scenes and maximizing the likelihood of T given I with respect to the model parameters, θ, by gradient ascent. We trained the network on 10^5 scenes, which takes a day on an Nvidia TitanX GPU.

Our network can "derender" random synthetic images by doing a beam search decoding to recover trace sets maximizing $\mathbb{P}_\theta[T|I]$. But, if the network makes a mistake (predicts an incorrect drawing command), it has no way of recovering from the error. To derender an image with n objects, it must correctly predict n commands – so its probability of success will decrease exponentially in n, assuming it has a 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 (Doucet et al., 2001). 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 $L(\cdot|\cdot) : \text{image}^2 \to \mathbb{R}$ 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.

Fig. 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 ≤ 12 objects, it successfully parses scenes with many more objects than the training data. We compare with a baseline that models the problem as image captioning (LSTM in Fig. 4). Given the target image, this baseline produces the trace set in one shot by using a CNN to extract features of the input which are passed to an LSTM which finally predicts the trace set token-by-token. This architecture is used in several successful neural models of image captioning (e.g., Vinyals et al. (2015)), but, for this domain, does not suffice to parse cluttered scenes with many objects.

2.1 Generalizing to hand drawings

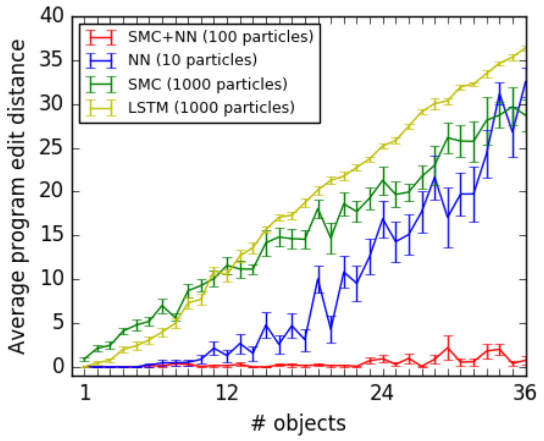


Figure 4: Parsing LaTeX output after training on diagrams with $\leq$ 12 objects. It generalizes to scenes with many more objects. Neither the stochastic search nor the neural network are sufficient on their own. # particles varies by model: we compare the models *with equal runtime* ($\approx$ 1 sec/object)

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 (Fig. 5; see supplement for details).

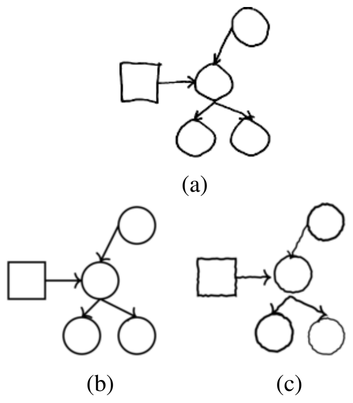


Figure 5: (a): hand drawing. (b): Rendering of the trace set our model infers for (a). (c): noisy rendering of (b).

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 trace sets conditioned upon their renderings. We train our likelihood surrogate to approximate the symmetric difference, which is the number of drawing commands by which two trace sets differ:

$$-\log L_{\text{learned}}(\text{render}(T_1)|\text{render}(T_2)) \approx |T_1 - T_2| + |T_2 - T_1| \tag{1}$$

We evaluate our system on 100 hand-drawn figures; see Fig. 6–7. These were drawn carefully but not perfectly. Because our model assumes that objects are snapped to a 16×16 grid, we made the drawings on graph paper. For each drawing we annotated a ground truth trace set and have the neurally guided SMC sampler produce many candidate interpretations 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 trace sets that are closer to the ground truth annotation (Fig. 8). We will show that the program synthesizer corrects some of these small errors (Sec. 4.1).

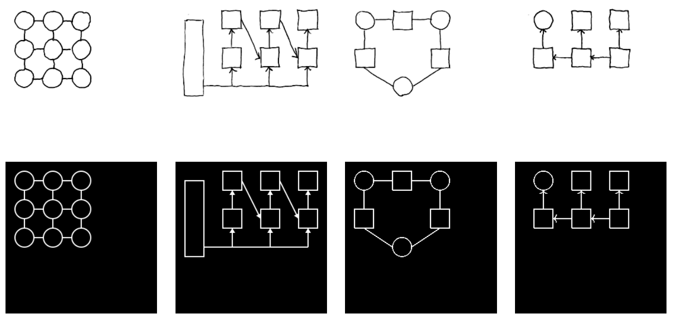


Figure 6: Left to right: Ising model, recurrent network architecture, figure from a deep learning textbook Goodfellow et al. (2016), graphical model

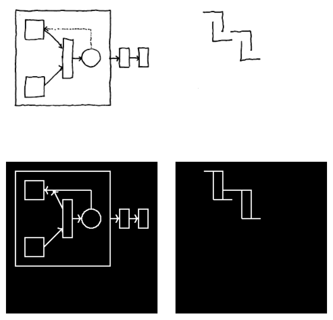


Figure 7: Near misses. Rightmost: illusory contours (note: no SMC)

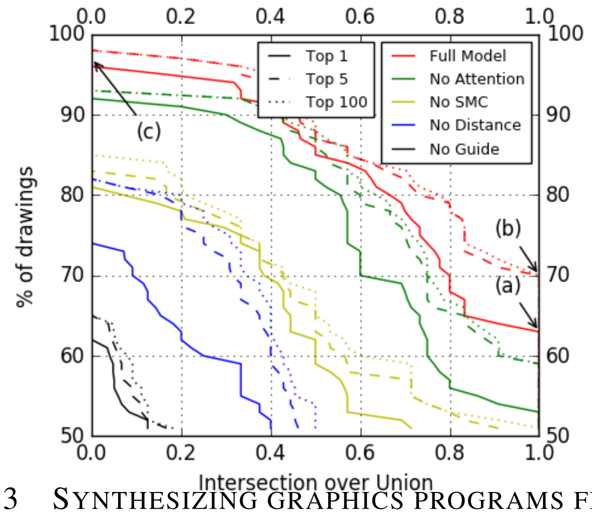


Figure 8: How close are the model's outputs to the ground truth on hand drawings, as we consider larger sets of samples (1, 5, 100)? Distance to ground truth trace set measured by the intersection over union of predicted vs. ground truth. (a) for 63% of drawings the model's top prediction is exactly correct; (b) for 70% of drawings the ground truth is in the top 5 model predictions; (c) for 4% of drawings all of the model outputs have no overlap with the ground truth. Red: the full model. Other colors: lesioned versions of our model.

3 Synthesizing Graphics Programs from Trace Sets

Although the trace set 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 describes such structures. We seek to synthesize graphics programs from their trace sets.

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) (Polozov & Gulwani, 2015). Our DSL (Tbl. 2) encodes prior knowledge of what graphics programs tend to look like.

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

Program$\to$	Statement; $\cdots$; Statement
Statement$\to$	`circle`(Expression,Expression)
Statement$\to$	`rectangle`(Expression,Expression,Expression,Expression)
Statement$\to$	`line`(Expression,Expression,Expression,Expression,Boolean,Boolean)
Statement$\to$	`for`($0 \leq$ Var $<$ Expression) `{ if` (Var > 0) `{` Program `};` Program `}`
Statement$\to$	`reflect`(Axis) `{` Program `}`
Expression$\to$	$\mathbb{Z}\times$Var+$\mathbb{Z}$
Var$\to$	A free (unused) variable
$\mathbb{Z}\to$	an integer
Axis$\to$	`X = `$\mathbb{Z}$ | `Y = `$\mathbb{Z}$

Given the DSL and a trace set 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:

$$\text{program}(T) = \underset{p\in\text{DSL, s.t. } p \text{ evaluates to } T}{\arg\min} \text{cost}(p) \tag{2}$$

We define the cost of a program to be the number of Statement's it contains (Tbl. 2). We also penalize using many different numerical constants; see supplement.

The constrained optimization problem in Eq. 2 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 (Solar Lezama, 2008). 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. Sketch requires a *finite program space*, which here means that the depth of the program syntax tree is bounded (we set the bound to 3), but has the guarantee that it always eventually finds the globally optimal solution. In exchange for this optimality guarantee it comes with no guarantees on runtime. For our domain synthesis times vary from minutes to hours, with 27% of the drawings timing out the synthesizer after 1 hour. Tbl. 3 shows programs recovered by our system. 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 (Sec. 3.1), timing out on 2% of the drawings and solving 58% of problems within 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 Eq. 2, but also offers additional constraints on the structure of the program (Tbl. 4). 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 set T and predicts a distribution over synthesis problems, each of which is written σ and corresponds to a set of possible programs to search over (so $\sigma \subset$ DSL). Good policies will prefer tractable program spaces, so that the search procedure will terminate early, but should also prefer program spaces likely to contain programs that concisely explain the data. These two desiderata are in tension: tractable synthesis problems involve searching over smaller spaces, but smaller spaces are less likely to contain good programs. Our goal now is to find the parameters of the policy, written θ, which best navigate this trade-off.

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* (Schmidhuber, 2004):

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 in σ 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 σ contains a solution to the search problem.

Intuitively, an n-bias optimal search algorithm spends at least $\mathbb{P}_{\text{bias}}[\sigma]/n$ of its time exploring solution space σ. 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 σ. 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 synthesizer takes to search the programs in σ. This means that the search algorithm explores the entire program space, but spends most of its time searching through the regions of the space that the policy judges to be most promising.

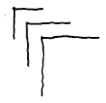
Now in theory any $\pi_\theta(\cdot|\cdot)$ is a bias-optimal searcher. But the actual runtime of the algorithm depends strongly upon the bias $\mathbb{P}_{\text{bias}}[\cdot]$. Our new approach is to learn $\mathbb{P}_{\text{bias}}[\cdot]$ by picking the policy minimizing the expected bias-optimal time to solve a training corpus, $\mathcal{D}$, of graphics program synthesis problems:

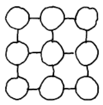
$$\textsc{Loss}(\theta; \mathcal{D}) = \mathbb{E}_{T\sim\mathcal{D}}\left[\min_{\sigma\in\textsc{Best}(T)} \frac{t(\sigma|T)}{\pi_\theta(\sigma|T)}\right] + \lambda\|\theta\|_2^2 \tag{3}$$

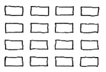
where $\sigma \in \textsc{Best}(T)$ if a minimum cost program for T is in σ.

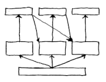
Table 3: Example drawings (left), their ground truth trace sets (middle left), and programs synthesized from these trace sets (middle right). Compared to the trace sets the programs are more compressive (right: programs have fewer lines than traces) and automatically group together related drawing commands. Note the nested loops and special case conditionals in the Ising model, combination of symmetry and iteration in the bottom figure, affine transformations in the top figure, and the complicated program in the second figure to bottom.

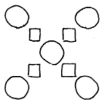
Drawing	Trace Set	Program	Compression factor

	`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)`
		`for(i<3)`
		` line(i,-1*i+6,`
		`      2*i+2,-1*i+6)`
		` line(i,-2*i+4,i,-1*i+6)`
			$\frac{6}{3} = 2\text{x}$

	`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)`
	... etc. ...; 21 lines
		`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)`
			$\frac{21}{6} = 3.5\text{x}$

	`Rectangle(1,10,3,11)`
	`Rectangle(1,12,3,13)`
	`Rectangle(4,8,6,9)`
	`Rectangle(4,10,6,11)`
	... etc. ...; 16 lines
		`for(i<4)`
		` for(j<4)`
		`  rectangle(-3*i+9,-2*j+6,`
		`            -3*i+11,-2*j+7)`
			$\frac{16}{3} = 5.3\text{x}$

	`Line(3,10,3,14,arrow)`
	`Rectangle(11,8,15,10)`
	`Rectangle(11,14,15,15)`
	`Line(13,10,13,14,arrow)`
	... etc. ...; 16 lines
		`for(i<3)`
		` line(7,1,5*i+2,3,arrow)`
		` for(j<i+1)`
		`  if(j>0)`
		`   line(5*j-1,9,5*i,5,arrow)`
		`  line(5*j+2,5,5*j+2,9,arrow)`
		` rectangle(5*i,3,5*i+4,5)`
		` rectangle(5*i,9,5*i+4,10)`
		`rectangle(2,0,12,1)`
			$\frac{16}{9} = 1.8\text{x}$

	`Circle(2,8)`
	`Rectangle(6,9, 7,10)`
	`Circle(8,8)`
	`Rectangle(6,12, 7,13)`
	`Rectangle(3,9, 4,10)`
	... etc. ...; 9 lines
		`reflect(y=8)`
		` for(i<3)`
		`  if(i>0)`
		`   rectangle(3*i-1,2,3*i,3)`
		`  circle(3*i+1,3*i+1)`
			$\frac{9}{5} = 1.8\text{x}$

To generate a training corpus for learning a policy which minimizes this loss, we synthesized minimum cost programs for each trace set of our hand drawings and for each σ. 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:

$$\pi_\theta(\sigma|T) \propto \exp\left(\phi_{\text{params}}(\sigma)^\top \theta \phi_{\text{trace}}(T)\right) \tag{4}$$

where $\phi_{\text{params}}(\sigma)$ is a one-hot encoding of the parameter settings of σ (see Tbl. 4) and $\phi_{\text{trace}}(T)$ extracts a few simple features of the trace set T; see supplement for details.

We contrast our learned search policy with two alternatives (see Fig. 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 σ 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 σ's contain minimum cost programs nor does it know how long each σ 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 synthesize 58% of programs within a minute.

Table 4: Parameterization of different ways of posing the program synthesis problem. The policy learns to choose parameters likely to quickly yield a minimal cost program. We slightly abuse notation by writing σ to mean an assignment to each of these parameters (so σ assumes one of 24 different values) and to mean the set of programs selected by that parameterization (so $\sigma \subseteq$ DSL)

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

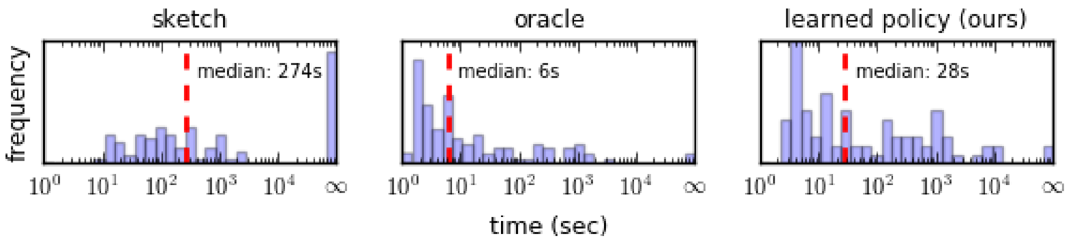


Figure 9: How long does it typically take to synthesize a minimum cost program? Sketch: out-of-the-box performance of the Sketch (Solar Lezama, 2008) program synthesizer. Oracle: baseline that picks the easiest synthesis problem containing a minimum cost program. Learned policy: a bias-optimal learned search policy running on an ideal timesharing machine. Oracle upper bounds the performance of any search policy. ∞ = timeout. Red dashed line is median time. Learned policy evaluated using 20-fold cross validation.

4 Applications of Graphics Program Synthesis

Why synthesize a graphics program, if the trace set 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 corrects errors made by the neural network by favoring trace sets 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 run the program synthesizer on the Top-k most likely trace sets output by the neurally guided sampler. Then, the system reranks the Top-k by the prior probability of their programs. The prior probability of a program is learned by picking the prior maximizing the likelihood of the ground truth trace sets; see supplement for details.

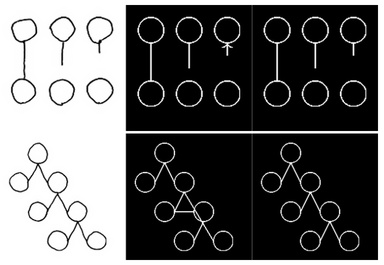


Figure 10: Left: hand drawings. Center: interpretations favored by the deep network. Right: interpretations favored after learning a prior over programs. Our learned prior favors shorter, simpler programs, thus (top example) continuing the pattern of not having an arrow is preferred, or (bottom example) continuing the "binary search tree" is preferred.

This procedure can only fix mistakes made by the neural network when a correct trace set is in the Top-k. Our sampler could only do better on 7/100 drawings by looking at the Top-5 or Top-100 samples (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. 10 for a representative example of the kinds of corrections that it makes. See supplement for details.

Close in program space, far in image space **Close in image space, far in program space**

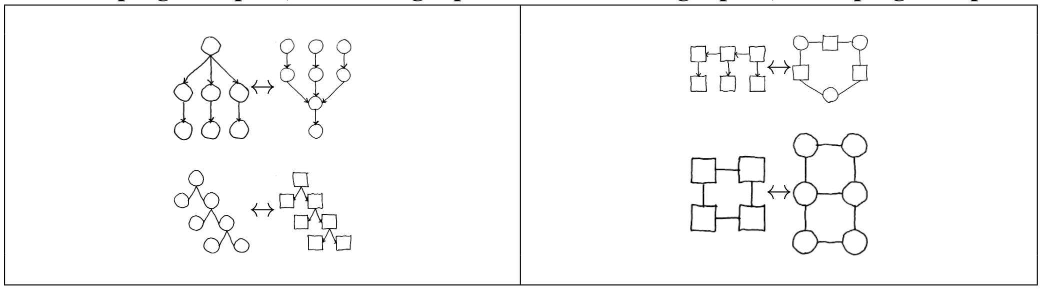


Figure 11: Pairs of images either close together or far apart in different features spaces. The symbol $\leftrightarrow$ points to the compared images. Features of the program capture abstract notions like symmetry and repetition. Distance metric over images is $L_{\text{learned}}(\cdot|\cdot)$ (see Sec. 2.1). Similarity of programs captures high-level features like repetition and symmetry, whereas similarity of images corresponds to similar drawing commands being in similar places. See Sec. 2 of the supplement for more examples.

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 lowest-cost programs that describe them. Our features are counts of the number of times that different components in the DSL were used (Tbl. 2). We then find drawings which are either close together or far apart in program feature space. 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 trace sets, $L_{\text{learned}}(\cdot|\cdot)$, to find drawings that are either close together or far apart according to the learned distance metric over images. Fig. 11 illustrates the kinds of drawings that these different metrics put closely together, while Sec. 2 of the supplement shows low dimensional projections of program and image features.

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 could change all of the circles to squares or make all of the lines be dashed. We extrapolate figures by increasing the number of times that loops are executed. Extrapolating repetitive visuals 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 produced by our system.

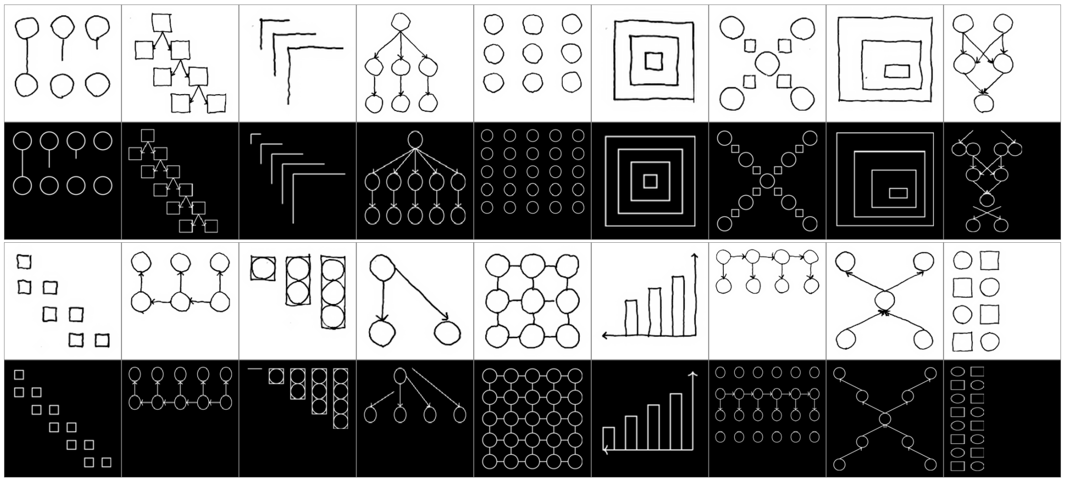


Figure 12: Top, white: hand drawings. Bottom, black: extrapolations produced by running loops for extra iterations.

5 Related work

Program Induction: Our approach to learning to search for programs draws theoretical underpinnings from Levin search (Levin, 1973; Solomonoff, 1984) and Schmidhuber's OOPS model (Schmidhuber, 2004). DeepCoder (Balog et al., 2016) is a recent model which, like ours, learns to predict likely program components. Our work differs because we treat the problem as *metareasoning*, identifying and modeling the trade-off between tractability and probability of success. TerpreT (Gaunt et al., 2016) systematically compares constraint-based program synthesis techniques against gradient-based search techniques, like those used to train Differentiable Neural Computers (Graves et al., 2016). The TerpreT experiments motivate our use of constraint-based techniques.

Deep Learning: Our neural network bears resemblance to the Attend-Infer-Repeat (AIR) system, which learns to decompose an image into its constituent objects (Eslami et al., 2016). AIR learns an iterative inference scheme which infers objects one by one and also decides when to stop inference. Our network differs in its architecture and training regime: AIR learns a recurrent auto-encoding model via variational inference, whereas our parsing stage learns an autoregressive-style model from randomly-generated (trace, image) pairs.

IM2LATEX (Deng et al., 2017) is a recent work that also converts images to LaTeX. Their goal is to derender LaTeX equations, which recovers a markup language representation. Our goal is to go from noisy input to a high-level program, which goes beyond markup languages by supporting programming constructs like loops and conditionals. Recovering a high-level program is more challenging than recovering markup because it is a highly under constrained symbolic reasoning problem.

Our image-to-trace parsing architecture builds on prior work on controlling procedural graphics programs (Ritchie et al., 2016). 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 trace through the guide program as a symbolic parse of the target image. We then show how to efficiently synthesize higher-level programs from these traces.

The Trace Hypothesis: The idea that an execution trace could assist in program learning goes back to the 1970's (Summers, 1977) and has been applied in neural models of program induction, like Neural Program Interpreters (Reed & de Freitas, 2015), or DeepCoder, which predicts what functions occur in the execution trace (Balog et al., 2016). Our contribution to this idea is the trace hypothesis: that trace sets can be inferred from perceptual data, and that the trace set is a useful 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 Contributions

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 trace sets; 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 the trace hypothesis, as realized in our two-phase system—parsing into trace sets, 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

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. URL `http://tensorflow.org/`. Software available from tensorflow.org.

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. DeepCoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, November 2016. URL `https://arxiv.org/abs/1611.01989`.

Tony Beltramelli. pix2code: Generating code from a graphical user interface screenshot. *CoRR*, abs/1705.07962, 2017. URL `http://arxiv.org/abs/1705.07962`.

Yuntian Deng, Anssi Kanervisto, Jeffrey Ling, and Alexander M. Rush. Image-to-markup generation with coarse-to-fine attention. In *ICML*, 2017.

Arnaud Doucet, Nando De Freitas, and Neil Gordon (eds.). *Sequential Monte Carlo Methods in Practice*. Springer, 2001.

SM Eslami, N Heess, and T Weber. Attend, infer, repeat: Fast scene understanding with generative models. arxiv preprint arxiv:..., 2016. 2016.

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.

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.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

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.

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, pp. 379–390, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4189-9. doi: 10.1145/2984511.2984575. URL `http://doi.acm.org/10.1145/2984511.2984575`.

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.

Max Jaderberg, Karen Simonyan, Andrew Zisserman, et al. Spatial transformer networks. In *Advances in Neural Information Processing Systems*, pp. 2017–2025, 2015.

Leonid Anatolevich Levin. Universal sequential search problems. *Problemy Peredachi Informatsii*, 9(3):115–116, 1973.

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.

Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. *ACM SIGPLAN Notices*, 50(10):107–126, 2015.

Scott Reed and Nando de Freitas. Neural programmer-interpreters. *CoRR*, abs/1511.06279, 2015. URL `http://arxiv.org/abs/1511.06279`.

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*, pp. 622–630, 2016.

Jürgen Schmidhuber. Optimal ordered problem solver. *Machine Learning*, 54(3):211–254, 2004.

Armando Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-177.html`.

Raymond J Solomonoff. Optimum sequential search. 1984.

Phillip D. Summers. A methodology for lisp program construction from examples. *J. ACM*, 24(1):161–175, January 1977. ISSN 0004-5411. doi: 10.1145/321992.322002. URL `http://doi.acm.org/10.1145/321992.322002`.

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*, pp. 3156–3164, 2015.

Jiajun Wu, Joshua B Tenenbaum, and Pushmeet Kohli. Neural scene de-rendering. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.