Learning to Draw Vector Graphics: Applying Generative Modeling to Font Glyphs

by

Kimberli Zhong

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2018

Author .....
Department of Electrical Engineering and Computer Science
May 25, 2018

Certified by .....
Frédo Durand
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Christopher J. Terman
Chair, Master of Engineering Thesis Committee

Learning to Draw Vector Graphics: Applying Generative Modeling to Font Glyphs

by

Kimberli Zhong

Submitted to the Department of Electrical Engineering and Computer Science
on May 25, 2018, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Today, designers work in tandem with computerized tools to create stylized graphic designs, diagrams, and icons. In this work, we explore the applications of generative modeling to the design of vectorized drawings, with a focus on font glyphs. We establish a data-driven approach for creating preliminary graphics upon which designers can iterate. To accomplish this, we present an end-to-end pipeline for a supervised training system on Scalable Vector Graphics (SVGs) that learns to reconstruct training data and produce similar but novel examples. We demonstrate its results on selected characters using a Google Fonts dataset of 2552 font faces. Our approach uses a variational autoencoder to learn sequences of SVG drawing commands and is capable of both recreating ground truth inputs and generating unseen, editable SVG outputs. To investigate improvements to model performance, we perform two experiments: one on the effects of various SVG feature encodings on generated outputs, and one on a modified architecture that explictly encodes style and class separately for multi-class generation.

Thesis Supervisor: Frédo Durand
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

TODO: acknowledgements

Contents

List of Figures

List of Tables

TODO: run spell check

Chapter 1

Introduction

The computerization of graphic design has had wide ranging effects on the design process, from introducing software for arranging visual and text layouts to providing platforms for distributing and sharing creations. As design tools such as bitmap and vector image editing programs improve, the design process becomes less complex and tedious, leaving designers room to focus on the high-level creation process. In this work, we explore the applications of generative modeling to the design of vector graphics and establish a data-driven approach for creating preliminary vector drawings upon which designers can iterate. We present an end-to-end pipeline for a supervised training system on Scalable Vector Graphics (SVGs) that learns to reconstruct training data and generate novel examples, and we demonstrate its results on font glyphs.

Our motivation for examining the generation of designs is two-fold. One, we see practical purpose in a recommendation tool that augments a designer's experience, and we believe such a tool would be a valuable addition to a designer's creative process. Two, we hope learning to algorithmically mimic the process by which glyphs are created will offer insight into the intent and structure behind human-created designs.

Although much work has been done in understanding and synthesizing rasterized images and designs, primarily with deterministic computer vision algorithms and convolutional neural networks, we focus our investigation on the domain of vectorized

images in this work. The two representations are quite different, and we aim to both produce generated designs with fewer aesthetically displeasing artifacts as well as investigate what new information about designs' underlying shapes and structures can be quantified and learned with the vectorized data format. Importantly, our method produces ready-to-edit SVG designs whose component lines and curves can be easily adjusted and rearranged.

In the next chapter, we provide further background on the domain and discuss related work. Then, in the following chapters, we delve into the methods used to train our vector graphics generator on the data processing side as well as the model architecture side. We then demonstrate our methods as applied to font glyph generation, using a selected set of characters. Finally, we describe two experiments, one for comparing feature encodings, and one for explicitly encoding style in the model's latent space.

Chapter 2

Background

The problem space we explore ties together work across a number of different disciplines, including graphics, graphic design, and machine learning modeling. In applying generative methods to vectorized glyphs, we are inspired by previous work in computer vision on non-natural images and draw upon recent developments in the generative modeling of line drawings.

2.1 Non-natural images

While natural images are photographs of real-world scenes and objects, non-natural images are computationally generated, either by hand with a computer design tool or automatically. Images in this category include graphs, pictograms, virtual scenes, graphic designs, and more. Algorithmically understanding, summarizing, and synthesizing these images pose unique challenges because of the images' clean and deliberately drawn designs, amalgamation of distinct visual and textual elements, and purposeful or narrative nature.

While much attention has been focused on research problems like object recognition, scene segmentation, and classification on natural images, interest in applying computer vision methods to non-natural images has been growing. Much progress has been made towards computationally understanding non-natural images on datasets including XML-encoded abstract scenes [1], comic strips [2], and textbook diagrams [3].

Recent work by our group has explored the space of infographics, complex diagrams composed of visual and textual elements that deliver a message [4].

2.1.1 Font faces

Within the space of non-natural images, we focus specifically on designer-crafted font faces. Fonts are used to typeset text with particular styles and weights, and many types of fonts exist, including serif, sans serif, and handwriting style fonts (see Figure 2-1). Each font is defined by a set of glyphs, which include letters, digits, symbols, and potentially other Unicode characters. Within a font face, glyphs generally share the same style for properties like angles, curvature, presence of serifs, and stroke width.

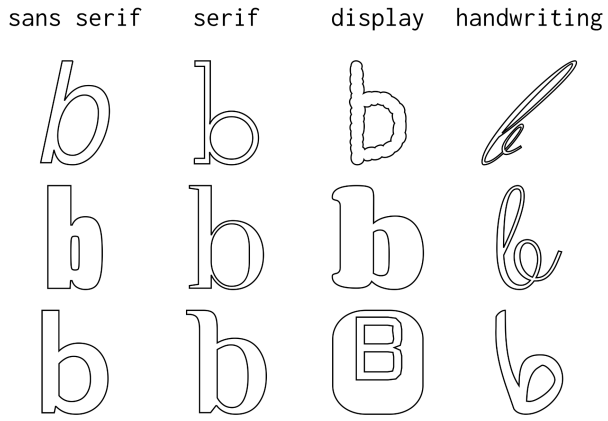


Figure 2-1: Examples of glyphs from each font type. Serif glyphs have *serifs*, or slight extensions off the ends of strokes, while sans serif glyphs do not.

In the context of computational modeling and generation, font glyphs offer certain advantages and pose distinctive challenges when compared to other types of designed graphics. Unlike more complicated designs such as infographics and diagrams, they can be represented as drawings composed of simple lines and curves and, as such, can be modeled using sequential drawing commands. However, font glyphs have distinct classes (i.e. each letter, number, symbol, etc. is a different class) and thus designing a generation procedure that is able to create clearly defined instances of different classes presents a challenge, as generation has to operate under strict constraints.

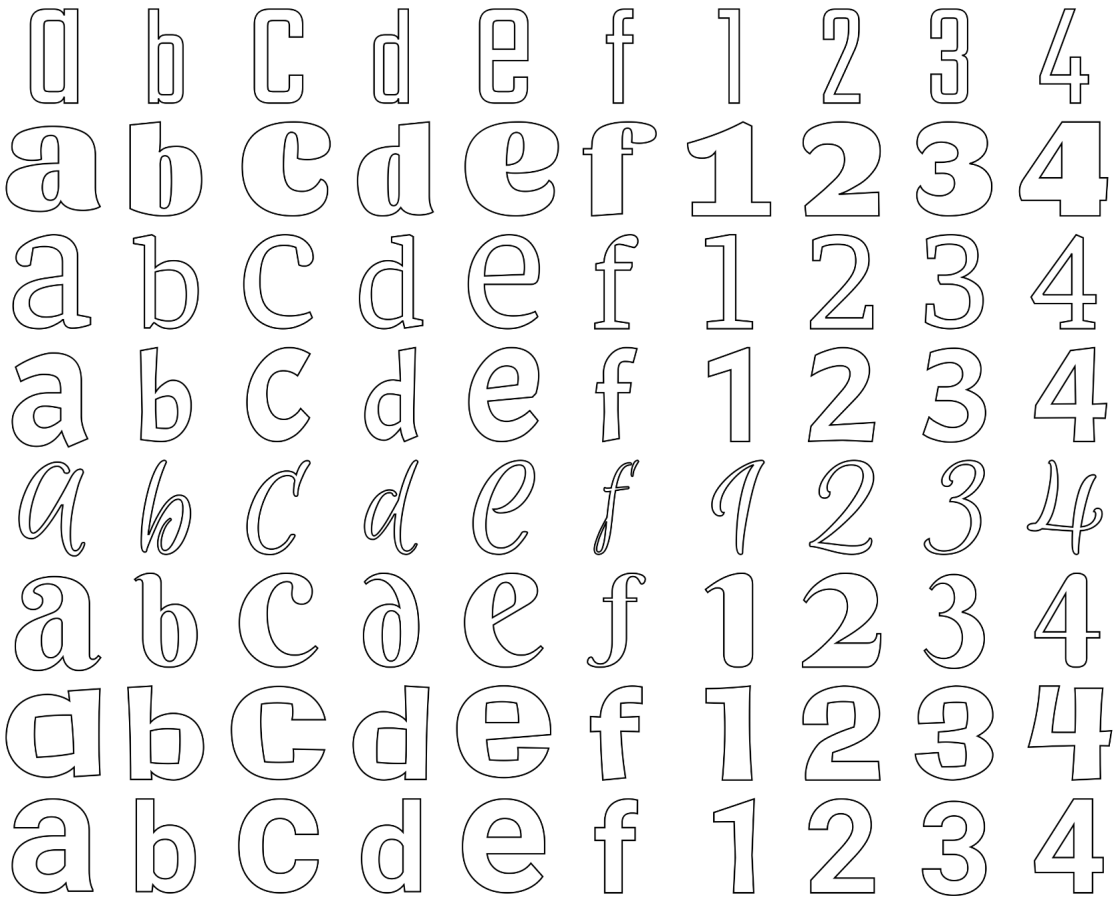


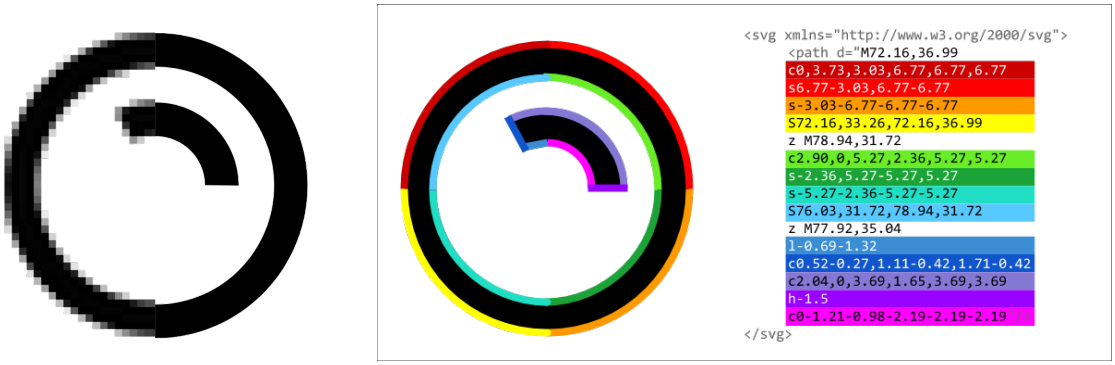
Figure 2-2: A sample of the types of font glyphs used in our dataset. Lowercase letters "a" through "f" are shown, as well as digits 1 through 4. Note the variety of styles represented: the dataset includes serif, sans serif, display, and handwriting style fonts.

In this work, our computational methods are applied to font faces downloaded from Google Fonts[1]. In Figure 2-2, a sampling of glyphs from the Google Fonts dataset is shown.

2.1.2 Vector graphics

We are primarily interested in applying computational models to vector graphics as opposed to raster images. While raster images encode color values for each point (or *pixel*) in a two-dimensional grid, vector graphics describe a set of curves and shapes

[1]Downloaded from `https://github.com/google/fonts`, a GitHub repository containing all fonts in the dataset.



(a) Raster images are defined as two-dimensional arrays of pixel values, while vector graphics define curves and paths mathematically. When scaled, vector graphics (right half) can still be rendered smoothly while raster images (left half) degrade in quality.

(b) SVG is an XML-based markup language that describes geometric elements within a vector image. A single path is used to draw this soap bubble, and colored curves in the image correspond approximately to the highlighted attribute commands that describe them. For example, the command `l-0.69-1.32` indicates a line drawn from a starting point to a point 0.69 units to the left and 1.32 units down.

Figure 2-3: A visual comparison of raster and vector graphics, and a sample SVG path. Image source: *Dog Wash* by Llisole from the Noun Project.

parameterized by mathematical equations and control points. Many specifications exist for describing images in vector format, including SVG, Encapsulated PostScript (EPS), and Adobe PDF. Font faces are often distributed as TrueType (TTF) files, which encode line and curve commands as well as metadata to aid with rendering.

Although the focus of our research is on font glyph inputs, our vision is to build a generalizable system for the variety of vector graphics generated by designers, including icons and logos. Thus, our system accepts SVG data as input, as most vector graphics formats (including TTF) can be expressed using the commands available in the SVG specification.

Modeling vector graphics

Applying computer vision techniques to vector graphics raises new challenges. While bitmap and vector data can both decode into the same visual output, their underlying encoding structures have vast differences (Figure 2-3a). As a data format, SVGs

describe lists of geometric objects (among other elements such as text which are outside the scope of this work), including lines, circles, polygons, and splines. The SVG `path` element, in particular, can be used to create all other element shapes. Its attributes describe a series of commands that move from point to point, drawing curves or lines between them, as shown in Figure 2-3b.

The early popular deep convolutional network architectures designed to classify natural images, such as in [5] and [6], were designed to take in length $M \times N$ input vectors whose values directly represent corresponding pixel values in size $M \times N$ pixel input images. While this model works well for raster images, this fixed-dimension format is incompatible with SVG element lists because their paths can describe any number of different point locations. Instead, SVGs as sequential, variable-length, structured text are better suited to representation in models such as recurrent neural nets (RNNs). RNNs are designed to model temporal sequences by unraveling across timesteps; long short-term memory (LSTM) models in particular use gated units to optimize for learning longer term dependencies [7].

2.2 Generative modeling

To solve the problem of creating novel vector drawings, we look towards the tools provided by generative machine learning methods. While discriminative modeling techniques focus on separating and identifying inputs to produce output labels learned from high-dimensional data such as images, generative algorithms create previously unseen instances of a class based on representative inputs. They are trained in an unsupervised or semi-supervised manner to learn a data distribution P, estimating the true data distribution P_{gt} from which samples are drawn. By drawing probabilistically from P, they can then be used to synthesize novel, unseen examples similar to input data.

Popular neural network-based approaches include generative-adversarial networks (GANs) [8] and variational autoencoders (VAEs). Introduced in 2014 [9], GANs pit a generative model $G(z; \theta_g)$ against an adversary $D(x; \theta_d)$ that learns to discriminate

between samples from the ground truth dataset and the generative model's latent space. When both G and D model differentiable functions, backpropagation can be used to train them towards convergence in a computationally efficient manner.

2.2.1 Variational autoencoders

Variational autoencoders, introduced in [10], learn an encoder function mapping training examples from the input space $\mathcal{X}$ to vectors z in the latent space $\mathcal{Z}$, as well as a decoder function that takes z vectors sampled from a probability distribution $P(z)$ and applies a function that produces a random vector in the space $\mathcal{X}$ [11]. Intuitively, the goal is to train the model to produce outputs that look similar to the x inputs but can be generated with some random noise such that they are distinct from training inputs. To accomplish this goal, training a VAE tunes parameters θ to maximize the likelihood of reconstructing training examples after passing them through the entire pipeline, since increasing the probability of recreating x inputs also increases the probability of creating similar random outputs. Formally, we aim to maximize

$$P(x) = \int P(x|\theta; z)P(z)dz \tag{2.1}$$

Often, $P(x|\theta; z) = \mathcal{N}(x|f(\theta; z), \sigma^2 * I)$ where σ is a parameter that can be set to manipulate the divergence of the generated output from training examples. In training, we constrain $P(z) = \mathcal{N}(0, 1)$ to simplify the loss calculation.

Transforming this objective function to be differentiable and thus trainable using stochastic gradient descent requires a key insight: instead of sampling many z_i then averaging $P(x|z_i)$, we focus only on the z values that are likely to produce x and compute $P(x)$ from those. Our encoder then learns $Q(z|x; \phi)$ that approximates $P(z|x)$, while the decoder learns $P(x|z; \theta)$. We can then define a loss function that describes a variational lower bound, where Kullback-Leibler (KL) divergence $\mathcal{D}$ accounts for the similarity between our $Q(z|x; \phi)$ and the true $P(z)$ and the reconstruction loss accounts for similarity between input and output:

$$\mathcal{L}_i = -E_{z\sim Q(z|x_i;\phi)}[\log P(x_i|z;\theta)] + \mathcal{D}(Q(z|x_i;\phi)||P(z)) \tag{2.2}$$

In [12], the reconstruction loss function is expanded to account for sequential inputs and outputs, combining log loss for each item in the sequence. This adjusted reconstruction loss can be used in a recurrent VAE architecture, where encoders and decoders digest sequential data.

2.3 Related work

Our end-to-end SVG generation model is inspired by prior work in line drawing generation, as both domains share an underlying temporal data structure. Furthermore, our application to font generation is preceded by a history of computational approaches to font parameterization and style classification.

2.3.1 Generating drawings

Unconditionally generating parameterized curves extends the established problem of polyline generation. Thus, we look towards contributions in handwriting and sketch generation, such as Graves's RNN-based handwriting prediction and synthesis work [12]. DRAW, a system introduced in [13], uses a pair of recurrent networks in an VAE architecture to model attention in MNIST character generation. Recent work by Ganin *et al.* uses a reinforcement learning approach to train an agent to draw sketches [14].

Polyline results can easily be vectorized to produce splines, as in [15] or [16]. However, our approach aims to model the entire SVG input to directly produce ready-to-edit spline output. In our work, we build upon the variational autoencoder method presented by Ha and Eck in [17]. We use a similar bidirectional sequence-to-sequence VAE, with an overall loss calculation that includes drawing location losses, pen state losses, and KL loss.

2.3.2 Font style

Knuth's Metafont format demonstrates pioneering work in font style parameterization and has since motivated high-level font classification systems with font faces parameterized by human-controlled features like curvature and stroke width [18][19][20].

Outside of manual feature selection approaches, many existing methods for modeling font style use font glyph raster images. Tenenbaum and Freeman present a method for modeling style and content separately and apply it to font style extrapolation [21]. In [22], polyline outlines are used to match glyphs across fonts using an energy optimization process, resulting in a learned manifold of fonts from which novel styles can be sampled. Approaches for learning stylized calligraphy, such as [23], take a more procedural approach, where strokes within characters are first extracted using shape segmentation approaches before use in training. Neural network and VAE techniques to learning font style are increasingly common, such as in [24] and [25], and Lian *et al.* use a combination of stroke segmentation and feature learning to generate handwriting fonts for Chinese characters [26].

Chapter 3

SVG feature representation

Our goal is to extend beyond polyline modeling and capture the higher-level shapes of SVG objects. Thus, one major challenge is to choose an adequate representation that captures all drawing information from an SVG and can be fed as an input vector to our neural network architecture. Although many different elements are allowed by the SVG specification, including text, images, and basic shapes, we simplify the problem space to focus only on the generic `path` element since paths can be used to compose other elements.

3.1 Overview of SVG commands

There are five commands possible in an SVG `path` as seen in Figure 3-1. Further detail about command parameters can be found in Table 3.1 [27].

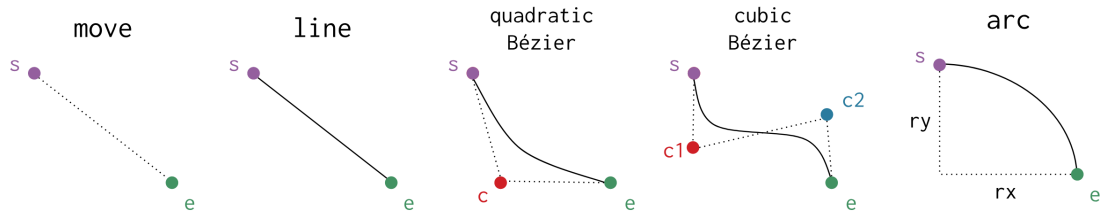


Figure 3-1: A visualization of the five commands in the SVG `path`

Table 3.1: A description of possible SVG path commands. For simplicity, we omit the relative coordinate variants of the commands, which specify (dx, dy) coordinates instead of absolute (x, y) for all control points. We also omit commands for vertical (`V`) and horizontal (`H`) lines as well as shorthand smoothed quadratic (`T`) and cubic (`S`) Béziers.

Command	Code & Description
Move	`M x y` moves the pen to a specified point (x, y)
Line	`L x y` draws a line from the current (start) point to the end point (x, y)
Quadratic Bézier	`Q cx cy, x y` draws a quadratic Bézier curve according to given control point (cx, cy) from the current point to the end point (x, y) with the parametric equation $\mathbf{B}(t) = (1-t)^2\mathbf{s} + 2(1-t)t\mathbf{c} + t^2\mathbf{e}$
Cubic Bézier	`C cx1 cy1, cx2 cy2, x y` draws a cubic Bézier curve according to given control points (cx_1, cy_1) and (cx_2, cy_2) from the current point to the end point (x, y) with the parametric equation $\mathbf{B}(t) = (1-t)^3\mathbf{s} + 3(1-t)^2t\mathbf{c}_1 + 3(1-t)t^2\mathbf{c}_2 + t^3\mathbf{e}$
Arc	`A rx ry t fl fs x y` draws a section of an ellipse with the given r_x and r_y radii from the current point to the end point (x, y) over angle t, with large-arc f_l and sweep f_s flags

3.2 Modeling SVGs

We would like to model SVG inputs without loss of information about pen movements. In essence, since SVGs name ordered lists of paths and their drawing commands, we model them as a sequence of mathematical parameters for the pen drawing commands and add a command for representing the transition between paths. The sequential nature of this representation makes the generation task well-suited to a recurrent neural network architecture, as we cover in Chapter 4.

3.2.1 Preprocessing

SVG images often have additional properties like stroke and fill style. As we focus on path generation exclusively, our first step in transforming input SVGs to a feature

representation is to strip away those styles and focus only on the `path` elements of the input, often resulting in an image that depicts the outlines of the input shape—see Figure 2-2 for examples.

Often, designers do not craft SVGs by editing XML by hand but rather with interactive image editing software such as Adobe Illustrator[1] or Inkscape[2]. Thus, human-created SVGs often have varying path compositions, layers, and canvas sizes.

To constrain the modeling process, we first homogenize our dataset by preprocessing inputs, rescaling to the same overall canvas size (set to 256×256 pixels) and reordering paths in the drawing sequence so that paths with larger bounding boxes are drawn first.

There is also variability in the directions and starting positions of path drawing. Instead of controlling these factors in the preprocessing stage, our architecture is designed for bidirectionality, and all command sequences are interpreted such that the pen starts at coordinate $(0, 0)$ and moves to the first point in the input drawing.

3.2.2 Simplifying path commands

We aim to produce a system capable of modeling general SVGs, so inputs can contain any path command specified in Table 3.1. To avoid training bias and to constrain the problem space, we consolidate the different path commands into a single feature representation that encompasses all possible path pen movements.

Out of the five SVG path commands, three are parametic equations of differing degrees, so we can model these three (lines, quadratic Béziers, and cubic Béziers) using the parameter space for the highest degree cubic-order equation. An elliptical arc segment, on the other hand, cannot be perfectly transformed into a cubic Bézier. Arcs have five extra parameters used to describe them (x-radius, y-radius, angle of rotation, the large arc flag, and the sweep flag), so to compress their encoding we approximate them with the same parameter space as used for our parametric commands. We use the method described in Appendix **??** to approximate arc segments as cubic Béziers.

[1] `www.adobe.com/illustrator`
[2] `https://inkscape.org`

After all path drawing command types have been transformed to use the parameters needed for modeling cubic Bézier segments, we can represent each SVG command as a feature vector comprising those parameters and a three-dimensional one-hot pen state vector, similar to the feature representation used in [17].

Finally, for each `move` command and each disjoint path, we insert a feature vector that encodes the new end point and sets the pen state to note that the pen is up.

In all, our feature representation models each drawing command as a nine-dimensional vector (in contrast to the five-dimensional feature vector for polyline drawings in [17]). Six dimensions are used to model three x, y coordinate parameters of cubic Béziers, and three dimensions are reserved for the *pen down* (p_d), *pen up* (p_u), and *end drawing* (p_e) pen states. Each input SVG is transformed into a sequence of commands, which is in turn translated into a sequence of these feature vectors. In Chapter 6, we examine further details of this feature transformation process and how our representation modifications affect modeling performance.

Chapter 4

Model architecture

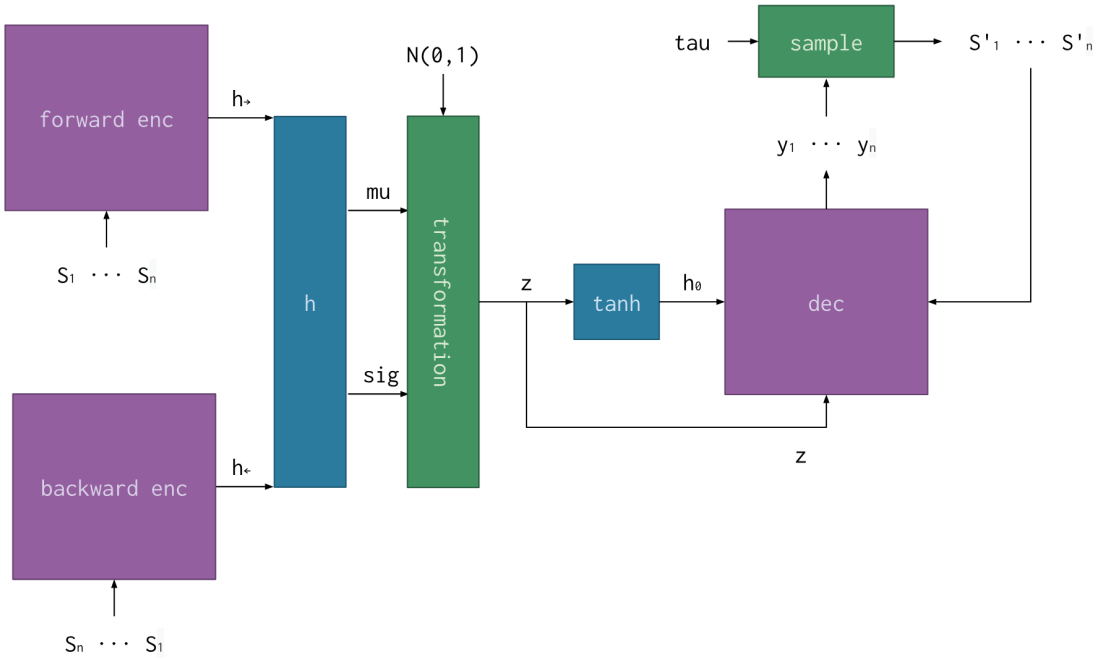
For our end-to-end SVG generation method, we build upon the work of Sketch-RNN Ha and Eck and use a similar bidirectional sequence-to-sequence variational autoencoder model [17]. We maintain a similar encoder and latent space architecture, but we modify the decoder to output parameters corresponding to four probability distributions instead of two. In Sketch-RNN, the decoder output parameterizes a Gaussian Mixture Model (GMM) for the pen coordinates as well as a categorical distribution for the pen state (*pen up*, *pen down*, or *end drawing*). To model general SVG commands, we modify the decoder to output parameters for three GMMs, one for each of the possible control points in a cubic Bézier curve (except the start position, since we assume we start drawing from the pen's current location), as well as for a categorical pen state distribution. An overview of the model architecture is depicted in Figure 4-1.

4.1 VAE modules

Dual RNNs are used in the encoder module, one for modeling forward and one for modeling backward sequences of feature vectors, with each feature vector representing a single SVG command. Both RNNs use LSTM cells with layer normalization, as introduced in [28]. After transforming an input SVG into a sequence of feature vectors $S = (S_1, S_2, \cdots, S_n)$ which includes padding with *end drawing* vectors to ensure S

is length n_{max}, each S_i is passed in to the encoder RNNs in the appropriate order. The final hidden states of both RNNs, $h_{\leftarrow}$ from the backward encoder and $h_{\rightarrow}$ from the forward encoder, are concatenated to form a combined output h. Assuming the latent space vector has dimension n_z, this output is then transformed into μ and σ vectors using a fully connected layer (and an exponentiation operation to produce a non-negative σ) such that both vectors have dimension n_z.

Figure 4-1: An overview of the basic SVG model architecture. Although similar overall to [17], the model is adapted such that y_i parameterizes three location GMMs and a pen state distribution.



The resulting μ and σ vectors are combined with a vector $\mathcal{N}$ of n_z normally distributed Gaussian samples from $\mathcal{N}(0, 1)$ to create a random latent vector z, with $z = \mu + \sigma \circ \mathcal{N}$.

A single-layer network takes in z and outputs the initial input h_0 to the decoder module. Each cell in the decoder takes in z, the output from the previous cell h_i, and the generated output SVG command S'_i. The output from a decoder cell y_i is a vector composed of three values for the categorical pen state distribution plus parameters for each of the three location GMM models.

The number of normal distributions in each location GMM is a tunable hyper-

parameter. By default, each GMM contains 20 normal distributions, and each distribution is parameterized by means and standard deviations for both the x and y locations and a correlation between x and y (μ_x, σ_x, μ_y, σ_y, ρ_{xy}). The models in the GMM also each have a mixture weight π. The values corresponding to σ values in y_i are again exponentiated to produce non-negative standard deviations, and we apply tanh operations to the values corresponding to ρ values to ensure they produce correlations between -1 and 1.

To generate S_i, each GMM is sampled to produce pen locations x and y for each coordinate in the SVG feature vector, and the pen state distribution is sampled to generate one of $\{p_d, p_u, p_e\}$, corresponding to *pen down*, *pen up*, and *end drawing*. A temperature multiplier (τ) is also used when sampling from GMMs and from the pen state distribution to adjust the randomness of the output, allowing for control over how expected (and thus how similar to existing inputs) the output is. Finally, all generated S_i are ordered in sequence to produce a generated list of SVG commands. The specific transformation from the three coordinates in the feature vector to the output SVG command parameters varies (see Chapter 6), but all feature encodings we use require three coordinates.

The model is also capable of generating unconditional output, in which no input SVG is supplied. The produced result is an unseen example not explicitly related to any particular image in the dataset. We train the model to produce unconditional output by setting the decoder's initial states to 0 and treating it as a standalone RNN without a z input.

Chapter 5

Application: font glyph generation

To demonstrate our end-to-end SVG generation method, we train the model architecture described in Chapter 4 on a dataset of font glyphs.

5.1 Dataset

Our dataset of font faces is downloaded from a GitHub repository containing all fonts available on Google Fonts (`https://github.com/google/fonts`). The dataset contains 2552 font faces total from 877 font families. Font faces exhibit much stylistic variation: a breakdown of font types for the 877 font families can be found in Table 5.1, and examples of font glyphs from each type can be found in Figure 2-1.

Table 5.1: The 2552 font faces in the Google Fonts dataset are from 877 total font families. Each font family can be classified as one of the following categories, with the "display" category encompassing bold fonts of a wide variety of styles that can be used as title text. Here, we report the counts of font families in the dataset belonging to each type.

Serif	Sans serif	Display	Handwriting	Monospace
180	249	296	135	17

All font faces are downloaded as TrueType (TTF) files then converted to SVG format using the FontForge command-line tool[1].

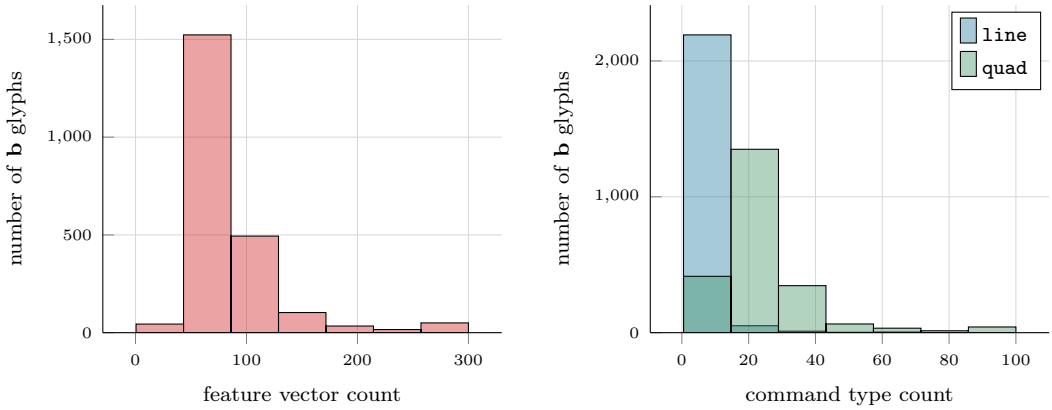
[1] `https://fontforge.github.io`

Input statistics highlighting the number and type of SVG commands within drawings for the glyph **b** across all font faces are shown in Figure 5-1. The remaining glyph statistics can be found in Appendix **??**.

Figure 5-1: Dataset statistics for the glyph **b** across all 2552 font faces in the Google Fonts dataset.

(a) A histogram depicting the number of feature vectors used to encode glyphs of the character **b** across our font faces. The glyph **b** has, on average, 91 feature vectors in its representation. Glyphs with more than 250 feature vectors were left out of our dataset when training.

(b) A breakdown of the occurrences of line and quadratic Bézier drawing commands per glyph. Note that TTFs support only these two types of drawing commands, although SVGs in general can contain any of the commands listed in Table 3.1.



5.2 Training

We train a separate instance of the model for each of the glyphs **b**, **f**, **g**, **o**, and **x**, chosen because they cover a variety of shapes and curves. Each input SVG is transformed to a list of feature vectors using the method described in Chapter 3. We use encoding B to translate SVG commands to feature fectors, details of which are described in Chapter 6. The datasets are then randomly partitioned into training, test, and validation sets; glyphs from the overall dataset of 2552 font faces are pruned if they consist of more than 250 feature vectors. Every glyph except **x** has a training set of 1920 SVGs, while **x** has a training set of 1960. All glyphs have test and validation sets of 240 SVGs each. Training is done with a batch size of 40, a learning

rate of 0.001, and a KL weight in the loss function that starts at 0.01 and increases asymptotically over time. The size of the latent space vector z is set to 128, and we use recurrent dropout to reduce overfitting. Loss graphs and details about trained models can be found in Appendix **??**.

5.3 Results

Here, we report quantitative results as well as a qualitative evaluation of model performance. In Figure 5-2, we highlight ground truth inputs and their conditionally decoded outputs for each of the letter glyph models.

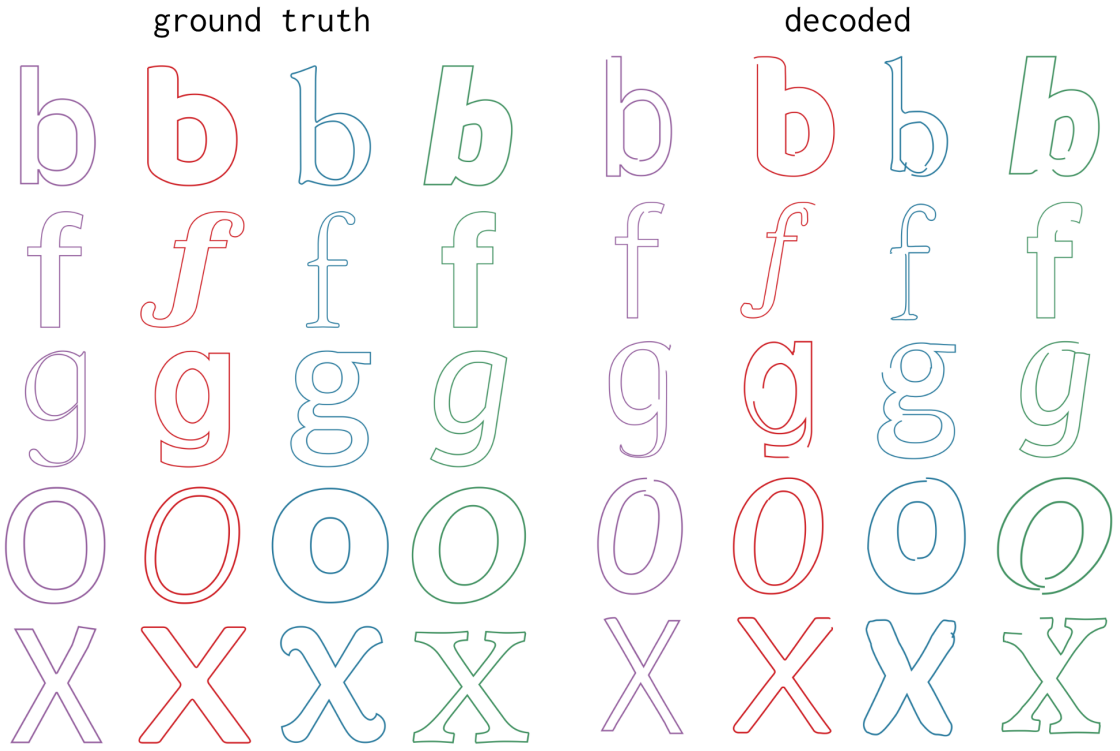


Figure 5-2: Selected glyphs conditionally generated by the trained single-class letter glyph model. Ground truth inputs on the left are fed into the encoder and decoded into the generated outputs on the right.

Table 5.2: Modified Hausdorff distance for models trained on each glyph on a test set of N images. We also provide two baselines: one comparison between input images and random noise with the same dimensions (to provide a reasonable upper bound), and one comparison between randomly sampled pairs of the same glyph class.

Glyph class	Mean	Std. dev.	Kurtosis	N pairs
Conditionally generated vs. ground truth				
b	19.5341	8.3484	1.9515	240
f	15.7533	9.2132	4.7359	240
g	19.7714	9.6469	6.8383	240
o	27.8857	11.9887	5.8081	240
x	28.8542	14.2474	1.9820	238
Random noise vs. ground truth				
b	180.9758	18.7591	27.0412	240
f	188.4756	40.3329	2.7637	240
g	185.9078	31.1026	3.7460	240
o	204.1991	21.1567	2.91859	240
x	188.7353	16.9588	0.1258	240
Random ground truth pairs				
b	23.2421	12.8087	2.8928	120
f	26.8287	13.0423	2.7861	120
g	24.6996	11.2764	6.9264	120
o	25.1075	13.5501	3.9498	120
x	21.8579	9.1399	1.0702	120

5.3.1 Quantitative results

To quantify model performance, we compute an image similarity metric between each ground truth image and a corresponding image conditionally generated by the model with $\tau = 0.3$, where temperature τ specifies randomness when sampling from the decoder as defined in [17]. Images are first converted to point clouds containing every pixel in the raster image with nonzero value. The resulting sets of points are translated to be mean-centered for each image, and the modified Hausdorff distance in the range $[0, \infty\}$ is calculated from the point set of each generated image to the point set of its corresponding ground truth image [29]. While we also considered using metrics such as pixel loss and feature extraction, we chose the Hausdorff distance for its simplicity as a measure of mutual polygonal proximity. Evaluation is run on a set

of N test images for each glyph, and quantitative results can be found in Table 5.2. We also provide baselines for the Hausdorff metric: one comparison between input images and random noise with the same dimensions, and one comparison between randomly sampled pairs of the same glyph class.

5.3.2 Qualitative results

We demonstrate the model's learned ability with a few illustrative examples.

In Figure 5-3, we interpolate between latent vectors for input **b** and **f** glyphs of different styles. If interpolated latent vectors tend to produce coherent outputs, it indicates that the KL regularization term in the loss function is sufficiently forcing the latent space to be used efficiently.

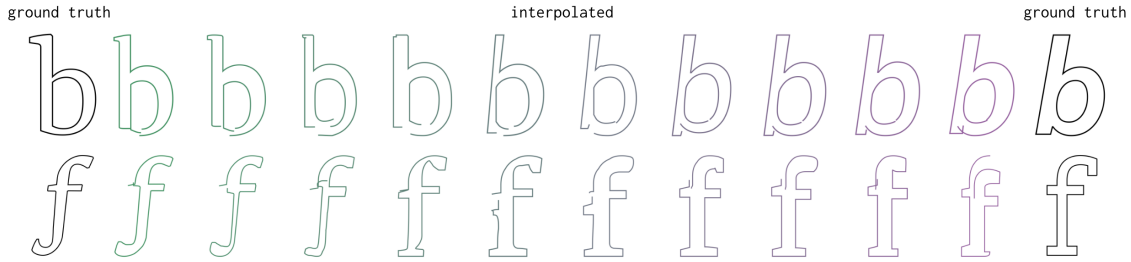


Figure 5-3: The latent space abstractly encodes style and must learn to identify different types of glyphs. The input glyphs are on the far sides of the figure, and we spherically interpolate between their latent vectors, decoding with $\tau = 0.1$.

In Figure 5-4, we demonstrate how temperature τ affects the decoding process as a randomness scaling factor in sampling from the decoder output GMMs. Intuitively, a lower temperature indicates that the decoder samples outputs that it believes are more likely.

Figure 5-5 depicts common failure modes during conditional generation. Because the model is drawing sequences of commands using pen displacement, it often struggles with closing the loop and returning to its original path start point. Additionally, since some font face styles are exceptionally rare, the model fails to learn how to represent non-standard font styles.

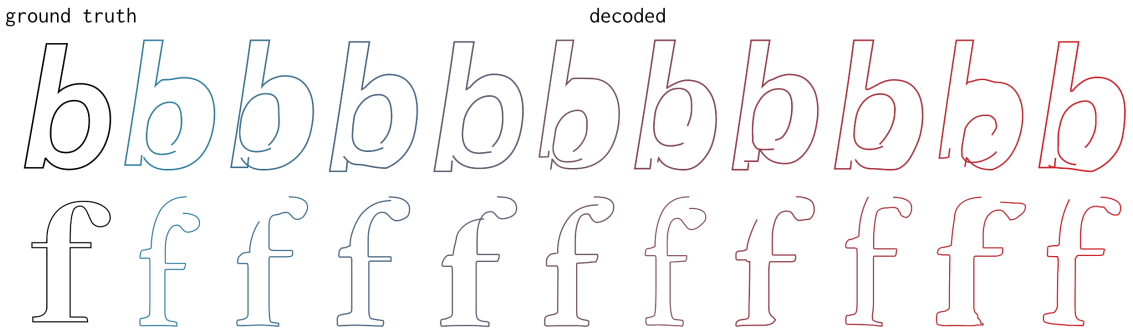


Figure 5-4: To demonstrate how temperature affects decoding, we decode the same latent vector at different temperature settings. At the far left, $\tau = 0.1$, and at the far right, $\tau = 1.0$, with τ increasing by 0.1 for every intermediate image. As temperature increases, model generates outputs less likely to match the input image.

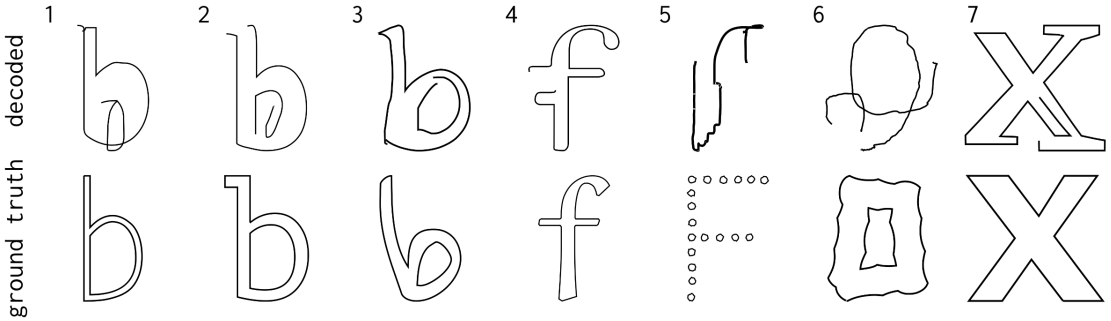


Figure 5-5: The model generally makes a set of common mistakes, as shown here. It struggles with positioning disconnected components (1), closing paths (2, 4), and understanding uncommon styles (3, 5, 6). It also sometimes confuses styles (7).

Finally, Figure 5-6 demonstrates the creative ability of the model: generating unseen examples of a variety of styles. Instead of encoding an input SVG and passing in the resulting latent vector into the decoder, we train the decoder separately to learn unconditional generation without a z input, with the decoder RNN initial states instead set to 0.

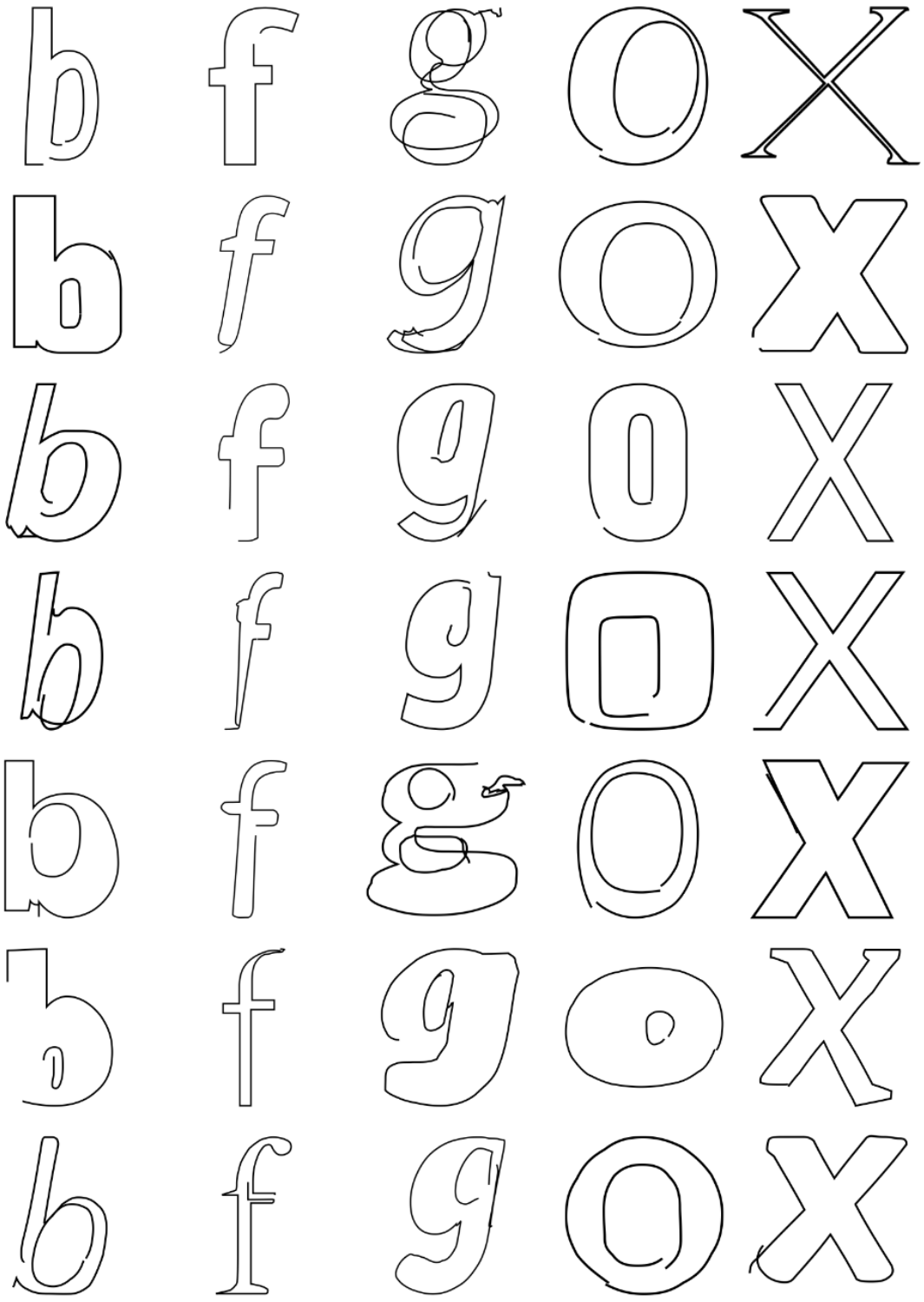


Figure 5-6: We generate novel, unseen examples by training the decoder only without any latent variable input, feeding only the previously generated command into the decoder cells. The decoder is run with initial states set at 0, and sampling outputs is done at $\tau = 0.01$.

Chapter 6

Feature representation variability

When translating input SVGs into feature vectors, we have to make key decisions about how SVG commands are specifically transformed into the nine-dimensional encoding that our model accepts as input. Some examples of variation are:

- **Absolute vs. relative coordinates**: are start, end, and control points represented in terms of their absolute position in the drawing or their relative displacement between each other?

- **Coordinate system orientation**: if relative, do we specify displacement vectors in terms of the normal $\{(1, 0), (0, 1)\}$ basis, or do we transform their values such that they represent deviations from continuing in the same direction as the previous point?

- **Pairwise coordinate choice**: if relative, between which points in the curves' coordinate parameters do we measure displacement?

Here, we report results of an experiment in which we find that alternative forms of our feature adaptation process have varying "learnability"; training differently transformed inputs with the same model architecture produces outputs of varying quality.

Table 6.1: The differences between the five feature representations used for the encoding efficacy experiment. Each feature encoding uses six dimensions for the cubic Bézier parameters (two for each point vector) and three for the pen state (pen up, pen down, end drawing). Note that `s` represents the start coordinates of the curve, `e` represents the end coordinates of the curve, `c1` represents the coordinates of the curve's first control point, and `c2` represents the coordinates of the curve's second control point. `disp(a, b)` indicates displacement between points `a` and `b`, and `rot(v)` indicates that vector `v` has been rotated such that its coordinates are in the direction of the previous vector in the encoding.

Encoding	Feature vector description
A	`disp(s, e), disp(s, c1), disp(s, c2), pen_state`
B	`disp(s, c1), disp(c1, c2), disp(c2, e), pen_state`
C	`disp(s, e), rot(disp(s, c1)), rot(disp(c2, e)), pen_state`
D	`e, rot(disp(s, c1)), rot(disp(c2, e)), pen_state`
E	`e, c1, c2, pen_state`

6.1 Evaluating feature encodings

To investigate the effects of choices for the above questions, we train five models each with a different feature encoding for input and output drawings. Code for generating each encoding can be found in Appendix **??**.

All models are trained using the same architecture as described in Chapter 4 for 125k steps. To generate the differently encoded inputs, the same base dataset of SVGs for the glyph **b** in various font faces is transformed to produce a list of feature vectors per glyph for each representation in Table 6.1. The base dataset is then partitioned randomly into 1920 training examples, 240 validation examples, and 240 test examples. We follow the original training procedure described in Section 5.2 with the same hyperparameters. Training loss graphs and test loss values are included in Appendix **??**.

6.2 Results

We evaluate results quantitatively by computing the Hausdorff similarity metric between each ground truth image and a corresponding image conditionally generated by the model with $\tau = 0.3$, using the same method as described in Section 5.3.1.

Table 6.2: Modified Hausdorff distance between conditionally generated and ground truth images for models trained on the **b** dataset with each encoding on a test set of N images. Some pairs are omitted from comparison because the model failed to decode inputs into a valid (non-null) output.

Encoding	Mean	Std. dev.	Kurtosis	N pairs
A	28.7707	11.9332	1.3354	239
B	15.4189	8.4912	3.1740	239
C	24.8122	14.5086	17.9476	233
D	15.9723	7.6134	0.9394	240
E	16.7207	7.1926	0.8920	240

Evaluation is run on a set of N test set images for each encoding, and quantitative results are reported in Table 6.2. Sample conditionally generated images can be found in Figure 6-1.

While encoding E maintains high image similarity as measured by the Hausdorff metric, its generated outputs tend to lack smooth curves and straight lines and are characterized instead by jagged, bite-mark shaped curves. This demonstrates a potential shortcoming of our Hausdorff similarity metric: training a model on strokes' absolute positions seems to result in greater preservation of the original glyph shape but may make learning style properties more difficult.

Encodings B and D seem to result in the glyphs most visually similar to ground truth glyphs and score relatively well on generated image similarity. We interpret this finding as suggesting that the model learns style and structure better when SVG data is encoded such that features represent displacement between adjacent control points—for example, start point and first control point, or second control point and end point. Based on this experiment, we use encoding B for our model evaluations in Chapters 5 and 7.

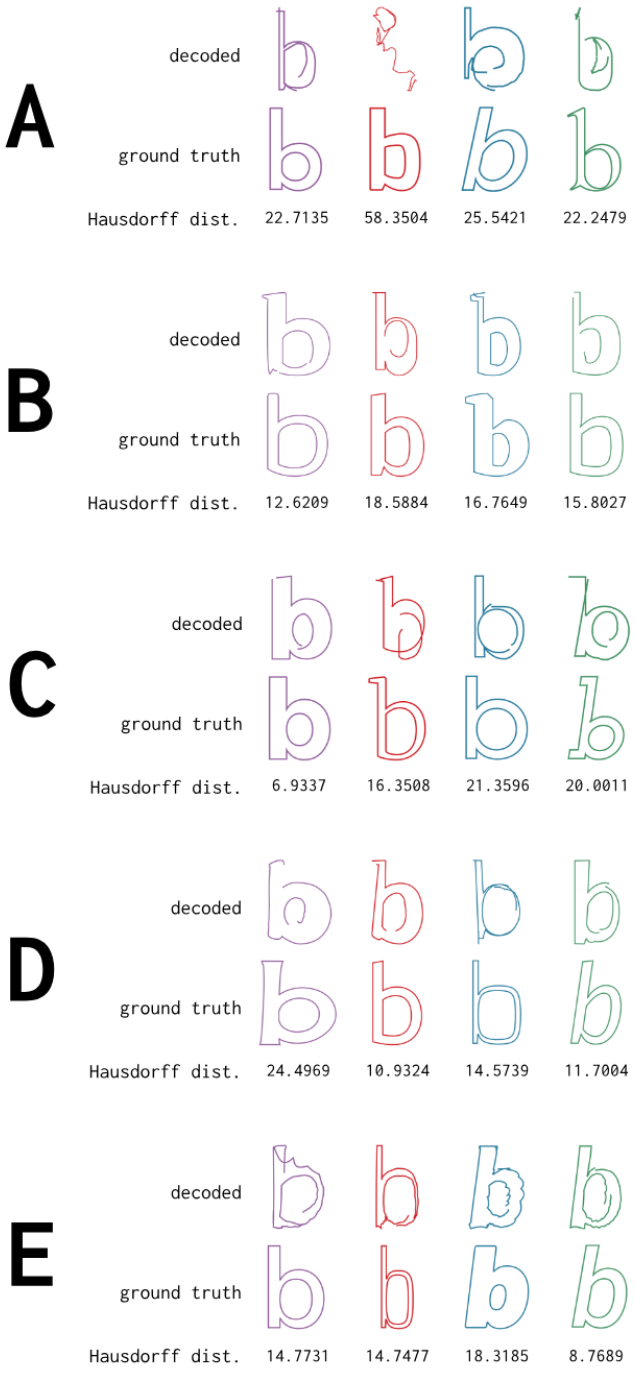


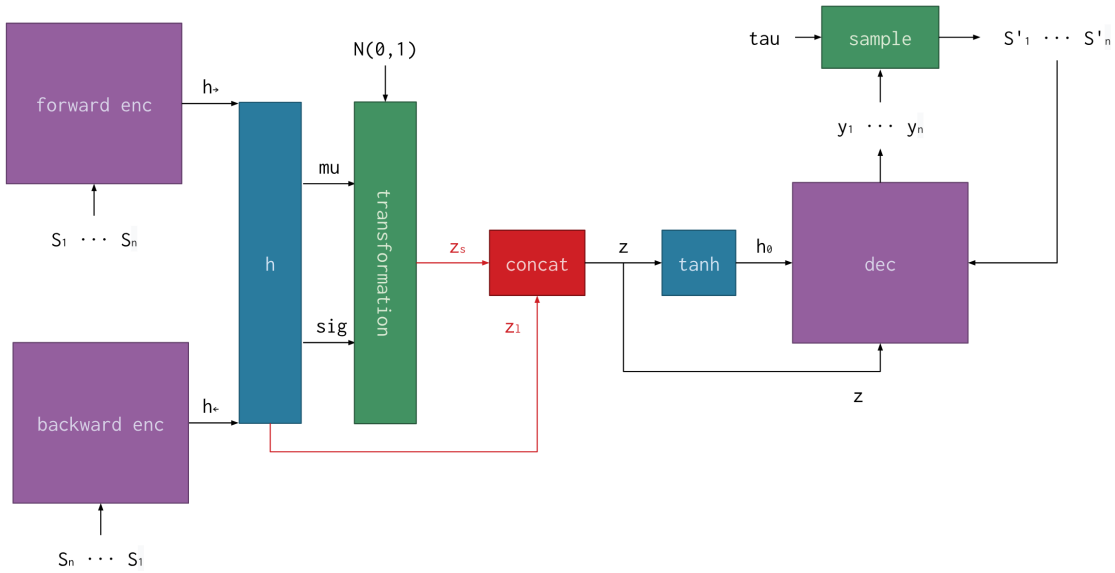
Figure 6-1: Randomly sampled input-output pairs from the SVG models trained on the five encodings described in Table 6.1. Decoded outputs were generated at $\tau = 0.3$. Decoded outputs are found in the top row for each encoding, while ground truth inputs are in the bottom row.

Chapter 7

Style and content

Although the model presented in Chapter 4 learns to model SVGs from a single class, training on a multi-class dataset forces it to encode style and class together in its latent space. Here, we investigate a modified architecture that explicitly encodes style and class separately. We report experiment results and demonstrate applications in classification and style transfer.

Figure 7-1: An overview of the SVG model architecture with explicit label encoding. The differences between this figure and Figure 4-1 are highlighted in red.



7.1 Model architecture

The model presented originally transforms the final layer of the encoder RNN into μ and σ vectors, which are then combined with random $\mathcal{N}(0, 1)$ samples to form a latent space vector z.

We modify this architecture to also generate a label vector z_l from the encoder's final layer that models a categorical distribution over input classes. Assuming there are n_c input classes and a latent space dimension of n_z, we adjust μ and σ to be of dimension $n_z - n_c$, then perform the original transformation from μ and σ to generate what we now denote as z_s. We concatenate these two vectors, z_l and z_s, to form z, which is then fed into the decoder as before.

Finally, we add another term in the loss function representing the log loss of the label prediction with the ground truth label (encoded as a n_c-dimensional one-hot vector). With the ground truth label encoded as $(c_1, \cdots, c_{n_c})$, the log loss of label prediction is:

$$L_c = \sum_{i=1}^{c} c_i \log(z_{li}) \tag{7.1}$$

7.2 Classifying experiment

We perform an experiment comparing two models: one version of the original model and one version of the modified classification model. To generate a multi-class dataset, we combine datasets for the digit glyphs (i.e. digits 0 through 9). Input SVGs are encoded using encoding B as described in Chapter 6. Overall, the resulting dataset has 20k training examples, 2.4k test examples, and 2.4k validation examples.

Both models were trained for 450k iterations with the same parameters as models described in Chapter 5, and details about training loss can be found in Appendix **??**.

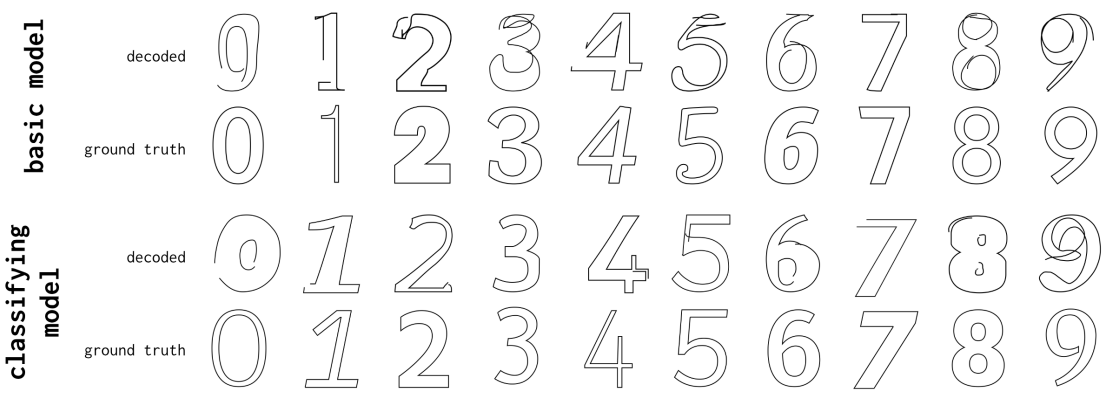


Figure 7-2: Selected conditionally generated digits from the original model in Chapter 4 and the modified model presented in Section 7.1 trained on a multi-class digits dataset.

7.3 Results

In conditionally recreating input images, we find that the models are comparable, with the original model performing slightly better in quantitative evaluation. Selected decoded outputs from both models are shown in Figure 7-2.

Table 7.1: Modified Hausdorff distance for models trained on the digit glyph dataset with each encoding on a test set of N images. We include two baselines: one comparing every input digit image against random noise of the same image dimensions, and one comparing randomly selected pairs of input images.

Model	Mean	Std. dev.	Kurtosis	N pairs
Original	25.5308	10.5062	3.1082	2400
Modified	26.6192	16.9867	15.8120	2326
Baselines				
Random noise	183.6398	27.1631	1.8888	2400
Input pairs	33.6121	13.3207	0.6291	500

We evaluate results quantitatively by computing the Hausdorff similarity metric between each ground truth test set image and a corresponding image conditionally generated by the model with $\tau = 0.1$, using the same method as described in Section 5.3.1. Results are reported in Table 7.1. We also include a random noise baseline and an input pairs baseline.

Table 7.2: Classification accuracy for the modified multi-class model.

Digit	Classification accuracy
0	82.08%
1	41.25%
2	92.91%
3	97.08%
4	49.16%
5	56.66%
6	61.25%
7	80.83%
8	95.83%
9	88.33%

Finally, we report the classification accuracy from the modified model, generated by comparing the highest probability predicted class to the ground truth class. The modified model has an overall classification accuracy of 74.54%, with accuracy rates per class reported in Table 7.2.

One factor that may partially explain the classifying model's lack of improvement in generating conditional images over the basic model is the reduced latent vector size. Both models set n_z to be 128, but the basic model is able to use the entire latent vector to encode the input drawing, while the classifying model must reserve 10 dimensions for the digit class vector. Future work can explore this hypothesis as well as experiment with other adjustments to the model architecture to further improve performance.

7.4 Style transfer

We verify that the class encoding is used by the decoder by manually substituting out the portion of z corresponding to z_c with a different class encoding. Although we did not quantitatively evaluate these results, we report visual results for more common font styles in Figure 7-3.

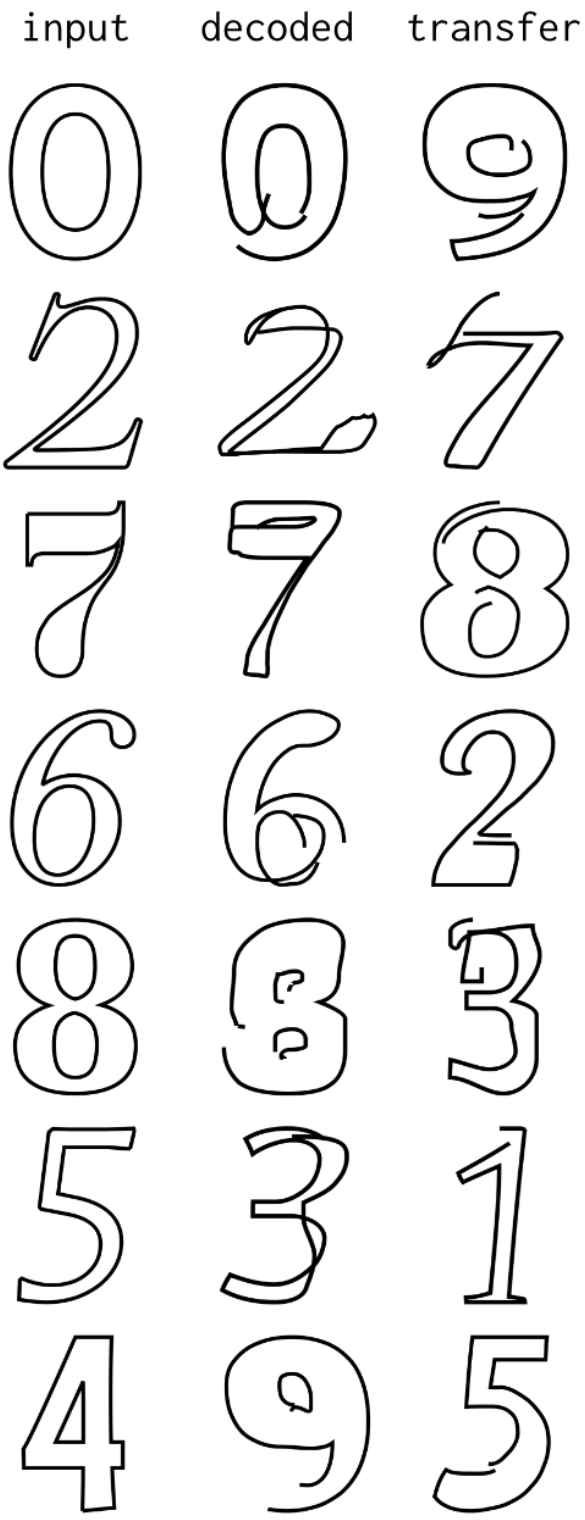


Figure 7-3: Examples of style transfer using the classifying model. Outputs were decoded at $\tau = 0.1$. In the last two rows, the model misclassified the input 5 as 3 and the input 4 as 9, resulting in the incorrect decoded output.

Chapter 8

Conclusion

Through this work, we demonstrate the viability of an automatic vector graphics generation method by modeling SVGs as sequences of drawing commands. Inspired by [17], our approach extends the sequence-to-sequence variational autoencoder model to learn curve control points as well as pen drawing points, producing parameterized SVG curves as output.

We train the model on font glyphs, first establishing single-class models of selected character glyphs and evaluating produced output qualitatively and quantitatively. Next, we investigate the effects of different feature encodings on model performance, quantified using an image similarity metric on raster image output, and we find significant differences in generation performance likely related to control point proximity in the feature representation. Lastly, we explore adjustments to explicitly encode style and content separately in the architecture and train on a multi-class set of digit glyphs. Although we do not see major differences in conditional generation performance in this experiment, we demonstrate learned classification and style transfer.

There is certainly room for future work. The model's generative performance could always be improved, perhaps with the help of latent space interpretation, and post-processing of output glyphs may help solve common failure modes in the current model like disconnected paths and misplaced components. We also believe a proof-of-concept design suggestion tool that proposes preliminary drawings and allows for interactive editing would be illustrative. Additionally, further work is needed to

demonstrate the generalizability of the model, especially on other domains such as icons and logos.

Our approach may be exploratory, but it sets the groundwork for future development of creative tools for designers. By avoiding explicit parameterizations for vectorized images, we build a framework for generalizable tools that propose novel first draft designs. Through this work, we make steps towards a future in which the design process becomes less monotonous and more creative.

TODO: put appendix back in

Bibliography

[1] J. Wu, J. B. Tenenbaum, and P. Kohli, "Neural scene de-rendering," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 699–707.

[2] M. Iyyer, V. Manjunatha, A. Guha, *et al.*, "The amazing mysteries of the gutter: Drawing inferences between panels in comic book narratives," *ArXiv preprint arXiv:1611.05118*, 2016.

[3] M. J. Seo, H. Hajishirzi, A. Farhadi, and O. Etzioni, "Diagram understanding in geometry questions.," in *AAAI*, 2014, pp. 2831–2838.

[4] Z. Bylinskii, S. Alsheikh, S. Madan, *et al.*, "Understanding infographics through textual and visual tag prediction," *ArXiv preprint arXiv:1709.09215*, 2017.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[6] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *ArXiv preprint arXiv:1409.1556*, 2014.

[7] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[8] A. Karpathy, P. Abbeel, G. Brockman, *et al.* (2016). Generative models, [Online]. Available: `https://blog.openai.com/generative-models/`.

[9] I. Goodfellow, J. Pouget-Abadie, M. Mirza, *et al.*, "Generative adversarial nets," in *Advances in neural information processing systems*, 2014, pp. 2672–2680.

[10] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *ArXiv preprint arXiv:1312.6114*, 2013.

[11] C. Doersch, "Tutorial on variational autoencoders," *ArXiv preprint arXiv:1606.05908*, 2016.

[12] A. Graves, "Generating sequences with recurrent neural networks," *ArXiv preprint arXiv:1308.0850*, 2013.

[13] K. Gregor, I. Danihelka, A. Graves, D. J. Rezende, and D. Wierstra, "Draw: A recurrent neural network for image generation," *ArXiv preprint arXiv:1502.04623*, 2015.

[14] Y. Ganin, T. Kulkarni, I. Babuschkin, S. Ali Eslami, and O. Vinyals, "Synthesizing programs for images using reinforced adversarial learning," 2017.

[15] R. D. Janssen and A. M. Vossepoel, "Adaptive vectorization of line drawing images," *Computer vision and image understanding*, vol. 65, no. 1, pp. 38–56, 1997.

[16] T. Birdal and E. Bala, "A novel method for vectorization," *ArXiv preprint arXiv:1403.0728*, 2014.

[17] D. Ha and D. Eck, "A neural representation of sketch drawings," *ArXiv preprint arXiv:1704.03477*, 2017.

[18] D. E. Knuth, *TEX and METAFONT: New directions in typesetting*. American Mathematical Society, 1979.

[19] V. M. Lau, "Learning by example for parametric font design," in *ACM SIGGRAPH ASIA 2009 Posters*, ACM, 2009, p. 5.

[20] T. Hassan, C. Hu, and R. D. Hersch, "Next generation typeface representations: Revisiting parametric fonts," in *Proceedings of the 10th ACM symposium on Document engineering*, ACM, 2010, pp. 181–184.

[21] J. B. Tenenbaum and W. T. Freeman, "Separating style and content," in *Advances in neural information processing systems*, 1997, pp. 662–668.

[22] N. D. Campbell and J. Kautz, "Learning a manifold of fonts," *ACM Transactions on Graphics (TOG)*, vol. 33, no. 4, p. 91, 2014.

[23] S. Xu, F. C. Lau, W. K. Cheung, and Y. Pan, "Automatic generation of artistic chinese calligraphy," *IEEE Intelligent Systems*, vol. 20, no. 3, pp. 32–39, 2005.

[24] P. Upchurch, N. Snavely, and K. Bala, "From a to z: Supervised transfer of style and content using deep neural network generators," *ArXiv preprint arXiv:1603.02003*, 2016.

[25] Z. Wang, J. Yang, H. Jin, *et al.*, "Deepfont: Identify your font from an image," in *Proceedings of the 23rd ACM international conference on Multimedia*, ACM, 2015, pp. 451–459.

[26] Z. Lian, B. Zhao, and J. Xiao, "Automatic generation of large-scale handwriting fonts via style learning," in *SIGGRAPH ASIA 2016 Technical Briefs*, ACM, 2016, p. 12.

[27] A. Grasso, C. Lilley, D. Jackson, *et al.*, "Scalable vector graphics (SVG) 1.1 (second edition)," W3C, W3C Recommendation, Aug. 2011, http://www.w3.org/TR/2011/REC-SVG11-20110816/.

[28] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *ArXiv preprint arXiv:1607.06450*, 2016.

[29] M.-P. Dubuisson and A. K. Jain, "A modified hausdorff distance for object matching," in *Pattern Recognition, 1994. Vol. 1-Conference A: Computer Vision & Image Processing., Proceedings of the 12th IAPR International Conference on*, IEEE, vol. 1, 1994, pp. 566–568.