

Interpretable Actions: Controlling Experts with Understandable Commands

Shumeet Baluja, David Marwood, Michele Covell

Google Research, Google, Inc.
{shumeet,marwood,covell}@google.com

Abstract

Despite the prevalence of deep neural networks, their single most cited drawback is that, even when successful, their operations are inscrutable. For many applications, the desired outputs are the composition of externally-defined bases. For such decomposable domains, we present a two-stage learning procedure producing combinations of the external bases which are trivially extractable from the network. In the first stage, the set of external bases that will form the solution are modeled as differentiable generator modules, controlled by the same parameters as the external bases. In the second stage, a controller network is created that selects parameters for those generators, either successively or in parallel, to compose the final solution. Through three tasks, we concretely demonstrate how our system yields readily understandable commands. In one, we introduce a new form of artistic style transfer, learning to draw and color with crayons, in which the transformation of a photograph or painting occurs not as a single monolithic computation, but by the composition of thousands of individual, visualizable strokes. The other two tasks, single-pass function approximation with arbitrary bases and shape-based synthesis, show how our approach produces understandable and extractable actions in two disparate domains.

1 Introduction

Perhaps the longest-standing drawback of neural networks has been their lack of interpretability (Chakraborty et al. 2017; Fan, Xiong, and Wang 2020; Harnad 1990; Smolensky 1986). This work improves interpretability using a training approach that naturally lends itself to understandable actions and transformations. For many domains, the target solution is the composition of externally-defined operations or *external bases*. The bases can be complex, non-differentiable, or otherwise ill-behaved. In our approach, these external bases are individually approximated by differentiable deep neural networks (DNNs) called *generator modules*. The generators are frozen and a *controller* DNN is trained to output *interpretable control commands* to the generators. Under the guidance of the controller, the system composes the output of the generators to create the final output. The control commands are trivial to extract from the network and reveal how to construct the output using solely the external bases.

We demonstrate this technique through three tasks:

Copyright © 2021, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

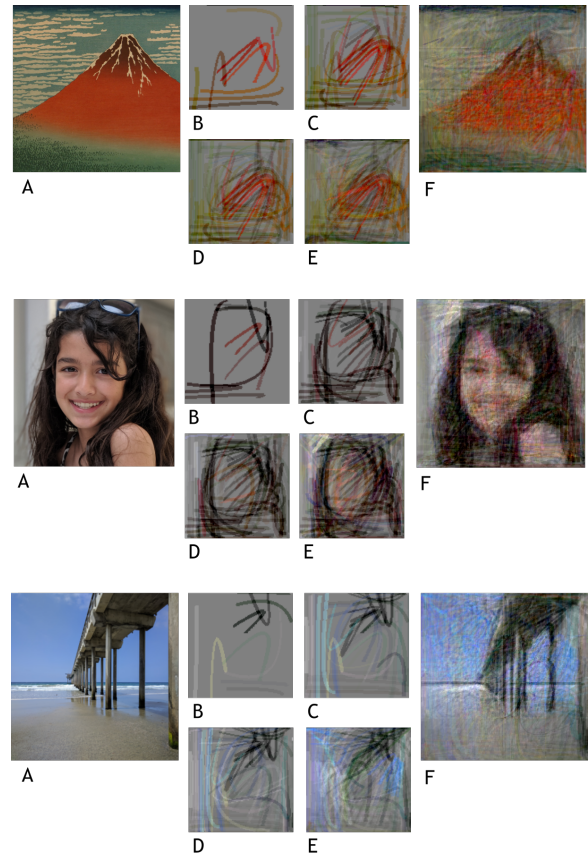


Figure 1: A network recreates an image with individual crayon strokes. A: Original. B: first 10 crayon strokes. C-E: successive strokes. Clear representation of dominant image features are created quickly. F: Final result. Top-Bottom: A painting by Hokusai, a portrait and a landscape photograph.

- **Function Approximator:** Composes an arbitrarily complex input function using simple external bases.
- **Style Transfer:** Thousands of individual crayon strokes are composed to stylistically reproduce an image (see Figure 1). This introduces a novel approach to style transfer.
- **Inverse Renderer:** A target image is deconstructed into simple geometric shapes.

In all the tasks, though simple, the external bases are not well-suited to differentiation or classic optimization. Our approach is a novel framework for working with unconstrained, interpretable bases to solve a variety of classic problems using an $O(1)$ one-shot process for each new input. Our approach is most akin to a *behaviorist’s* approach. We observe the actions taken to accomplish a task rather than interpreting the internal workings of the network.

Over the past four decades, a number of approaches to extract meaning from neural processing have been pursued. Early examples include (Ludermir 1970; Touretzky and Hinton 1985). As explained by (Montavon, Samek, and Müller 2018), the approaches broadly fall into two categories: (1) interpreting the concepts learned - often involving examining activations within the network and (2) explaining the network’s decisions by techniques such as sensitivity analysis on the input variables. Many interpretability studies attempt to explain trained networks using visualizations, ranging from examining the gradients with respect to individual layers to reconstructions of inputs that maximize selected network activations (Selvaraju et al. 2017; Yosinski et al. 2015; Mahendran and Vedaldi 2016). Adversarial perturbations aid in visualizing the strengths and weaknesses of a network (Goodfellow, Shlens, and Szegedy 2014; Samangouei, Kabkab, and Chellappa 2018; Tramèr et al. 2017; Dong et al. 2017).

To achieve interpretability, an alternative to analyzing trained networks is to create models that lend themselves to understandable computations. This has been studied within deep learning (Afchar and Hennequin 2020; Zhang, Nian Wu, and Zhu 2018; Li et al. 2017) and other ML-models (Grosse et al. 2012; Lloyd et al. 2014). Our work similarly imposes structure, but allows us to find, without search, solutions composed from externally defined, unconstrained bases.

In addition to the tasks presented here, interpretability techniques have been used in numerous domains, including character recognition and inverse kinematics (Lake, Salakhutdinov, and Tenenbaum 2015; Andreas et al. 2015; Jacobs and Jordan 1993; Devin et al. 2017; Oyama et al. 2001). In real-world applications, the constraints on the bases are imposed from the outside: for example by the external software systems to control or by physical constraints, such as the articulation of robots. Domains in which it is important to adhere to these external constraints is where our system shines. For each of our tasks, we model the bases in forward-mode and achieve interpretable commands by controlling them.

2 Interpretable Control of Atomic Actions

An overview of our method is given in Figure 2, where the 5 high-level steps are listed. For clarity in exposition, we will concurrently describe the algorithm while tackling our first task: a non-search, non-optimization approach to function approximation. Given an arbitrary, potentially non-continuous, non-monotonic, one-dimensional function, such as in Figure 3(top), how do we approximate it as a sum of fixed bases, such as those shown in Figure 3(bottom)?

Step 1: Select the Bases/Actions and Train Generators

For this task, our external bases are defined as $B_{1..n}^{\lambda, \phi}(t)$,

Overview of 5 Step Procedure

1. Select the fundamental bases/actions relevant to the task and train the **Generator** models.
2. Set the number of generators and train a **Controller** network, via back-propagation through the Generator models, to direct the models.
3. Correct any residual error with repeated calls to the controller.
4. Extract the interpretable commands that control the original bases.
5. Fine tune the extracted commands.

Figure 2: High Level Description of Steps.

parameterized by the temporal scale, λ , and the time offset, ϕ , with a time index, t . The specific set of $n = 5$ bases are shown in Figure 3(bottom).¹ Notice the large impact of λ and ϕ ; a wide variety of shapes can be created by varying the two parameters. Given a predefined set of over-complete bases, there are numerous methods to find good basis parameters for arbitrary function approximation; usually they involve search and/or iterative optimization. In contrast, after training our system, every new function will be approximated using a one-shot, forward-propagation only (non-search, non-iterative) technique that produces interpretable results.

We train a set of n generator DNNs, $\{G_{1..n}^{\lambda, \phi}()\}$, to approximate the bases $\{B_i^{\lambda, \phi}(t)\}_{t=1..500}$ where $t = 1..500$. Each generator module is trained independently. Training examples for each generator are created by randomly selecting values for λ and ϕ and sampling the corresponding basis function at 500 points. At completion, each of five generator modules is specialized; it is proficient only at generating its specific external basis given as input any λ and ϕ . In Figure 3(bottom), the true external basis values are in blue and the generator outputs in orange. The generators are good, though not perfect, approximators².

Step 2: Set the Generators & Train a Controller

Once the generator modules are trained, their weights are frozen — they can no longer change. Given the fully trained generator modules, $G_{1..n}^{\lambda, \phi}$, a controller DNN is then trained to solve the actual approximation task. Succinctly, the goal of the controller is to emit the appropriate parameters λ , ϕ for each generator, such that when the generators’ outputs are combined, they reveal a good approximation to the target.

The full target function (Figure 3(top)) is used as input into the controller network. The controller outputs interpretable control commands, real values that are fed as input into each of the generators (λ , ϕ). Additionally, the controller emits two extra parameters (A , k) per generator: A and k are used to scale and bias the amplitudes of the generator-network

¹There is nothing special about this combination of bases. They were chosen because they are commonly found in literature. Anecdotally, as long as one basis creates step-like edges and another smooth transitions, other combinations worked equivalently.

²One monolithic generator could be used to model all the functions. However, separate generators are simpler and more modular.

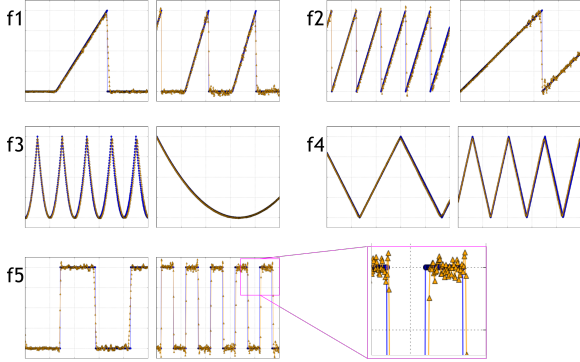
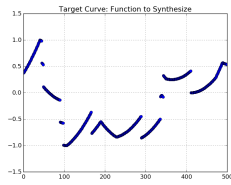


Figure 3: (top) A difficult target function to approximate; it is discontinuous and poorly behaved. We learn to approximate it with the small set of externally defined bases, f1-f5. (bottom) 5 basis functions that are non-monotonic and discontinuous. Two examples shown for each with random λ and ϕ settings, $t = (1..500)$. Blue: the actual external basis values ($1 \leq t \leq 500$). Orange: the generator's approximation. f5 zoomed: close, but not perfect, approximations.

outputs. The complete output is therefore: $\sum_{i=1..n} V_i$ where $V_i = A_i \times G_i^{\lambda_i, \phi_i}() + k_i$. See Figure 4.

This approach works because (1) when training the controller, during the forward pass, the generator modules constrain the controller to produce only interpretable control commands. (2) During back-prop, the generator modules are differentiable, *unlike the external bases B*, and pass error-gradients through them indicating how to change λ and ϕ even though the generator modules' weights do not change.

The controller network is trained to minimize the L_2 error between $\sum_{i=1..n} V_i$ and the input curve. The training examples for the controller are randomly generated functions employing arbitrary combinations of the 5 external bases, as well as a number of other randomly chosen trigonometric functions.

The training is done in Tensorflow; a full description of the training regime and architectures is given in the Appendix. As mentioned earlier, the only variation from standard training is that the weights of the generator modules (with blue backgrounds in Figure 4) *are not updated* when the weights of the controller network are trained.

Consider the similarities of this procedure and classic autoencoders (Jiang 1999; Kramer 1991; Theis et al. 2017; Larsen, Sønderby, and Winther 2015). Both are trained to minimize the differences between the reconstruction and inputs. In autoencoder networks, the decoder manipulates an internal representation of the inputs directly back into outputs

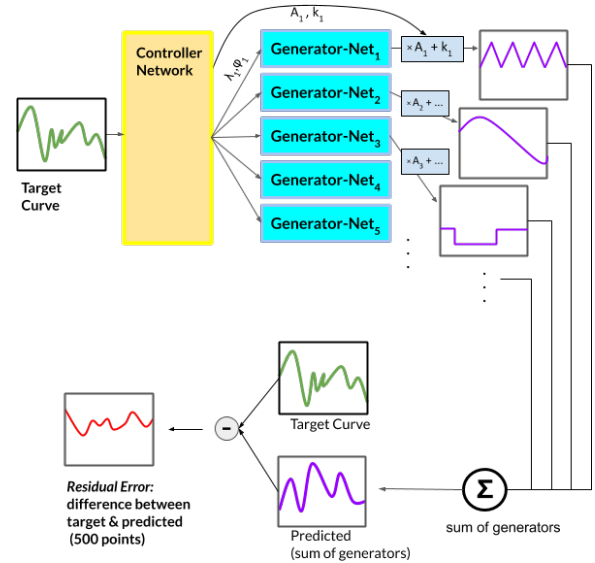


Figure 4: The controller outputs the parameters to control each of the *generator modules*. The outputs of the generator modules are summed and the L_2 error computed. Errors are propagated back through entire system – including through the frozen generator modules – to train the controller network.

that are in the same space as the inputs. Here, we require a level of indirection; the decoder must control generator modules through which the mapping back into the correct space occurs. An interesting related work is found in (Alet, Lozano-Pérez, and Kaelbling 2018), in which a similar application was addressed; however, their emphasis was on discovering a larger set of reusable generators rather than finding the parameters for pre-specified (*e.g.* externally given) bases.

We test the system on 1,000 randomly generated functions that were not used in training; in general all are poorly behaved, *e.g.* non-monotonic, non-periodic, and discontinuous. They include trigonometric functions other than the 5 bases. See Table 1-Line 1. Since 5 generator modules are used, there are 20 parameters that are output by the controller network. (More accurately, note that the amplitude biases ($k_{1..n}$) are additive and can be grouped into Δ , reducing the parameters to 16). The final error between the resultant waveform and the input is 4.85 ($\sqrt{L_2}$). This is the baseline.

Step 3: Correct the Residual Error

To improve the results, note that because the controller is trained to approximate any input function, we can feed the residual error back into the controller network and then approximate it as well, updating the combined reconstruction accordingly. The addition of the residual approximation reduces the error by 19% (Table 1-Line 2). Lines 3,4 continue this same process with subsequent residuals, yielding a 29% error reduction. A visualization of the benefits of residual correction are shown in Figure 5(first column).

To summarize the procedure to this point, see the equations below defined for n generators: $C(\mathbf{I})$ is the controller network applied to the input, $I(t)$, that is trained to produce

Experiment Description	#Gen Nets	Params (w/ Δ)	$\sqrt{L_2}$ error	% relative to base
1 baseline	5	20 (16)	4.85	—
2 + residual ₁	10	40 (31)	3.94	81.2%
3 + residual ₂	15	60 (46)	3.59	74.0%
4 + residual ₃	20	80 (61)	3.44	70.9%
5 w/extracted cmds	0	80 (61)	4.22	87.0%
6 + rescaled A, Δ	0	61	3.43	70.7%

Table 1: Function Approximation Results on the Test-Set

the generator controls, λ and ϕ , and the generator amplitudes scales and offsets, \mathbf{A} and \mathbf{k} . The iterative application of the controller network is achieved by applying it to the original target $\mathbf{I} = \mathbf{I}_0$ as the first pass and to the residual $\mathbf{I}_{t+1} = \mathbf{I} - \sum_{m=0}^t R_m$ for subsequent passes.

$$C(\mathbf{I}_t) \rightarrow \begin{bmatrix} \lambda_{t,1} & \cdots & \lambda_{t,n} \\ \phi_{t,1} & \cdots & \phi_{t,n} \\ A_{t,1} & \cdots & A_{t,n} \\ k_{t,1} & \cdots & k_{t,n} \end{bmatrix}$$

$$R_t = \sum_{i=1}^n A_{t,i} G_i^{\lambda_{t,i}, \phi_{t,i}}(\mathbf{t}) + k_{t,i}$$

Step 4: Extract the Interpretable Commands

The primary goal of this paper is to obtain understandable control commands. Thus far, we have used the controller network to “send commands” to each of the 5 generators. Within the system, the generator modules serve as differentiable proxies for the external bases. Assuming that the generator modules are good approximators, we can swap the proxies, $G_{1..n}$, with the actual bases, $B_{1..n}$. For the replacement, the controller’s outputs, $\lambda_i, \phi_i, A_i, k_i, i = 1..20$, initially used as the inputs to generator modules, are now the parameters to the externally defined bases. As before, the output amplitudes of $B_i^{\lambda_i, \phi_i}(t)$ are scaled and biased by A_i, k_i .

As shown in Table 1-Line 5, the error actually increases when using the controller network to specify $B_{1..n}$ directly instead of approximating them through $G_{1..n}$. See Figure 5(second column). Initially, this appears counter-intuitive as the original waveforms, and not their approximations, are used to create the input-functions that we are reconstructing. However, this occurs because G was not a perfect estimator of B . The controller network, which has only been trained with G , has learned to operate with the generators’ inaccuracies. When the external basis functions, B , are used, there is a mismatch in the training and operating regimes.

One solution is to train better generators. In this task, there are an infinite number of training samples, so this will work. However, in more complex applications, further training may not be feasible. A simpler approach is possible (Step 5).

Step 5: Fine Tune the Extracted Commands

Recall that in addition to the control parameters, the controller also emitted the amplitude-scale and -bias parameters, $A_{1..20}$ and $k_{1..20}$. Let us only recompute these scaling values and the combined bias parameter $\Delta = \sum_{i=1}^{20} k_i$. Modifying

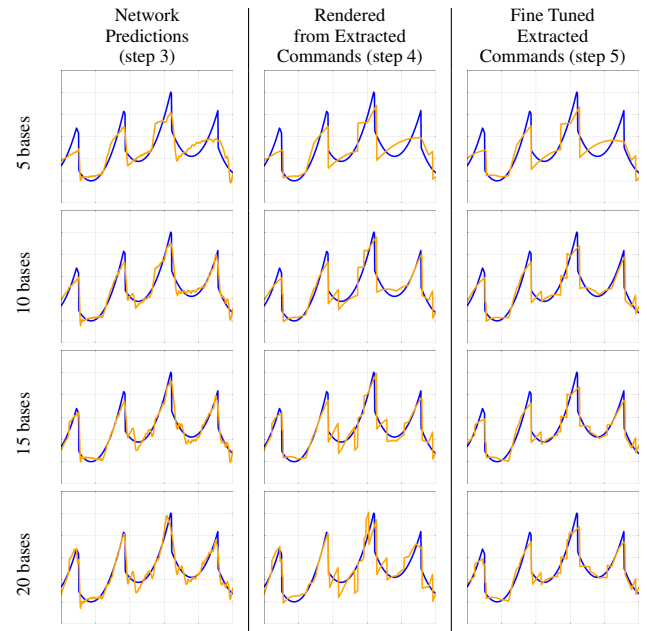


Figure 5: Visualizing the residual corrections. Top (5-bases). Column 1: the result (orange) of using 5 bases for fitting the target (blue) — this is the result of steps 1-3. Column 2: the fit of the actual extracted commands (step 4). Column 3: the fit after the fine tuning (step 5). Subsequent rows show effect of adding bases via residual corrections.

these parameters does not reshape the curves or intrinsically change the external bases. The least-squares-error (LSE) solution to this reweighting can be solved algebraically. After reweighting, the loss incurred by removing the network approximators is recovered. See Table 1-Line 6, and Figure 5.

A visualization of the results, *based solely on the extracted commands fed into the external bases*, is shown in Figure 6. Despite the ill-behaved, non-differentiable nature of both the target functions and the externally defined bases, good approximations are found without search or fitting, through a single forward pass of the network. Most importantly, the set of external bases is fully specified, extractable and understandable. We do not need to trust that the network has approximated the function well; we can see it by using the extracted commands directly on the external bases.

It is interesting to consider these processes in the context of basis *selection* and importance weighting. The non-zero A weights indicate which bases are most important. Further, with the controller’s ability to set A to 0.0 or by moving A to 0.0 during fine tuning, a basis can also be effectively *deselected* such that it does not contribute to the final solution.

Non-Approximation (non-DNN) Based Methods

Though we do not expect this one-shot, single forward-pass, neural network approximation approach to perform as well as an iterative search based method, we briefly examine one here for background. Recall that our approximations must not only determine the offsets for the curves, \mathbf{A} and \mathbf{k} , but also its shape, λ and ϕ . A more traditional method is to

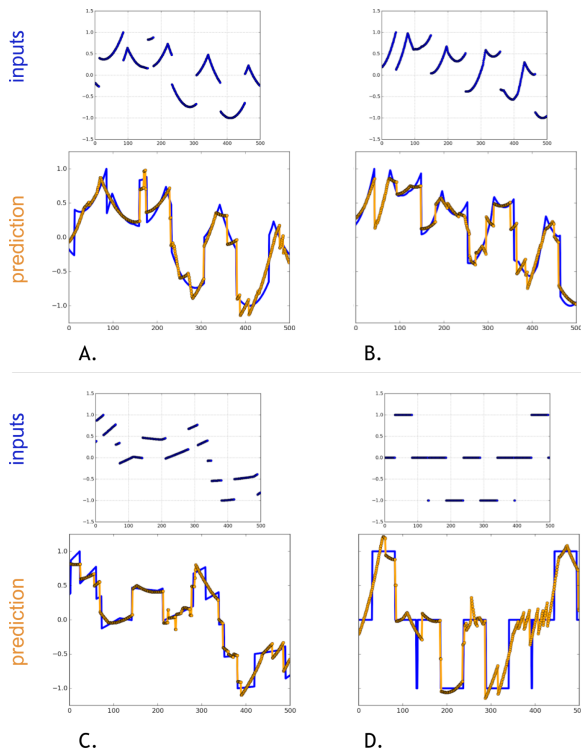


Figure 6: In each pair, top: 500 point input to the controller network. Bottom: the approximation (orange) and the target (blue). No generator modules are used; these are all created based on the *extracted commands*. Note the severe discontinuities and non-linearities throughout. (D) an example of poor performance from a noisy approximation.

use orthogonal matching pursuit (OMP) (Mallate and Zhang 1993). OMP uses greedy selection of bases and readjusts all of the weighting coefficients after each step. This requires that we generate every shape of the curves to be considered. We sample the 5 basis families 2000 times each by varying λ and ϕ , yielding a 10000-element dictionary to search.

Note that, unlike OMP, our method’s computation is *independent of the size* of the equivalent OMP dictionary. OMP computation grows linearly with the size of the dictionary. That search is possible (although slow) for this task but, in the next two tasks (Sections 3 and 4), the analogous OMP dictionary would be too prohibitively large to use.

In our approach, the controller effectively subdivides the dictionary and selects 5 entries at a time, in parallel. Modifying OMP to operate under those same constraints gives us $\sqrt{L_2}$ errors of 5.9 (after the first round of 5 selections), 4.0 (after the second round), 3.2 (3rd round), and 2.8 (4th round).³ It is interesting to note that the performance of our approach is better than the greedy search of OMP for the first half of the reconstruction process. A possible explanation is that our controller net has learned to differentially match

³Better OMP results are possible by pure sequential search (by removing the simultaneous basis selection and the basis-family constraints). However, this further increases OMP’s computation.

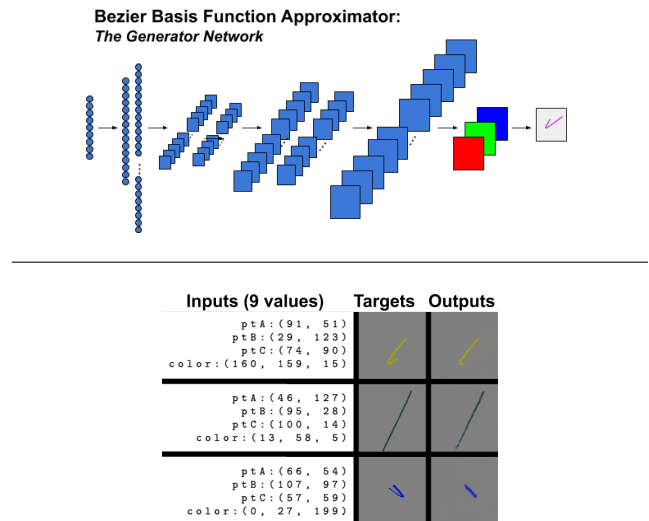


Figure 7: Top: Generator DNN to model Bezier curves given 3 points and a color onto a 128×128 canvas. After the FC-layers, the image is successively upscaled from 32×32 to 128×128 . It takes in 9 inputs (3 x,y coordinates and 3 RGB values). Bottom: examples of training samples. 9 input values and the target and actual output images are shown.

the available generator shapes to the inputs. In the next two sections, where the search spaces are much larger and less constrained, the brute-force search of OMP is not possible.

3 Images from Individual Crayon Strokes

Perhaps one of the most creative uses of deep neural networks in the past few years has been in artistic style transfer (Gatys, Ecker, and Bethge 2015; Johnson, Alahi, and Fei-Fei 2016; Kotovenko et al. 2019; Isola et al. 2017; Zhang, Zhang, and Cai 2020). We present a novel approach to non-photo-realistic rendering of images. Imagine that instead of transferring the style of an artist, we wish to describe each *individual action* an artist made to create the image.

Our Style Transfer task recreates paintings and photographs with crayon strokes. In addition to the end result, it yields the specifics of each of the $10^3 - 10^5$ crayon strokes that can be fed into any commercial external rendering engine. Recently, several studies have also tackled crayon-like drawing (Das et al. 2020; Ha and Eck 2017) but have developed neural-programmatic descriptions of sketch-like figures rather than the recreation of complex images. Next, we follow the same 5 steps from the previous section to tackle this task.

Step 1. The selection of the basis function is simple: an individual crayon stroke is the sole externally defined basis function and the final result is composed only of these. We describe each stroke by a Bezier curve specified with 3 control points (Barnhill and Riesenfeld 2014). Each stroke is considered a single pen-down, move, pen-up, atomic action. Therefore, we require only a single generator module — one which takes in the parameters of the stroke and outputs an image of the stroke. See Figure 7. The generator has 9 inputs: 3 (x,y) coordinates (the control points) and 3 RGB values

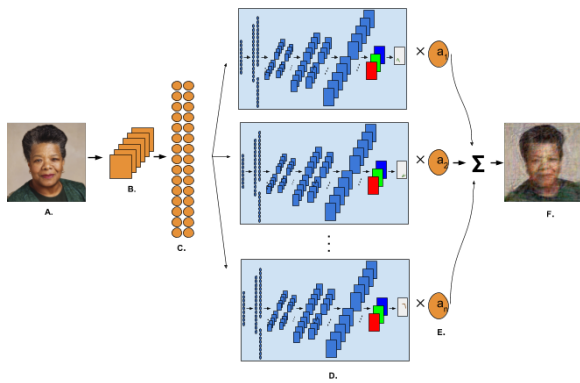


Figure 8: Full architecture. A. The input image. B&C. 2D convolution + FC layers. The controller emits 9 values and an alpha value to each generator. (D) The generators emit a canvas that is multiplied by alpha (E) and the summed result is shown in F. The networks in light blue *do not* change their weights in this stage. Nonetheless, loss-gradients from the reconstruction are passed through to the controller.

for color. The output is the stroke displayed on a 128×128 gray-background. Training the generator module in forward-mode is simple: we just generated training examples using integer-valued locations for the Bezier control points.

Recall that in the previous task, the controller network directed a set of 5 basis functions 1-4 times (depending on the number of residuals). For this task, the controller network will direct a single generator $10^3 - 10^5$ times to specify each stroke required to create the entire image.

Step 2. The controller network reconstructs the input image by producing commands instructing the generator to place crayon strokes on an initially empty canvas. The reconstructed image, I' is specified as $I' = \sum_{i=1}^n G_i(C_i) \cdot A_i$ for $n > 0$ generators, where $\{C_i\}$ and $\{A_i\}$ are the n sets of control commands and scalars, respectively, output from the controller. Each generated canvas, $G_i(C_i)$, is multiplied by a scalar weighting, an alpha value A_i , from the controller, then summed to produce the final image.

Following the same procedures as described earlier, the controller takes as input only the target – in this case the original image. It simultaneously sends commands to multiple copies of the generator module as well as the alpha value (similar to A, k in the previous section) to each of the generators. See Figure 8. We experimented with the number of generators simultaneously controlled by the controller (10, 50, 100, 200). 100 and 200 yielded pleasing results; 100 is used here. The 100 canvases are combined in precisely the same manner as the 1-D functions in the previous section.

The loss, \mathcal{L} , propagates through the generator modules, G_i , to the controller network and is based directly on the pixels of all of the training examples: $\mathcal{L} = \sum_{j=1}^{N_T} \|I_j - I'_j\|^2$ where $\{I_j\}$ are the N_T training images. As before, the generator module’s weights do not change during controller training. Training details are given in the Appendix.

Step 3. The system can also be improved in the similar manner: by passing back the residual error to be approx-

imated again by the controller. To be consistent with the previous experiments, it was passed back 4 times, and 400 strokes were combined to yield I' .

Step 4. Most importantly, the control commands are extractable, interpretable instructions for constructing the image. The controller outputs 10 parameters for each generator: the curve control points and colors C , and the alpha value A . Once extracted, these parameters can be used to draw Bezier curves in any commercial drawing system.

We take this opportunity to illustrate how to create higher resolution images than the initial 128 pixels the system is trained upon. In the first step, we reduce the target image to 128×128 pixels and run the system to obtain I' . Next, the target image is resized to 256×256 and the 128×128 I' up-sampled to the same. The residual for the entire image is computed. Each 128×128 residual sub-image (stride 64) is, in turn, sent to the controller and the set of 400 strokes collected from each. Together, these are composited to create I'' of size 256×256 . Larger images can be created similarly. As before, the commands are extracted and rendered in an external renderer (Clark and Contributors 2021).

Step 5. Finally, to rectify the differences between the network outputs and the actual rendering, only the alpha values of I'' are fine-tuned after the strokes are completely specified (using the exact same procedure described in Section 2).

Results. Results are shown in Figure 9. Several points should be noted about the results. First, the rendered images appear similar to the originals, while having the characteristic strokes of a crayon upon closer inspection. Second, we explicitly tested dark straight lines against a bright background to see if the network learns to “color within the lines” and whether it can render sharp boundaries (see first zoomed inset, Figure 9). It appears to be able to do so. Third, remember that the defining parameters to every stroke are maintained. Each stroke can be recovered and individually visualized. Lastly, note that these images are *not approximations* from the neural system, they are entirely created from the *extracted* commands to create Bezier curves.

In the process of training the controller, we found an unanticipated source of error. Not only did the controller adapt to the training inaccuracies of the generators (as described previously), it learned to exploit holes in training. The controller network used the non-integer bits in the control-point-location and color parameters to the generator to exploit parts of the generator’s operations that diverged from the intended rendering process. Only when the generator networks were replaced with an external rendering engine were the inaccuracies uncovered as the exploitable gaps were no longer present in the external renderer.

The controller used this exploit in *every* trial conducted. To prevent this exploit, the outputs of the controller were forced to only one of d (128 for coordinates, 256 for colors) values between $[-1.0, 1.0]$ during the forward pass (training and testing). In training, the backward-pass used a straight-through estimator (Hinton, Srivastava, and Swersky 2012). With this change, the only avenue the controller had to reduce the error was through the correct manipulation of the generators. Therefore, when the generators were replaced with an external renderer, the system worked as designed.



Figure 9: Eight images reproduced with the crayon-like strokes specified by the controller network, extracted from the system, and rendered entirely in an external graphics package. Three zoomed in regions shown to visualize the individual crayon strokes. Additional results in Figure 1.

4 Image Composing with Basic Shapes

For the final task, we show how an Inverse Renderer can easily fit into our framework.

4.1 Context and Background Work

This task has a wealth of related work. The most closely applies learning-based approaches to inverse-graphics, which can be found in early computer vision as well as recent studies (Baumgart 1974; Loper and Black 2014; Wu, Tenenbaum, and Kohli 2017; Romaszko et al. 2017; Patow and Pueyo

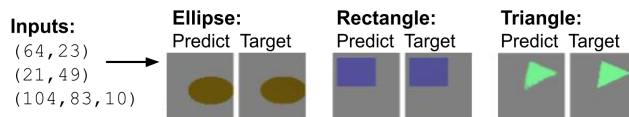


Figure 10: Basis shape generators for third task: shape-based image composition. Inputs shown for the ellipse generator.

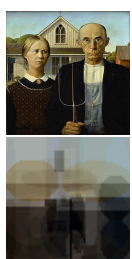
2003; Tian et al. 2019). Most commonly, the problem is tackled by first creating a probabilistic model of the types of images to be generated with latent parameters and then creating an inference algorithm to find their most likely setting for the parameters (Mansinghka et al. 2013; Kulkarni et al. 2014; Jampani et al. 2015). The ongoing work in using visual entities as a representation of images (Hinton et al. 2012; Hinton, Krizhevsky, and Wang 2011; Sabour, Frosst, and Hinton 2017; Jaiswal et al. 2018) is also closely related. (Tieleman 2014) has nice extensions using domain-specific decoders to approximate rendering functions. Recently, Spatial Transformers Networks (Jaderberg et al. 2015; Lin et al. 2018; Lin and Lucey 2017) provide a method to incorporate image-warpings within a deep network, which can be a powerful tool for both image registration and image generation. Explicitly handling occlusions with basic shapes has also been tackled using a deep learning approach in the above studies and also in constructive solid geometry (Sharma et al. 2019). As we are working only with 2D general shapes, occlusions are handled implicitly in our system by the order in which the shapes are drawn and the alpha channel.

Exciting new directions towards making inverse-rendering interpretable have been proposed. Representative approaches are briefly described here. The first utilized DNNs for program induction, specifically outputting programs that can render sketches and for inverse-CAD (Ellis et al. 2018, 2019). This was extended by (Liu and Wu 2019; Tian et al. 2019); the latter in which programs are developed via back-propagation to inverse-procedural graphics of 3-d voxel representations – such as those in ShapeNet. Other work forced interpretable outputs by requiring the commands to be fed into a graphics engine (Wu, Tenenbaum, and Kohli 2017; Ganin et al. 2018). This presented a very important distinct path of research – the fundamental difficulty of a non-differentiable actions in a rendering engine, which we tackled with creating differentiable approximate models of the actions, was instead handled by using a reinforcement learning scheme around the renderer.

We do not attempt to supplant this rendering work; this section demonstrates the ease of adapting to a new domain.

4.2 Following the Steps

Step 1. As before, we first select the external bases to control and train the generators. Based purely on aesthetic/simplicity considerations, we chose the most common geometric shapes, see Figure 10. In other experiments (not reported here), a variety of other, more complex, shapes have been used with no modifications to the procedure. The rectangle and ellipse bases have 7 inputs specifying the color and bounding box of the shape: $(x_1, y_1), (x_2, y_2), (r, g, b)$. The triangle basis genera-



shape	bounding box	color
rect	30 6 35 16	245 253 252
rect	0 0 63 23	204 218 217
rect	0 32 19 35	246 246 240
rect	36 16 43 28	131 104 73
rect	38 0 63 31	130 151 151
ellipse	26 27 51 49	238 244 244
ellipse	8 48 58 63	2 7 22
ellipse	10 33 27 52	115 114 110
ellipse	48 20 63 43	7 6 0
ellipse	0 12 27 33	140 160 200

Figure 11: Deconstruction of *American Gothic*, Grant Wood (1930). First 50 shapes shown and 10 extracted commands.

tor has 9 inputs, employing an additional (x_3, y_3) for the third vertex. The generator modules employ the same inputs as the external bases; they map the coordinate and color inputs to a filled colored shape on a 64×64 canvas. These generator modules train quickly; details are in the Appendix.

Step 2. After training the generator modules, their weights are frozen and the controller network is trained. The controller network emits the bounding box for each shape (or vertices for triangles) and alpha channel parameters, $\{A_i\}$, as done in the previous section. The most impactful parameter that we determine empirically is how many generator modules to simultaneously control by the controller; we tried 10, 50, 100 & 500 shape generators (uniformly distributed among the shapes). The generators’ output canvases are summed to recreate the input image, and the L_2 difference computed. As expected, the more shapes that are used, the lower the error.

The controller training used 64×64 ImageNet (Deng et al. 2009) photos as inputs and targets. Testing was conducted on a held-out set of 1000 ImageNet photographs. In these experiments, 500 total shape generators were used. Ablative testing, including with fewer generators, as well as complete training details, are given in the Appendix.

Step 3. Next, the same extensions as in the previous tasks are applied. First, we feed back the residual error to the controller. To make the system more tractable, instead of initially generating 500 canvases simultaneously, we generate 50 canvases, composite the results, and compute the residual. Exactly as before, the residual is passed through the controller network and another image generated. The resulting images are summed and the process repeated. This error rectification is iterated 10 times, yielding a total of 500 shapes from 10 iterations of the controller (9 residuals).

Step 4. Command extraction is trivially accomplished by examining the outputs of the controller (e.g. Figure 11) as they fully specify the shape to draw, its color and its weight. With this information, the command can be given, without any modification, directly to any external renderer.

Step 5. To further improve the results, as before, simply recompute $\{A_i\}$, the N multiplicative weighting scalars (e.g. the alpha channel). Recall that this can be solved algebraically. Though not meaningful by itself, we note that on the objective metric of L_2 error, the fine-tuning dropped the error by 5.3% from 15.1 to 14.3. Final samples are shown in Figure 12. The reconstructions, done with only ellipses and rectangles, reveal high fidelity to the originals. The harder examples

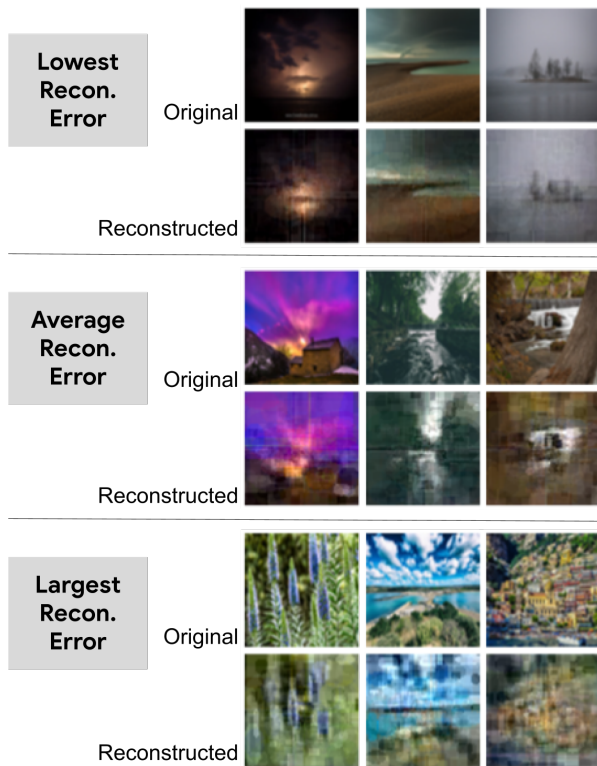


Figure 12: Reconstructions with 500 shapes, $\frac{1}{2}$ ellipses and $\frac{1}{2}$ rectangles on test-set images. Reconstructions with a variety of performances shown. These are created by extracting the commands and alpha parameters from our system and sending them to an external rendering engine.

(bottom) occur when the target image contains many edges or high-frequency features. More results are in the Appendix.

5 Discussion

There are two primary contributions of this paper. First, we presented a method to train networks to solve decomposable tasks so that the actions/transformations made by the network are extractable and interpretable combinations of externally defined operations/bases. This was shown in three distinct tasks. The second contribution is in the Style Transfer domain. Unlike the majority of work in neurally transforming images, our process did not operate as a monolithic black box. Instead, each stroke can be extracted and its contribution to the result visualized and understood.

Importantly, across all applications, we have found that the decomposition of the end tasks into units that fit this approach is natural – the *generator modules*, though potentially numerous, are simple both in concept and to train. The controller network emits directions to the generators in a straightforward manner that can then be extracted easily from the neural system and analyzed and used externally.

A primary objective of this work was to allow a user to fully specify the set of basis functions that comprise the final solution. For example, in the two latter tasks, the bases were chosen entirely based on aesthetic considerations. We have

successfully used this approach with a variety of different bases sets, such as irregular shapes, space-filling with glyphs from various fonts, varied hatching marks, etc. Our goal was to demonstrate the ability to select arbitrary shapes/bases as dictated by the needs of specific applications.

An interesting extension to this work would be to replace the L_2 error, used here in training, with a Generative Adversarial Network framework. Another promising direction is in finding a parsimonious set of bases to solve a task. Currently, we specify how many generators (and residual iterations) are used to create the output. By introducing a parsimony factor, we can dynamically reduce the number of bases used. Two successful methods of accomplishing this are by augmenting the controller’s loss during training to include a penalty for using the full set of allowed generators. A post-training method can be applied as well. During the final LSE-rescaling fine-tuning, rather than solely minimizing the L_2 reconstruction error, a regularization penalty proportional to the magnitude of the rescaled weight/alpha-channel can be applied. This has been successfully utilized in all three of the tasks described here. Extending this work to reveal more compact sets of results is currently underway.

Appendix

Function Approximation The generator networks (Figure 13) were trained in parallel on a 56 processor Intel Xeon E5-2690 V4 (non-gpu) machine. Networks with 5-7 hidden layers were tried; they worked similarly. However, significantly reducing the hidden units per layer hurt performance. One generator is trained for each basis function. The controller net was trained on a P100 GPU for 3+ days depending on the number of generators used. Learning rate = 10^{-4} .

Crayon Strokes / Style Transfer In this task, the generator network maps the input of 3 control points (x,y coordinates) of a Bezier curve to a full RGB 128x128 pixel canvas containing only the rendered curve. The network achieves acceptable results in a few hours on a computer with 56 Intel Xeon E5-2690 V4 processors. We trained it for 24 hours to improve results. The output is a 128x128 RGB image where each pixel is a float with 256 discretizations. The controller network was trained on P100 GPU for 1-3 days. To avoid confusion, note that after training, the same controller is used, unchanged, across all images. A controller is *not* trained per image.

Inverse Rendering Earlier, we provided results on a standard set of photographs. To measure robustness, we applied the system to logos, paintings, clip-art, and bw-photographs – all outside the types of images the system was trained on. Table 2 presents results on this harder set and ablative results. The previous experiments are marked with “photo”.

A generator is trained for each basis shape. For example, the rectangle generator takes as input $(x_1, y_1)(x_2, y_2)(r, g, b)$ and outputs a filled rectangle specified by the bounding box in the specified color. The same network as used with the crayon generators could be used here. However, since we are only estimating 64x64 images instead of 128x128, smaller architectures also work. The controller network, was trained on a P100 GPU. Acceptable performance was achieved in 2 days, but was allowed to train for 1 week.

<p>Generator Network for Function Approximation: input $(\lambda, \phi [from\ controller]) \rightarrow$ $(5,6\ or\ 7) \times$ fully-connected-layers (1000 units, relu activation)\rightarrow — optional: include shortcuts from inputs fully-connected-layer (500 units, no activation)\rightarrow scaled and biased by $(A, k [from\ controller])$</p>
<p>Controller Network for Function Approximation: input (500 sampled points from the target function) \rightarrow $10 \times$ fully-connected-layers (1000 units, tanh activation)\rightarrow output $(\lambda, \phi, A, k) \times$ number-of-generators\rightarrow — (λ, ϕ) are used as direct inputs into the generator network to — specify the shape of the curve. — (A, k) are used as a scale and bias of the generator’s outputs</p>
<p>Generator Network for a Single Bezier Stroke: input $((x_1, y_1)(x_2, y_2)(x_3, y_3)(r, g, b)) \rightarrow$ $2 \times$ fully-conn-layer (50 units, relu)\rightarrowfully-conn-layer (70 units, relu)\rightarrow fully-connected-layer ($16 \times 16 \times 128$ units, relu)\rightarrow reshape-to (16,16,128)\rightarrow $2 \times \{$depth-to-space(2) \rightarrow conv2d(channels=32, kernel=2, stride=1, relu)$\}$$\rightarrow$ depth-to-space(2)\rightarrow conv2d(channels=3, kernel=2, stride=1, discretized-tanh₂₅₆) — this is a 128×128 pixel image in 3 channels.</p>
<p>Controller Network for Crayon Rendering input $(I, \text{the image to reproduce, } (128 \times 128 \times 3)) \rightarrow$ $3 \times$ Conv2d(channels=200, kernel=2, stride=1, relu)\rightarrow $3 \times$ Conv2d(channels=100, kernel=1, stride=1, relu)\rightarrow flatten$\rightarrow 2 \times$ fully-connected-layer (300 units, relu)\rightarrow fully-connected-layer (10 units * n generators, tanh)\rightarrow — 1:6 $(x_1, y_1)(x_2, y_2)(x_3, y_3)$ are discretized to 128 values — 7:9 (r,g,b) colors are discretized to 256 values — 10) A: alpha channel remains a real-value float Optional: After each residual pass, rescale new target image to [-1.0,1.0]</p>
<p>Controller Network for Image Composing input $(I, \text{the image to reproduce, } (64 \times 64 \times 3)) \rightarrow$ $2 \times$ Conv2d(200, kernel=2, stride=1, relu)\rightarrow Conv2d(100, kernel=2, stride=1, relu)\rightarrow Conv2d(50, kernel=1, stride=1, relu)\rightarrow Conv2d(25, kernel=1, stride=1, relu)\rightarrowflatten\rightarrow $2 \times$ fully-connected-layer (200 units, relu)\rightarrow fully-connected-layer (8 units * n generators, tanh)</p>

Figure 13: Details of all network architectures.

test set	Residual used?	Generators per controller	Pixel/Channel Error	LSE Re-weight?	% Improve
hard	no	10	41.3	no	—
hard	no	50	33.3	no	19.4%
hard	no	100	31.2	no	24.4%
hard	no	500	29.0	no	30.0%
hard	yes: 9	50	23.8	no	42.4%
hard	yes: 9	50	19.6	yes	52.5%
photo	yes: 9	50	15.1	no	n/a
photo	yes: 9	50	14.3	yes	n/a

Table 2: Regressive Summary for the Image Composition task (% improvement is measured relative to first row).

References

- Afchar, D.; and Hennequin, R. 2020. Making Neural Networks Interpretable with Attribution: Application to Implicit Signals Prediction. arXiv preprint arXiv:2008.11406.
- Alet, F.; Lozano-Pérez, T.; and Kaelbling, L. P. 2018. Modular meta-learning. arXiv preprint arXiv:1806.10166.
- Andreas, J.; Rohrbach, M.; Darrell, T.; and Klein, D. 2015. Deep Compositional Question Answering with Neural Module Networks. *CoRR* abs / 1511.02799. URL <http://arxiv.org/abs/1511.02799>.
- Barnhill, R. E.; and Riesenfeld, R. F. 2014. *Computer aided geometric design*. Academic Press.
- Baumgart, B. G. 1974. Geometric modeling for computer vision. Technical report, Stanford, Computer Science.
- Chakraborty, S.; Tomsett, R.; Raghavendra, R.; Harborne, D.; et al. 2017. Interpretability of deep learning models: a survey of results. In *2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing*, 1–6. IEEE.
- Clark, A.; and Contributors. 2021. Pillow 8.12, Python Imaging Library (Fork). <https://pypi.org/project/Pillow/>. URL <https://pypi.org/project/Pillow/>.
- Das, A.; Yang, Y.; Hospedales, T.; Xiang, T.; and Song, Y.-Z. 2020. BezierSketch: A generative model for scalable vector sketches. arXiv preprint arXiv:2007.02190.
- Deng, J.; Dong, W.; Socher, R.; Li, L.-J.; Li, K.; and Fei-Fei, L. 2009. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, 248–255. IEEE.
- Devin, C.; Gupta, A.; Darrell, T.; Abbeel, P.; and Levine, S. 2017. Learning modular neural network policies for multi-task and multi-robot transfer. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, 2169–2176.
- Dong, Y.; Su, H.; Zhu, J.; and Bao, F. 2017. Towards interpretable deep neural networks by leveraging adversarial examples. arXiv preprint arXiv:1708.05493.
- Ellis, K.; Nye, M. I.; Pu, Y.; Sosa, F.; Tenenbaum, J.; and Lezama, A. S. . 2019. Write, Execute, Assess: Program Synthesis with a REPL. *CoRR* abs / 1906.04604. URL <http://arxiv.org/abs/1906.04604>.
- Ellis, K.; Ritchie, D.; Solar-Lezama, A.; and Tenenbaum, J. 2018. Learning to infer graphics programs from hand-drawn images. In *Advances in Neural Information Processing Systems*, 6059–6068.
- Fan, F.; Xiong, J.; and Wang, G. 2020. On interpretability of artificial neural networks. arXiv preprint arXiv:2001.02522.
- Ganin, Y.; Kulkarni, T.; Babushkin, I.; Eslami, S. A.; and Vinyals, O. 2018. Synthesizing Programs for Images using Reinforced Adversarial Learning. *CoRR* abs / 1804.01118. URL <http://arxiv.org/abs/1804.01118>.
- Gatys, L. A.; Ecker, A. S.; and Bethge, M. 2015. A Neural Algorithm of Artistic Style. *CoRR* abs/1508.06576. URL <http://arxiv.org/abs/1508.06576>.
- Goodfellow, I. J.; Shlens, J.; and Szegedy, C. 2014. Explaining and harnessing adversarial examples. arXiv preprint arXiv:1412.6572.
- Grosse, R.; Salakhutdinov, R. R.; Freeman, W. T.; and Tenenbaum, J. B. 2012. Exploiting compositionality to explore a large space of model structures. arXiv preprint arXiv:1210.4856.
- Ha, D.; and Eck, D. 2017. A Neural Representation of Sketch Drawings. *CoRR* abs / 1704.03477. URL <http://arxiv.org/abs/1704.03477>.
- Harnad, S. 1990. The symbol grounding problem. *Physica D: Nonlinear Phenomena* 42(1-3): 335–346.
- Hinton, G.; Krizhevsky, A.; Jaitly, N.; Tieleman, T.; and Tang, Y. 2012. Does the brain do inverse graphics? In *Brain and Cognitive Sciences Fall Colloquium*, volume 2.
- Hinton, G.; Srivastava, N.; and Swersky, K. 2012. Neural networks for machine learning. *Coursera, video lectures* 264.
- Hinton, G. E.; Krizhevsky, A.; and Wang, S. D. 2011. Transforming auto-encoders. In *International Conference on Artificial Neural Networks*, 44–51. Springer.
- Isola, P.; Zhu, J.-Y.; Zhou, T.; and Efros, A. A. 2017. Image-to-image translation with conditional adversarial networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 1125–1134.
- Jacobs, R. A.; and Jordan, M. I. 1993. Learning piecewise control strategies in a modular neural network architecture. *IEEE Transactions on Systems, Man, and Cybernetics* 23(2): 337–345.
- Jaderberg, M.; Simonyan, K.; Zisserman, A.; et al. 2015. Spatial transformer networks. In *Advances in neural information processing systems, 2017–2025*.
- Jaiswal, A.; AbdAlmageed, W.; Wu, Y.; and Natarajan, P. 2018. CapsuleGAN: Generative Adversarial Capsule Network. In *The European Conference on Computer Vision (ECCV) Workshops*.
- Jampani, V.; Nowozin, S.; Loper, M.; and Gehler, P. V. 2015. The informed sampler: A discriminative approach to bayesian inference in generative computer vision models. *Computer Vision and Image Understanding* 136: 32–44.
- Jiang, J. 1999. Image compression with neural networks—a survey. *Signal Processing: Image Communication* 14(9): 737–760.
- Johnson, J.; Alahi, A.; and Fei-Fei, L. 2016. Perceptual losses for real-time style transfer and super-resolution. In *European Conference on Computer Vision*.
- Kotovenko, D.; Sanakoyeu, A.; Lang, S.; and Ommer, B. 2019. Content and Style Disentanglement for Artistic Style Transfer. In *The IEEE International Conference on Computer Vision (ICCV)*.
- Kramer, M. A. 1991. Nonlinear principal component analysis using autoassociative neural networks. *AICHE journal* 37(2): 233–243.

- Kulkarni, T. D.; Mansinghka, V. K.; Kohli, P.; and Tenenbaum, J. B. 2014. Inverse graphics with probabilistic cad models. *arXiv preprint arXiv:1407.1339*.
- Lake, B. M.; Salakhutdinov, R.; and Tenenbaum, J. B. 2015. Human-level concept learning through probabilistic program induction. *Science* 350(6266): 1332–1338.
- Larsen, A. B. L.; Sønderby, S. K.; and Winther, O. 2015. Autoencoding beyond pixels using a learned similarity metric. *arXiv preprint arXiv:1512.09300*.
- Li, O.; Liu, H.; Chen, C.; and Rudin, C. 2017. Deep learning for case-based reasoning through prototypes: A neural network that explains its predictions. *arXiv preprint arXiv:1710.04806*.
- Lin, C.-H.; and Lucey, S. 2017. Inverse compositional spatial transformer networks. In *IEEE-CVPR*, 2568–2576.
- Lin, C.-H.; Yumer, E.; Wang, O.; Shechtman, E.; and Lucey, S. 2018. ST-GAN: Spatial Transformer Generative Adversarial Networks for Image Compositing. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.
- Liu, Y.; and Wu, Z. 2019. Learning to describe scenes with programs. In *International Conference on Learning Representations*.
- Lloyd, J. R.; Duvenaud, D.; Grosse, R.; Tenenbaum, J. B.; and Ghahramani, Z. 2014. Automatic construction and natural-language description of nonparametric regression models. *arXiv preprint arXiv:1402.4304*.
- Loper, M. M.; and Black, M. J. 2014. OpenDR: An approximate differentiable renderer. In *European Conference on Computer Vision*, 154–169. Springer.
- Ludermir, T. B. 1970. Extracting Rules From Neural Networks: A Data Mining Approach. *WIT Transactions on Information and Communication Technologies* 22.
- Mahendran, A.; and Vedaldi, A. 2016. Visualizing deep convolutional neural networks using natural pre-images. *International Journal of Computer Vision* 120(3): 233–255.
- Mallate, S.; and Zhang, Z. 1993. Matching pursuit with time-frequency dictionaries. *IEEE Transactions on Signal Processing* 41(12).
- Mansinghka, V. K.; Kulkarni, T. D.; Perov, Y. N.; and Tenenbaum, J. 2013. Approximate bayesian image interpretation using generative probabilistic graphics programs. In *Advances in Neural Information Processing Systems*, 1520–1528.
- Montavon, G.; Samek, W.; and Müller, K.-R. 2018. Methods for interpreting and understanding deep neural networks. *Digital Signal Processing* 73: 1–15.
- Oyama, E.; Agah, A.; MacDorman, K. F.; Maeda, T.; and Tachi, S. 2001. A modular neural network architecture for inverse kinematics model learning. *Neurocomputing* 38: 797–805.
- Patow, G.; and Pueyo, X. 2003. A survey of inverse rendering problems. In *Computer graphics forum*, volume 22, 663–687. Wiley Online Library.
- Romaszko, L.; Williams, C. K.; Moreno, P.; and Kohli, P. 2017. Vision-as-inverse-graphics: Obtaining a rich 3d explanation of a scene from a single image. In *IEEE-CVPR*, 851–859.
- Sabour, S.; Frosst, N.; and Hinton, G. E. 2017. Dynamic Routing Between Capsules. In Guyon, I.; Luxburg, U. V.; Bengio, S.; Wallach, H.; Fergus, R.; Vishwanathan, S.; and Garnett, R., eds., *Advances in Neural Information Processing Systems 30*, 3856–3866. Curran Associates, Inc. URL <http://papers.nips.cc/paper/6975-dynamic-routing-between-capsules.pdf>.
- Samangouei, P.; Kabkab, M.; and Chellappa, R. 2018. Defense-gan: Protecting classifiers against adversarial attacks using generative models. *arXiv preprint arXiv:1805.06605*.
- Selvaraju, R. R.; Cogswell, M.; Das, A.; Vedantam, R.; Parikh, D.; and Batra, D. 2017. Grad-CAM: Visual Explanations From Deep Networks via Gradient-Based Localization. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*.
- Sharma, G.; Goyal, R.; Liu, D.; Kalogerakis, E.; and Maji, S. 2019. Neural Shape Parsers for Constructive Solid Geometry. *arXiv preprint arXiv:1912.11393*.
- Smolensky, P. 1986. Neural and conceptual interpretations of parallel distributed processing models. Technical report, Colorado Univ at Boulder.
- Theis, L.; Shi, W.; Cunningham, A.; and Huszár, F. 2017. Lossy Image Compression with Compressive Autoencoders. In *International Conference on Learning Representations*. URL <https://openreview.net/pdf?id=rJiNwv9gg>.
- Tian, Y.; Luo, A.; Sun, X.; Ellis, K.; Freeman, W. T.; Tenenbaum, J. B.; and Wu, J. 2019. Learning to Infer and Execute 3D Shape Programs. *CoRR* abs/1901.02875. URL <http://arxiv.org/abs/1901.02875>.
- Tieleman, T. 2014. *Optimizing neural networks that generate images*. University of Toronto (Canada).
- Touretzky, D. S.; and Hinton, G. E. 1985. Symbols among the neurons: Details of a connectionist inference architecture. In *IJCAI*, volume 85, 238–243.
- Tramèr, F.; Kurakin, A.; Papernot, N.; Goodfellow, I.; Boneh, D.; and McDaniel, P. 2017. Ensemble adversarial training: Attacks and defenses. *arXiv preprint arXiv:1705.07204*.
- Wu, J.; Tenenbaum, J. B.; and Kohli, P. 2017. Neural scene de-rendering. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 699–707.
- Yosinski, J.; Clune, J.; Nguyen, A.; Fuchs, T.; and Lipson, H. 2015. Understanding neural networks through deep visualization. *arXiv preprint arXiv:1506.06579*.
- Zhang, Q.; Nian Wu, Y.; and Zhu, S.-C. 2018. Interpretable convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 8827–8836.
- Zhang, Y.; Zhang, Y.; and Cai, W. 2020. A Unified Framework for Generalizable Style Transfer: Style and Content Separation. *IEEE Transactions on Image Processing* 29: 4085–4098.