Neurally-Guided Procedural Models:
Learning to Guide Procedural Models with Deep Neural Networks

Daniel Ritchie*
Stanford University

Anna Thomas*
Stanford University

Pat Hanrahan*
Stanford University

Noah D. Goodman*
Stanford University

Guided
(N = 10 particles)

Unguided
(Equal N)

Unguided
(Equal Time)

Figure 1: (Top Row) Used as an importance sampler for Sequential Monte Carlo with \( N = 10 \) particles, our neurally-guided procedural models generate shape-matching results for each of the above letters in about a second. (Middle Row) The naïve, unguided procedural model does not converge to recognizable results using the same number of particles (\( N = 10 \)). (Bottom Row) The unguided model does better, but still does not reliably converge, when given the same amount of computation time as the guided model (\( \approx 1 \) sec).

Abstract

We present a deep learning approach for speeding up constrained procedural modeling. Probabilistic inference algorithms such as Sequential Monte Carlo (SMC) provide powerful tools for constraining procedural models, but they require many samples to produce desirable results. In this paper, we show how to create procedural models which learn how to satisfy constraints. We augment procedural models with neural networks: these networks control how the model makes random choices based on what output it has generated thus far. We call such a model a neurally-guided procedural model. As a pre-computation, we train these models to learn constraint-satisfying example outputs generated via SMC. They are then used as efficient importance samplers for SMC, generating high-quality results with very few samples. We evaluate our method on L-system-like models with image-based constraints. Given a desired quality threshold, neurally-guided models can generate satisfactory results up to 10x faster than unguided models.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Geometric algorithms, languages, and systems G.3 [Probability And Statistics]: Probabilistic algorithms (including Monte Carlo) I.2.6 [Artificial Intelligence]: Learning—Connectionism and neural nets

Keywords: Procedural Modeling, Probabilistic Programming, Sequential Monte Carlo, Deep Learning, Neural Networks

1 Introduction

*e-mail: {dritchie, thomasat, hanrahan, ngoodman}@stanford.edu
In this paper, we propose a method for automatically learning an approximation to such a perfect explicit model. Our method leverages advances in deep learning: it augments the procedural model with neural networks that control how the model makes random choices, based on what partial output the model has generated thus far. We call such a model a \textit{neurally-guided procedural model}. The neural networks are expressive enough to capture many implicit dependencies induced by the constraints.

We train neurally-guided procedural models using constraint-satisfying example outputs generated via SMC. Once trained, these models can be used as intelligent SMC important samplers. Our approach thus enables ‘bootstrapping’ samplers which train on their own outputs and become more efficient over time. Or, the system can invest time up-front generating and training on many examples, effectively ‘pre-compiling’ an efficient sampler.

We demonstrate our method through experiments with L-system-like procedural models with image-based soft constraints (Figure 1). For a given constraint satisfaction score threshold, our neurally-guided procedural model can generate results which reliably achieve that threshold using 10-20x fewer particles and up to 10x less compute time than an unguided procedural model.

In summary, our main contributions are:

- A general mathematical framework for defining and training neurally-guided procedural models that make implicit constraints into explicit generative processes.
- A specific implementation for L-system-like models with image-based constraints.
- Performance evaluation of neurally-guided image-constrained models, showing up to 10x speedups.

We give a high-level overview of our approach in Section 3 and then present the mathematical foundations of our method in Section 4. In Section 5, we describe how to implement neurally-guided procedural models with image-matching constraints. Finally, we evaluate the performance of those models in Section 6.

2 Related Work

Probabilistic Inference for Procedural Modeling Many research projects have used Bayesian probabilistic inference to control procedural models: constraining the shape of a 3D object [Talton et al. 2011; Ritchie et al. 2015b], creating functionally-plausible and aesthetically-pleasing furniture arrangements [Merrell et al. 2011; Yeh et al. 2012], coloring in patterns [Lin et al. 2013], and dressing virtual characters [Yu et al. 2012] are a few recent applications. Our work aims to make such systems more efficient: neurally-guided procedural models can capture many of the dependencies introduced by constraint likelihood functions, so samplers need fewer samples to find good results.

In recent work similar in spirit to our own, Dang and colleagues built a system which modifies a procedural grammar so that its output distribution reflects user preference scores given to example outputs [Dang et al. 2015]. Like us, they seek a model whose generative logic captures dependencies induced by a likelihood function (in their case, a Gaussian process regression over user-provided examples). Their method works by splitting non-terminal symbols in the original grammar, giving it more degrees of freedom to capture more dependencies. This approach works well for discrete dependencies, such as ensuring all floors of a building have the same architectural style. In contrast, our method captures dependencies using neural networks, making it better suited for complex, continuous constraint functions, such as shape-fitting.

Guided Procedural Modeling The shape of procedural models can be controlled using purely generative methods. The seminal work on open/environmentally-sensitive L-systems developed a formalism by which L-systems could query their spatial position and orientation [Prusinkiewicz et al. 1994; Méch and Prusinkiewicz 1996]. This ability allows them to prune their growth to an implicit surface. Recent follow-up work extends this technique to larger models by decomposing them into separate guide regions with limited interaction [Beneš et al. 2011]. These guide methods were carefully designed for the specific problem of fitting procedural models to shapes. In contrast, our method learns how to guide procedural models and is generally applicable to constraints which can be expressed as a likelihood function.

Neural Networks for Procedural Modeling Previous work has found other ways to apply neural networks to procedural modeling. One recent project uses neural networks as computationally inexpensive proxies for costly scoring functions in an inverse urban procedural modeling setting [Vanegas et al. 2012]. Another uses an autoencoder network to learn a low-dimensional representation space in which it is easy to explore the variability in a procedural model’s output [Yumer et al. 2015]. Our use of neural networks differs from both of the above projects, as we use them to capture constraint-induced dependencies via feedforward functions.

Neural Variational Inference Our method is also inspired by recent work in variational inference [Mnih and Gregor 2014; Rezende et al. 2014; Kingma and Welling 2014]. These algorithms use neural networks to define more expressive parametric families of probability distributions. They train stochastic deep belief networks and autoencoders, primarily modeling distributions over images for computer vision applications. Our method uses a different learning objective, and we focus on training procedural models with more complex recursive control flow.

The Neural Adaptive Sequential Monte Carlo algorithm is most similar to our method; it uses a similar learning objective and aims to train more efficient SMC importance samplers [Gu et al. 2015]. However, they focus on inference in time series models, such as nonlinear state space models.

3 Approach

In this section, we motivate and outline the process of creating, training, and using neurally-guided procedural models. Throughout this paper, we represent procedural models as probabilistic programs, i.e. programs that make random choices and support conditional inference queries about their distribution of outputs [Goodman and Stuhlmüller 2014].

3.1 Motivation

We motivation our approach using a simple program chain that recursively generates a random sequence of linear segments, constrained to match a target image. Figure 2a shows the text of this program, along with samples generated from it (drawn in black) against several target images (drawn in gray). Chains generated by running the program forward do not match the targets, since forward sampling is oblivious to the constraint. Instead, we can generate constrained samples using Sequential Monte Carlo (SMC) [Ritchie et al. 2015b]. SMC generates multiple samples, or particles, in parallel, resampling them at each step of the program to favor constraint-satisfying partial outputs. This results in final chains that more closely match the target images. However, the algorithm requires many particles—and therefore significant
function chain(pos, ang) {
  var newang = ang + gaussian(0, PI/8);
  var newpos = pos + polarToRect(LENGTH, newang);
  genSegment(pos, newpos);
  if (flip(0.5)) chain(newpos, newang);
}

function chain_neural(pos, ang) {
  var newang = ang + gaussMixture(nn1(...));
  var newpos = pos + polarToRect(LENGTH, newang);
  genSegment(pos, newpos);
  if (flip(nn2(...))) chain_neural(newpos, newang);

Forward Samples

(a)

SMC Samples

(N = 10)

(b)

Figure 2: Transforming a simple linear chain model into a neurally-guided procedural model. (a) The original program. When the program’s output (shown in black) is constrained to match a target image (shown in gray), forward sampling gives poor results. SMC sampling performs better but requires far more than 10 particles to achieve good results for all targets. (b) The neurally-guided program, where parameters of random choices are computed via neural networks. The neural nets receive the target image and all previous random choices as input (abstracted as "..."; see Figure 3b). Once trained, forward sampling from this program adheres closely to the target image, and SMC with 10 particles consistently produces good results.

computation—to produce acceptable results. Figure 2a shows that \( N = 10 \) particles is not sufficient.

In an ideal world, we would not need costly inference algorithms to generate constraint-satisfying results. Instead, we would have access to an ‘oracle’ program, \( \text{chain\_perfect} \), that perfectly fills in the target image when run forward. What form might this program take? At each step, it would need access to the target image, to know where to grow the chain next. It would also need to see the output it has already generated, to know when it has filled the target and can stop growing the chain.

Our insight is that while oracle programs such as \( \text{chain\_perfect} \) can be difficult or impossible to write by hand, it is possible to learn a program \( \text{chain\_neural} \) that comes close. Figure 2b shows our approach. For each random choice in the program text (e.g. \( \text{gaussian}, \text{flip} \)), we replace the parameters of that choice with the output of a neural network. This neural network’s inputs (abstracted as "...") include the target image as well the choices the program has made thus far. The network thus shapes the distribution over possible choices, guiding the program’s future output based on the target image and its past output. These neural nets affect both continuous choices (e.g. angles) as well as control flow decisions (e.g. recursion): they dictate where the chain goes next, as well as whether it keeps going at all. For continuous choices such as \( \text{gaussian} \), we also modify the program to sample from a mixture distribution. This helps the program handle situations where the constraints permit multiple distinct choices (e.g. in which direction to start the chain for the circle-shaped target image in Figure 2).

When properly trained, a neurally-guided procedural model such as \( \text{chain\_neural} \) generates constraint-satisfying results more efficiently than its un-guided counterpart. Figure 2b shows example outputs from \( \text{chain\_neural} \). Forward samples adhere closely to the target images, and SMC with 10 particles is sufficient to produce chains that fully fill the target shape. The next sections of the paper describe the process of building and training these neurally-guided procedural models in more detail.

3.2 System Overview

Figure 3 shows a high-level overview of our workflow for defining, training, and using neurally-guided procedural models. It consists of the following steps:

**Transform** The procedural model is first transformed by inserting one neural network for each random choice in the program text and turning continuous random choices into mixture distributions (Figure 3a-b). The network receives as input the constraint (e.g. a target image) and all previously-made random choices (shown grayed out in Figure 3a-b) and outputs the parameters for the choice (e.g. Gaussian means, variances, and mixture weights). We perform this transformation manually; it could be automated via source-to-source compilation. The neural networks can capture multiple different constraints, but an appropriate architecture for them depends on the generative paradigm and the output domain of the procedural model (e.g. images, 3D models, etc.) In Section 5, we present an architecture for 2D L-system-like procedural models which generate images. In particular, we describe how our implementation converts the previous random choices into a fixed-width vector appropriate for input to a neural net.

**Generate** Given a constraint, such as a target image, Sequential Monte Carlo generates samples from the constrained procedural model (Figure 3c). Our system uses the version of SMC for probabilistic programs presented by Ritchie et al. [2015b], where particles are resampled after the program generates a new piece of geometry. It also uses the trained models as importance samplers for this SMC algorithm when generating final results.

**Train** The generated samples are then used to train the neural networks: the desired outcome is a set of network parameters that make the model more likely to generate these samples when run forward. We derive the learning objective in Section 4 and the details of our stochastic gradient learning method in Section 5. The trained neurally-guided model can then quickly generate more samples, which can serve as further training data for refining the model, if desired.
4 Mathematical Foundations

Having outlined our approach, we now formally define neurally-guided procedural models. For our purposes, a procedural model is a generative probabilistic model of the following form:

\[
P_M(x) = \prod_{i=1}^{|x|} p_i(x_i; \Phi_i(x_1, \ldots, x_{i-1}))
\]

Here, \(x\) is the vector of random choices the model makes as it executes (the dimensionality of \(x\) may be variable, as with recursive procedural models such as stochastic L-systems). The \(p_i\)'s are local probability distributions from which each successive random choice is drawn. \(p_i\) is parameterized by a set of parameters (e.g. mean and variance, for a Gaussian distribution), which are determined by some function \(\Phi_i\) of the previous random choices \(x_1, \ldots, x_{i-1}\). The total probability density is the product of these local probabilities, according to the chain rule.

A constrained procedural model is a procedural model whose probability distribution is modulated by some likelihood function \(\ell(x, c)\), i.e. a scoring function indicating how well an output of the model satisfies some constraint \(c\). For example, \(c\) could be an image, with \(\ell(\cdot, c)\) measuring similarity to that image. By Bayes’ rule:

\[
P_{CM}(x|c) = \frac{1}{Z} \cdot P_M(x) \cdot \ell(x, c)
\]

where \(Z\) is a normalizing constant. The set of all constraints \(c\) supported by the procedural model forms the constraint space \(C\) (e.g. all images, all binary mask images, etc.)

A neurally-guided procedural model modifies a procedural model by, for each local probability \(p_i\), replacing the parameter function \(\Phi_i\) with a neural network:

\[
P_{CM}(x|c; \theta) = \prod_{i=1}^{|x|} \tilde{p}_i(x_i; \text{NN}_i(x_1, \ldots, x_{i-1}, c; \theta))
\]

The neural nets receive the previous random choice values and the constraint as input, and are themselves parameterized by \(\theta\). \(\tilde{p}_i\) is a mixture distribution if random choice \(i\) is continuous; otherwise, \(\tilde{p}_i = p_i\).

In training a neurally-guided procedural model, our goal is to find the parameters \(\theta\) such that \(P_{CM}\) is as close as possible to \(P_M\) for all supported constraints. Formally, we seek to minimize the conditional KL divergence \(\text{KL}(P_{CM}||P_M)\). Given some prior distribution \(P(c)\) over constraints \(c \in C\), our optimization objective is:

\[
\min_{\theta} \text{KL}(P_{CM}||P_M) = \min_{\theta} \mathbb{E}_{P(c)} \left[ \mathbb{E}_{P_{CM}|c}(x) \left[ \log \frac{P_{CM}(x|c)}{P_M(x|c; \theta)} \right] \right]
\]

In the last step, we approximate the expectations with an average over a finite set of samples \(x_i, c_i\) drawn from the constrained procedural model \(P_{CM}\) using SMC and the constraint prior \(P(c)\). If we view these samples as a training data set, then this optimization objective is simply maximizing the log-likelihood of the training data under the neurally-guided model \(P_M\).

With such a training set in hand, optimization proceeds via stochastic gradient ascent using the gradient

\[
\nabla \log P_{CM}(x;c; \theta) = \sum_{i=1}^{|x|} \nabla \log \tilde{p}_i(x_i; \text{NN}_i(x_1, \ldots, x_{i-1}, c; \theta))
\]

Is is worth noting that \(\text{KL}(P_{CM}||P_M)\) is not the only measure of distance between probability distributions we could have used. In particular, several related works have used the other direction of KL divergence, \(\text{KL}(P_M||P_{CM})\), due to its attractive properties: it requires training samples from \(P_{CM}\), which are much less expensive to generate than samples from \(P_M\). It is the optimization objective used in many variational inference algorithms [Wingate and Weber 2012; J. Manning and Blei 2014; Mnih and Gregor 2014] as well as the REINFORCE algorithm for reinforcement learning [Williams 1992]. When used for procedural modeling, however, this objective leads to models whose outputs lack diversity, making them unsuitable for generating visually-varied content. This behavior is due to a well-known property of the objective: minimizing it produces approximating distributions that are overly-compact, i.e. concentrating their probability mass in a smaller volume of the state space than the true distribution being approximated [MacKay 2002].
5 Implementation

In this section, we describe an implementation of neurally-guided accumulative procedural models: models that iteratively or recursively add new geometry to a structure. Most growth models, such as L-systems, are accumulative [Prusinkiewicz and Lindenmayer 1990]. This is in contrast with other modeling paradigms: spatial subdivision, such as architectural split grammar [ Müller et al. 2006]; object subdivision, such as fractal terrain [Lewis 1987]; or simulation, such as erosion-based terrain [Št’ava et al. 2008]. For our purposes, a procedural model is accumulative if, while executing, it provides a ‘current position’: a point \(p\) from which geometry generation will continue. We focus on 2D models \((p \in \mathbb{R}^2)\), though the techniques we present extend naturally to 3D.

We first describe the neural network architecture used by the neurally-guided models before giving details on how we train them.

5.1 Network Architecture

Our neural networks take as input the constraint \(c\) (in this case, a target image) and all previously-made random choices, and output the parameters of a random choice. Figure 4 shows our network architecture. We use a multilayer perceptron (MLP) architecture, because it is simple, easy to scale, and is a universal function approximator [Rumelhart et al. 1986; Cybenko 1989]. Our MLP takes \(n_f\) inputs, has one hidden layer of size \(n_f/2\) with a \(\tanh\) nonlinearity, and has \(n_p\) outputs, where \(n_p\) is the number of parameters the random choice expects. Since some parameters are bounded (e.g. Gaussian variance must be positive), each output is remapped via an appropriate bounding transform (e.g. \(e^x\) for non-negative parameters). We experimented with more hidden layers but found that this did not improve performance.

The inputs for the MLP come from several sources, each providing the network with decision-critical information. All of these features can be computed from the target image and the choices-so-far; for efficiency, we compute them incrementally as the program runs.

Local State Features The first set of relevant data is the local state of the procedural model: its current position \(p\), the current orientation of any local reference frame, its current recursion depth, etc. Our networks access this information via the arguments of the function in which the random choice occurs. We extract all \(n_a\) scalar arguments, normalize them to \([-1, 1]\), and pass them to the MLP.

Target Image Features To make appropriate decisions for matching a target image, the network must have access to that image. The raw pixels provide too much data; we need to summarize the relevant image contents. Convolutional neural networks reduce an image to a fixed-width feature vector but are aimed at classification tasks: they detect features but are intentionally invariant to where those features occur [Krizhevsky et al. 2012]. Instead, we use a different, location-sensitive architecture. We extract a 3x3 window of pixels around the model’s current position \(p\). We do this at four different resolution levels, with each level computed by convolving the previous level with a 3x3 kernel and then downsampling via a 2x2 box filter. For a 2D image with channel depth \(c\), this results in 36c features. Together, these features summarize what the target image looks like from the procedural model’s current position, where resolution decreases with distance. This architecture is similar to the foveated ‘glimpses’ used in recent work on neural models of visual attention [Mnih et al. 2014].

Partial Output Features The target image features provide the network with information it needs to generate matching content with accuracy (e.g. how to stay within a target shape) However, they do not provide the information necessary to achieve completeness (e.g. how to completely fill in a target shape). To give the network this capability, we also extract multi-resolution windows from the partial output image generated by the procedural model thus far. This adds another 36c input features.

The parameters \(\theta\) of this architecture consist of the weights and biases for both fully-connected layers in the MLP, as well as the kernel weights and biases for the three convolution + downsampling layers.
As shown in Equation 1, each training sample from a procedural machine with 16GB RAM running OSX 10.10.5.

6.1 Image Datasets

We wrote a WebPPL program which recursively generates vines with leaves and flowers and then trained a neurally-guided version of this program to capture the above likelihood. The model was trained on 10000 sample traces, each generated using SMC with 600 particles. Target images were drawn uniformly at random from the Scribbles dataset. Each sample took on average 17 seconds to generate; parallelized across four CPU cores, the entire set of samples took approximately 12 hours to generate (later in this section, we show that far fewer samples are actually needed). Training took 55 minutes in our single-threaded Javascript implementation. These times could be reduced with more efficient implementations (e.g. leveraging GPUs for training).

Figure 1 shows example outputs from the vines program. The weighting scheme of $\ell_{\text{shape}}$ causes the geometry to adhere to target shape contours, making the shape recognizable without cluttering interior regions and obscuring the vines’ structural characteristics. This behavior is not easy to achieve with a purely generative space-filling approach such as environmentally-sensitive L-systems [Prusinkiewicz et al. 1994], but it is simple to specify with constraints. The top row outputs were generated using 10-particle

6.2 Shape Matching

In our first set of experiments, we train neurally-guided procedural models to capture 2D shape matching constraints, where the target shape is specified as a binary mask image. If $D$ is the spatial domain of the image, and $I(x)$ is the function which renders the current partial output defined by random choices $x$, then the likelihood function for this constraint is

$$\ell_{\text{shape}}(x, c) = \mathcal{N} \left( \frac{\text{sim}(I(x), c) - \text{sim}(0, c)}{1 - \text{sim}(0, c)}, 1, \sigma_{\text{shape}} \right)$$

$$\text{sim}(I_1, I_2) = \sum_{p \in D} w(p) \cdot \mathbb{1} \{ I_1(p) = I_2(p) \}$$

$$w(p) = \begin{cases} 1 & \text{if } I_2(p) = 0 \\ 1 & \text{if } ||\nabla I_2(p)|| = 1 \\ w_{\text{filled}} & \text{if } ||\nabla I_2(p)|| = 0 \end{cases}$$

where $\mathcal{N}$ is the normal distribution. This function encourages the rendered image to be similar to the target mask, where similarity is normalized against the target’s similarity to an empty image $0$. Each pixel $p$’s contribution to the similarity is weighted by $w(p),$ determined by whether the target mask is empty, filled, or has an edge at that pixel. We use $w_{\text{filled}} = 0.6,$ so that empty and edge pixels are worth 1.5 times as much as filled pixels. This encourages the program to match perceptually-important contours before filling in flat regions. We set $\sigma_{\text{shape}} = 0.02$ in all experiments.

When using these images for training, we augment the datasets by also including a horizontally-mirrored duplicate of each image. We also annotate each image with a starting point and starting direction from which to initialize the execution of a procedural model. Figure 5 shows some representative images from each collection.

5.2 Training

We train neurally-guided procedural models by stochastic gradient ascent using the gradient in Equation 2. Our system computes this gradient via backpropagation from the log-p’s to the neural network parameters $\theta$. We use the Adam algorithm for stochastic gradient optimization, with $\alpha = \beta = 0.75$ and an initial learning rate of 0.01 [Kingma and Ba 2015]. We found that a mini-batch size of one worked best in our experiments: more frequent gradient updates led to faster convergence than less-frequent-but-less-noisy updates. We terminate training after 20000 gradient updates.

5.3 Implementation Details

We implemented our prototype system in the Javascript-based probabilistic programming language WebPPL [Goodman and Stuhlmüller 2014], with neural networks implemented using an open-source Javascript library for neural computation. The source code for our system is available at [LINK ANONYMIZED].

6 Experiments

In this section, we qualitatively and quantitatively evaluate how well our neurally-guided procedural models capture image-based constraints. We first describe our databases of target images before presenting the details of several experiments. All timing data reported in this section was collected on an Intel Core i7-3840QM machine with 16GB RAM running OSX 10.10.5.

6.1 Image Datasets

As shown in Equation 1, each training sample from a procedural model must be paired with a constraint $c$ drawn from a prior $P(c)$ over possible constraints. During training, we sample target images uniformly at random from a database of training images. In our experiments, we use the following image collections:

- **Scribbles:** A set of 49 binary mask images drawn by hand with the brush tool in Photoshop. These were designed to cover a range of possible shapes with thick and thin regions, high and low curvature, and different self-intersections.
- **Glyphs:** A subset of 197 glyphs from the FF Tartine Script Bold typeface. Consists of all glyphs which have only one foreground connected component and at least 500 foreground pixels when rendered at 129x97.
- **PhyloPic:** A set of 35 images from PhyloPic, a database of silhouettes for plants, animals, and other organisms.

When using these images for training, we augment the datasets by also including a horizontally-mirrored duplicate of each image. We also annotate each image with a starting point and starting direction from which to initialize the execution of a procedural model. Figure 5 shows some representative images from each collection.

6.2 Shape Matching

In our first set of experiments, we train neurally-guided procedural models to capture 2D shape matching constraints, where the target shape is specified as a binary mask image. If $D$ is the spatial domain of the image, and $I(x)$ is the function which renders the current partial output defined by random choices $x$, then the likelihood function for this constraint is

$$\ell_{\text{shape}}(x, c) = \mathcal{N} \left( \frac{\text{sim}(I(x), c) - \text{sim}(0, c)}{1 - \text{sim}(0, c)}, 1, \sigma_{\text{shape}} \right)$$

$$\text{sim}(I_1, I_2) = \sum_{p \in D} w(p) \cdot \mathbb{1} \{ I_1(p) = I_2(p) \}$$

$$w(p) = \begin{cases} 1 & \text{if } I_2(p) = 0 \\ 1 & \text{if } ||\nabla I_2(p)|| = 1 \\ w_{\text{filled}} & \text{if } ||\nabla I_2(p)|| = 0 \end{cases}$$

where $\mathcal{N}$ is the normal distribution. This function encourages the rendered image to be similar to the target mask, where similarity is normalized against the target’s similarity to an empty image $0$. Each pixel $p$’s contribution to the similarity is weighted by $w(p),$ determined by whether the target mask is empty, filled, or has an edge at that pixel. We use $w_{\text{filled}} = 0.6,$ so that empty and edge pixels are worth 1.5 times as much as filled pixels. This encourages the program to match perceptually-important contours before filling in flat regions. We set $\sigma_{\text{shape}} = 0.02$ in all experiments.

We wrote a WebPPL program which recursively generates vines with leaves and flowers and then trained a neurally-guided version of this program to capture the above likelihood. The model was trained on 10000 sample traces, each generated using SMC with 600 particles. Target images were drawn uniformly at random from the Scribbles dataset. Each sample took on average 17 seconds to generate; parallelized across four CPU cores, the entire set of samples took approximately 12 hours to generate (later in this section, we show that far fewer samples are actually needed). Training took 55 minutes in our single-threaded Javascript implementation. These times could be reduced with more efficient implementations (e.g. leveraging GPUs for training).

Figure 5 shows some representative images from each collection.

1https://github.com/dritchie/adnn

2http://phylopic.org
SMC with the trained neurally-guided model, which reliably produces recognizable results. In contrast, 10-particle SMC with the unguided model produces totally unrecognizable results (middle row). Because the neural networks make the guided model more computationally-expensive to evaluate, a more equitable comparison is to give the unguided model the same amount of wall-clock time as the guided model—this corresponds to \( \sim 50 \) particles, in this case (bottom row). While the resulting outputs fare better, the target shape is still obscured. We should also note that the unguided model is unpredictable at such low particle counts; results of even this quality took many tries to obtain at 50 particles. In practice, we find that the unguided model needs \( \sim 200 \) particles to reliably match the performance of the guided model. Figure 7 shows more outputs from the vines program, and Figure 8 shows example outputs from a neurally-guided procedural lightning program.

Figure 6 shows a quantitative comparison between five different models on the shape matching task:

- **Unguided**: The original, unguided procedural model.
- **Constant Params**: The neural network for each random choice is simply a set of constant parameters; training this model finds the optimal constants. This is also known as a partial mean field approximation [Wingate and Weber 2012].
- **+ Local State Features**: Adding the local state features described in Section 5.
- **+ Target Image Features**: Adding the target image features described in Section 5.
- **All Features**: The full neural net architecture described in Section 5, including local state features, target image features, and partial output features.

We test each model on the images in the Glyph dataset and report the median normalized similarity-to-target achieved (i.e. argument one to the Gaussian in Equation 3). Figure 6a plots this average similarity as the number of SMC particles increases. The performance of the neurally-guided models improves with the addition of more features; at 10 particles, the full model is already near the peak performance asymptote. Figure 6b shows the wall-clock time each method requires as the desired average similarity increases. The vertical gap between the two curves shows the speedup given by neural guidance, which can be as high as 10x. Note that we trained on the Scribbles dataset but tested on the Glyphs dataset; these results suggest that our models can generalize to qualitatively-different previously-unseen images.

Figure 9 shows the benefit of using mixture distributions for continuous random choices in the guided model. The experimental setup is the same as in Figure 6. We compare a model which uses 4-component mixture distributions with a no-mixture model. The with-mixtures model provides a noticeable performance boost, which we alluded to in Section 3: when matching complex shapes with junctions and intersections, such as the crossing of the letter ‘t’, the program benefits from modeling uncertainty at these points with multi-modal distributions. We found 4 mixture components sufficient for our examples.
| Target | “Ground Truth” | Guided | Unguided (Equal N) | Unguided (Equal Time) |
|--------|----------------|--------|--------------------|-----------------------|
| ![Target](image) | ![Ground Truth](image) | ![Guided](image) | ![Unguided (Equal N)](image) | ![Unguided (Equal Time)](image) |
| ![Target](image) | ![Ground Truth](image) | ![Guided](image) | ![Unguided (Equal N)](image) | ![Unguided (Equal Time)](image) |
| ![Target](image) | ![Ground Truth](image) | ![Guided](image) | ![Unguided (Equal N)](image) | ![Unguided (Equal Time)](image) |

**Figure 7:** Using Sequential Monte Carlo to make a vine-growth procedural model match target images. $N$ is the number of SMC particles used. The “Ground Truth” column shows an example result after running SMC with the unguided model with a large number of particles ($N = 600$); these images represent the best possible result for a given target. Our neurally-guided procedural models can generate results of nearly this quality in a couple seconds; the unguided model struggles given the same number of particles or the same computation time.
We also investigate how the number of training samples affects performance. Figure 10 plots the median similarity at 10 particles as training set size increases. Performance increases rapidly for the first few hundred samples before appearing to level off (the noise in the curve is due to randomness in neural net training initialization). This suggests that ∼1000 sample traces is sufficient, which may seem surprising, as many published deep learning systems require millions of training examples [Krizhevsky et al. 2012]. In our case, each training trace contains up to thousands of random choices, each of which provides a learning signal—in this way, the training data is “bigger” than it appears. Our implementation can generate 1000 samples in just over an hour using four CPU cores. As mentioned previously, this time could be reduced by ‘bootstrapping’ the system: training on smaller subsets of data and using the partially-learned model to generate further data faster.

6.3 Stylized “Circuit” Design

Thus far, we have focused on image-matching constraints. However, the architecture we have presented can learn other types of image-based constraints. In this section, we constrain the vines program to generate outputs which resemble stylized circuit designs.

Dense packing of long wire traces is one of the most striking visual characteristics of circuit boards. To achieve dense packing, we encourage the program to fill a certain percentage $\tau$ of the image ($\tau = 0.5$ in the subsequent results). To mimic the appearance of traces, we encourage the output image to have a dense, high-magnitude gradient field, as the vines program can best achieve this result by creating many long rectilinear or diagonal edges. These constraints result in the following likelihood:

$$\ell_{\text{circ}}(x) = \mathcal{N}(\text{edge}(I(x)) \cdot (1 - \eta(\text{fill}(I(x)), \tau)), 1, \sigma_{\text{circ}})$$

$$\text{edge}(I) = \frac{1}{|D|} \sum_{p \in D} ||\nabla I(p)||$$

$$\text{fill}(I) = \frac{1}{|D|} \sum_{p \in D} I(p)$$

Figure 10: The effect of training set size on performance (at 10 SMC particles), plotted on a logarithmic scale. Average similarity-to-target increases sharply for the first few hundred sample training traces, then appears to plateau at around 1000 traces. Noise in the plot is due to randomness in neural net training, as different training sessions converge to different local optima of the learning objective function.

To guide this program, we use the same architecture as before, minus the target image features (since there is no target image). We train the neurally-guided model using 2000 traces generated using SMC with 600 particles. Sample generation took about 10 hours on four CPU cores, and training took just under two hours. Figure 11 shows some outputs from this program, and Figure 12 shows a performance comparison between unguided and neurally-guided models for this task. As with the shape matching examples, the neurally-guided model generates high-scoring results significantly faster than the unguided model.

7 Discussion and Future Work

This paper introduced neurally-guided procedural models: constrained procedural models that use neural networks to capture constraint-induced dependencies. We developed a mathematical framework for defining and training such models. We also described a specific neural architecture for accumulative models that generate images. Finally, we evaluated the performance of neurally-guided models, demonstrating that they can generate high-quality results significantly faster than unguided models.

7.1 Limitations

One limitation of our system is its need for training data, which must be generated via expensive inference. This can be a significant up-front cost, especially for computationally-expensive models. Thus, our method is not well-suited for scenarios where the procedural model changes rapidly, such as speeding up the inner loop of a development and debugging cycle. Instead, it is best suited for scenarios where the model is fixed, such as deploying a finalized procedural model as part of a design tool. It may be particularly attractive for online, multi-user deployments, where the system can gather example results from the community, periodically retrain, and push the updated procedural model to users.
Our method is also not well-suited for capturing hard constraints, which some visual effects necessitate (e.g. symmetries), as it requires a continuous probability for each training sample. While hard constraints can sometimes be usefully approximated with tight soft constraints, neural networks such as ours are best at approximating noisy and/or random functions, not precise, deterministic relationships. Other techniques are needed for generatively capturing these kinds of constraints.

7.2 Future Work

The neural guide architecture we presented applies to image-generating programs. How might we extend it to other output domains? The architecture must compactly represent the model’s partial output at any point in the program. For 3D modeling, our architecture extends naturally to 3D, e.g. using voxel grids instead of images. For other output domains, it may be possible to develop architectures that learn a partial output state representation, as in recent work on recurrent sequence generation [Graves 2013].

Our architecture also focuses on accumulative procedural models, but applications of other generative paradigms could also benefit from neural guidance. One example is texture generation, which repeatedly generates content across its entire domain, often in a multiscale fashion. In such a setting, the guide model cannot rely on a “current position” for extracting decision-critical features. It might instead learn what parts of the current partial output are relevant, as recently-developed visual attention models learn where to look in an image to make classification decisions [Mnih et al. 2014].

Our method could also be extended to inference algorithms beyond Sequential Monte Carlo. For example, Markov Chain Monte Carlo works better when the likelihood of a partial output is not predictive of the likelihood of the final output (e.g. generating stable arches, where the structure is unstable until finished). Just as neurally-guided models are efficient importance samplers for SMC, they might also serve as efficient proposal distributions for MCMC.

By adding neural networks which predict random choice parameters, we have only explored one simple program transformation to enable constraint capture. More extensive transformations, such as changing the control flow of the program, may be necessary for capturing especially complex constraints. One step in this direction would be to combine our approach with the grammar-splitting technique of Dang et al. [2015].

Ultimately, we envision a future in which procedural models learn to encode complex constraints using purely generative methods, so that forward sampling alone produces beautiful results. The work we have presented in this paper takes a step toward that goal, but it is only the beginning. We hope that our work inspires other researchers to develop new and better neural network architectures and program transformations to attack this problem.
References

BENEŠ, B., ŠAVA, O., MÉCH, R., AND MILLER, G. 2011. Guided Procedural Modeling. In Eurographics 2011.

BROOKS, S., GELMAN, A., JONES, G., AND MENG, X. 2011. Handboook of Markov Chain Monte Carlo. CRC Press.

CYBENKO, G. 1989. Approximation by superpositions of a sigmoidal function. Mathematics of Control, Signals and Systems.

DANG, M., LIENHARD, S., CEYLAN, D., NEUBERT, B., WONKA, P., AND PAULY, M. 2015. Interactive Design of Probability Density Functions for Shape Grammars.

DOUCET, A., DE FREITAS, N., AND GORDON, N., Eds. 2001. Sequential Monte Carlo Methods in Practice. Springer.

GOODMAN, N. D., AND STUHLMÜLLER, A., 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org. Accessed: 2015-12-23.

GRAVES, A. 2013. Generating Sequences With Recurrent Neural Networks. CoRR abs/1308.0850.

GU, S., GHAHRAMANI, Z., AND TURNER, R. E. 2015. Neural Adaptive Sequential Monte Carlo. In NIPS 2015.

J. MANNING, R. RANGANATH, K. N., AND BLEI, D. 2014. Black Box Variational Inference. In AISTATS 2014.

KINGMA, D. P., AND BA, J. 2015. Adam: A Method for Stochastic Optimization. In ICLR 2015.

KINGMA, D. P., AND WELLING, M. 2014. Auto-Encoding Variational Bayes. In ICLR 2014.

KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In NIPS 2012.

LEWIS, J. P. 1987. Generalized Stochastic Subdivision. ACM Trans. Graph. 6, 3.

LIN, S., RITCHIE, D., FISHER, M., AND HANRAHAN, P. 2013. Probabilistic Color-by-Numbers: Suggesting Pattern Colorizations Using Factor Graphs. In SIGGRAPH 2013.

MACKAY, D. J. C. 2002. Information Theory, Inference & Learning Algorithms. Cambridge University Press.

MARES, J., ANDALMAN, B., BEARDSLEY, P. A., FREEMAN, W., GIBSON, S., HODGINS, J., KANG, T., MIRTICH, B., PFISTER, H., Ruml, W., RYALL, K., SEIMS, J., AND SHIEBER, S. 1997. Design galleries: A general approach to setting parameters for computer graphics and animation. In SIGGRAPH 1997.

MERRELL, P., SCHKUFZA, E., LI, Z., AGRAWALA, M., AND KOLTUN, V. 2011. Interactive Furniture Layout Using Interior Design Guidelines. In SIGGRAPH 2011.

MNIH, A., AND GREGOR, K. 2014. Neural Variational Inference and Learning in Belief Networks. In ICML 2014.

MNIH, V., HEESS, N., GRAVES, A., AND KAVUKCUOGLU, K. 2014. Recurrent Models of Visual Attention. In NIPS 2014.

MÜLLER, P., WONKA, P., HAEGLER, S., ULMER, A., AND VAN GOOL, L. 2006. Procedural Modeling of Buildings. In SIGGRAPH 2006.

MÉCH, R., AND PRUSINKIEWICZ, P. 1996. Visual Models of Plants Interacting with Their Environment. In SIGGRAPH 1996.

PRUSINKIEWICZ, P., AND LINDENMAYER, A. 1990. The Algorithmic Beauty of Plants. Springer-Verlag New York, Inc.

PRUSINKIEWICZ, P., JAMES, M., AND MÉCH, R. 1994. Synthetic Topiary. In SIGGRAPH 1994.

REZENDE, D. J., MOHAMED, S., AND WIERSTRA, D. 2014. Stochastic Backpropagation and Approximate Inference in Deep Generative Models. In ICML 2014.

RITCHIE, D., LIN, S., GOODMAN, N. D., AND HANRAHAN, P. 2015. Generating Design Suggestions under Tight Constraints with Gradient-based Probabilistic Programming. In Eurographics 2015.

RITCHIE, D., MILDENHALL, B., GOODMAN, N. D., AND HANRAHAN, P. 2015. Controlling Procedural Modeling Programs with Stochastically-Ordered Sequential Monte Carlo. In SIGGRAPH 2015.

RUMELHART, D. E., HINTON, G. E., AND WILLIAMS, R. J. 1986. Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1. MIT Press, ch. Learning Internal Representations by Error Propagation.

SCHWARZ, M., AND WONKA, P. 2014. Procedural Design of Exterior Lighting for Buildings with Complex Constraints. ACM Trans. Graph. 33, 5.

STAVA, O., PIRK, S., KRATT, J., CHEN, B., MCH, R., DEUSSEN, O., AND BENES, B. 2014. Inverse Procedural Modelling of Trees. Computer Graphics Forum 33, 6.

TALTON, J. O., LOU, Y., LESSER, S., DUKE, J., MÉCH, R., AND KOLTUN, V. 2011. Metropolis Procedural Modelling. ACM Trans. Graph. 30, 2.

VANEGAS, C. A., GARCIA-DORADO, L., ALIAGA, D. G., BENES, B., AND WADDELL, P. 2012. Inverse Design of Urban Procedural Models. In SIGGRAPH Asia 2012.

ŠT’AVA, O., BENEŠ, B., BRISBIN, M., AND KŘIVÁNEK, J. 2008. Interactive Terrain Modeling Using Hydraulic Erosion. In SCA 2008.

WILLIAMS, R. J. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. Machine Learning 8.

WINGATE, D., AND WEBER, T. 2012. Automated Variational Inference in Probabilistic Programming. In NIPS 2012 Workshop on Probabilistic Programming.

WONG, M. T., ZONGKER, D. E., AND SALSEN, D. H. 1998. Computer-generated Floral Ornament. In SIGGRAPH 1998.

YEH, Y.-T., YANG, L., WATSON, M., GOODMAN, N. D., AND HANRAHAN, P. 2012. Synthesizing Open Worlds with Constraints Using Locally Annealed Reversible Jump MCMC. In SIGGRAPH 2012.

YU, L.-F., YEUNG, S.-K., TERZOPoulos, D., AND CHAN, T. F. 2012. DressUp!: Outfit Synthesis Through Automatic Optimization. In SIGGRAPH Asia 2012.

YUMER, M. E., ASENTE, P., MECH, R., AND KARA, L. B. 2015. Procedural Modeling Using Autoencoder Networks. In UIST 2015.

ZHU, L., XU, W., SNYDER, J., LIU, Y., WANG, G., AND GUO, B. 2012. Motion-guided Mechanical Toy Modeling. In SIGGRAPH Asia 2012.