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
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

TODO:

Thesis Supervisor: Frédo Durand
Title: Professor

Acknowledgments

TODO: acknowledgements

Contents

List of Figures

List of Tables

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 distribting 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, unseen examples and 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 believe demystifying their generation would further solidify understanding of the intent and structure of human-created designs. In algorithmically mimicking the process by which glyphs are created, we hope to gain a deeper grasp of from where humans' notion of design and style arises.

Although many strides have been made in understanding and synthesizing rasterized images and designs, primarily with 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.

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 single-class as well as a multi-class approach. Finally, we evaluate our results quantitatively and qualitatively and consider future directions of work.

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 hoping to apply generative methods to vectorized glyphs, we are inspired by previous work in computer vision on non-natural images and draw upon recent developments in 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, sometimes multimodal nature, and human-imbued intent.

While much attention has been focused on research problems like object recognition, scene segmentation, and image recognition 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 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. Within a font face, glyphs generally share the same style for properties like angles, curvature, presence of serifs, and stroke width.

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 essentially 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 difficult problem.

In this work, our computational methods are applied to font faces downloaded from Google Fonts[1]. In Figure 2-1, 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 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 font hints to aid with rendering.

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

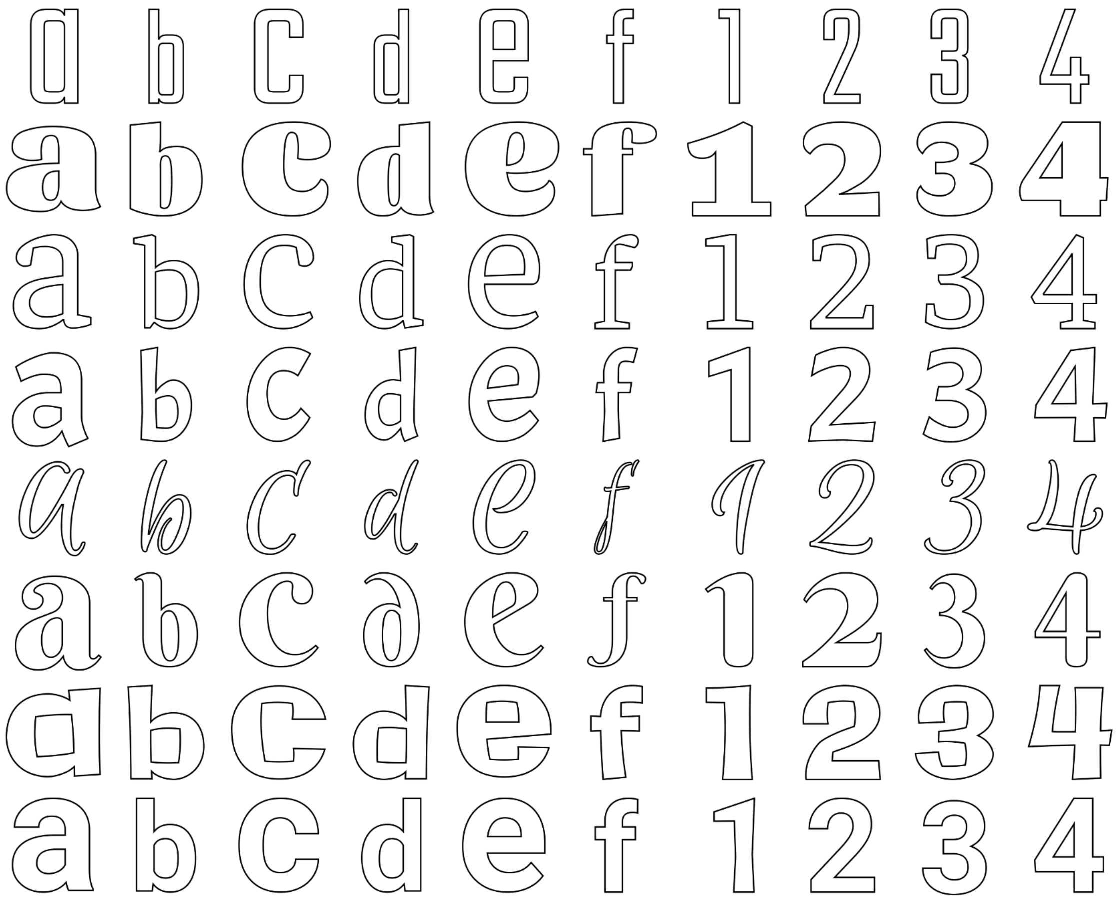
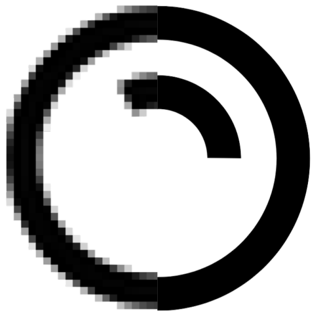
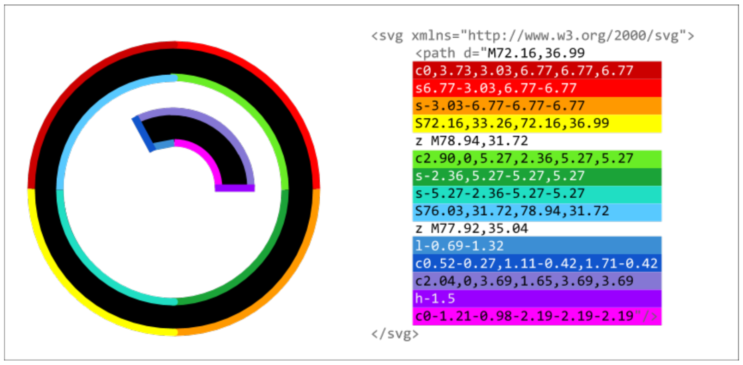


Figure 2-1: A sample of the types of font glyphs used in our dataset. Lowercase letters “a” through “f” are shown, as well as digits 1 through 4. Note the variety of styles represented: the dataset includes serif, sans serif, display, and handwriting style fonts.



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



```
<svg xmlns="http://www.w3.org/2000/svg">
    <path d="M72.16,36.99
    c0,3.73,3.03,6.77,6.77,6.77
    s6.77-3.03,6.77-6.77
    s-3.03-6.77-6.77-6.77
    S72.16,33.26,72.16,36.99
    z M78.94,31.72
    c2.90,0,5.27,2.36,5.27,5.27
    s-2.36,5.27-5.27,5.27
    s-5.27-2.36-5.27-5.27
    S76.03,31.72,78.94,31.72
    z M77.92,35.04
    l-0.69-1.32
    c0.52-0.27,1.11-0.42,1.71-0.42
    c2.04,0,3.69,1.65,3.69,3.69
    h-1.5
    c0-1.21-0.98-2.19-2.19-2.19"/>
</svg>
```

(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 2-2: a visual comparison of raster and vector graphics. Figure 2-2b walks through a sample vector graphics path. Image source: *Dog Wash* by Llisole from the Noun Project.

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. 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 2-2a). As a data format, SVGs describe lists of geometric objects (among other elements), 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 2-2b.

The early popular deep convolutional networks designed to process images seen 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 images. Naturally, this fixed-dimension format is incompatible with SVG element lists. 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.

Two 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 $D(x; \theta_g)$ that learns to discriminate samples from the ground truth dataset and the generative model's latent space. When both models are differentiable functions, backpropagation can be used to train G and D 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 that takes z vectors sampled from a probaility distribution $P(z)$ and applies a function that produces a random variable 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 $Q_\phi(z|x)$ that approximates $P(z|x)$, while the decoder learns $P_\theta(x|z)$. We can then define a loss function derived from Kullback-Leibler divergence $\mathcal{D}$ that describes a variational lower bound:

$$L_i = -E_{z \sim Q_\phi(z|x_i)}[\log P_\theta(x_i|z)] + \mathcal{D}(Q_\phi(z|x_i)||P(z)) \tag{2.2}$$

TODO: reference https://jaan.io/what-is-variational-autoencoder-vae-tutorial/

In [12], this VAE model is expanded to learn sequential inputs and outputs. TODO: explain VAE + RNN

2.3 Related work

In our work, we build upon the variational autoencoder method presented by Ha *et al.* in [13].

2.3.1 Generating drawings

TODO: UToronto handwriting generation

TODO: DRAW

TODO: SPIRAL

2.3.2 Font style

TODO: Learning a Manifold of Fonts

Chapter 3

SVG feature representation

Our goal is to extend beyond polyline modeling and capture the higher-level shapes of SVG objects. Thus, our first 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, we simplify the problem space to focus only on `path`s (since `path`s can be used to compose other elements anyway).

3.1 Overview of SVG commands

There are five commands possible in an SVG `path` as seen in Table 3.1 [14]:

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 Section 4.2.

Table 3.1: A description of possible SVG path commands. For simplicity, we omit the absolute coordinate variants of the commands, which specify absolute (x, y) coordinates instead of relative (dx, dy).

Command	Description	Code
Move	moves the pen to a specified position	`m dx dy`
Line	draws a line from the start to the end position	`l dx dy`
Quadratic Bézier	draws a quadratic curve according to given control point	`q dx1 dy1, dx dy`
Cubic Bézier	draws a cubic curve according to given control points	`c dx1 dy1, dx2 dy2, dx dy`
Arc	draws a section of an ellipse	`a rx ry t fl fs dx dy`

3.2.1 Preprocessing

SVG icons and font glyphs often have additional properties like stroke and fill style. As we focus on path generation exclusively, our first step in transforming input SVGs 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 2-1 for examples.

Often, designers do not craft SVGs by editing XML by hand but rather with interactive image editing software such as Adobe Illustrator or Inkscape (TODO: citations?). Thus, human-created SVGs often have varying path compositions, viewboxes, and canvas sizes.

To constrain the generation process slightly, we first homogenize our dataset by preprocessing inputs to rescale to the same overall canvas size (set to 256×256 pixels) as well as reorder paths in the drawing sequence so that paths with larger bounding boxes are drawn first.

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

3.2.2 Simplifying path commands

We aim to produce a system capable of modeling general SVGs, so inputs can contain all path commands specified in Table 3.1. To avoid bias and to constrain the problem space (TODO: wording?), we consolidate the different path commands into a single feature representation. TODO: this justification makes more sense with the dataset statistics of icons, but fonts use only quadratic beziers–how to better justify why we convert arcs to cubic beziers even though that's not a lossless process?

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), but they occur relatively rarely in the dataset, so it makes sense to approximate them with the same parameter space as used for our parametric commands. We use the following method to approximate arc segments as cubic Béziers (further mathematical explanation can be found in Appendix A).

1. Extract the arc parameters: start coordinates, end coordinates, ellipse major and minor radii, rotation angle from x and y axes, large arc flag, and sweep flag.

2. Transform ellipse into unit circle by rotating, scaling, and translating, and save those transformation factors.

3. Find the arc angle from the transformed start point to end point on the unit circle.

4. If needed, split the arc into segments, each covering an angle less than 90°.

5. Approximate each segment angle's arc on a unit circle such that the distance from the circle center to the arc along the angle bisector of the arc is equal to the radius, defining cubic Bézier control points.

6. Invert the transformation above to convert arcs along the unit circle back to the elliptical arc, transforming the generated control points accordingly.

7. Use these transformed control points to parameterize the output cubic Bézier curve.

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 [13].

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 [13]). Six dimensions are used to model three x, y coordinate parameters of cubic Béziers, and three dimensions are reserved for the pen up, pen down, and end drawing pen states. In the next section, we examine the details of this feature transformation process and how its tweaks affect modeling performance.

3.3 Feature representation variability

We find that alternative forms of this feature adaptation process have varying "learnability", as training variants of the transformed inputs with the same model architecture produces outputs of differing quality.

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?

Table 3.2: 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 applied a transformation to rotate its coordinate axes to be in the direction of the previous vector in the encoding TODO: wording?.

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`

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

3.3.1 Encoding evaluation

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.

All models are trained using the same architecture as described in Section 4.2, for TODO: count 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 set of feature vectors for each representation in Table 3.2. The base dataset is then partitioned randomly into 1920 training examples, 240 validation examples, and 240 test examples. SVGs whose total number of path commands is greater than 250 are pruned. Graphs depicting loss during the training process can be found in Appendix B.

Results

For each of the models, the model iteration with the best validation loss is selected for evaluation. Qualitative results from the encoding experiment can be found in

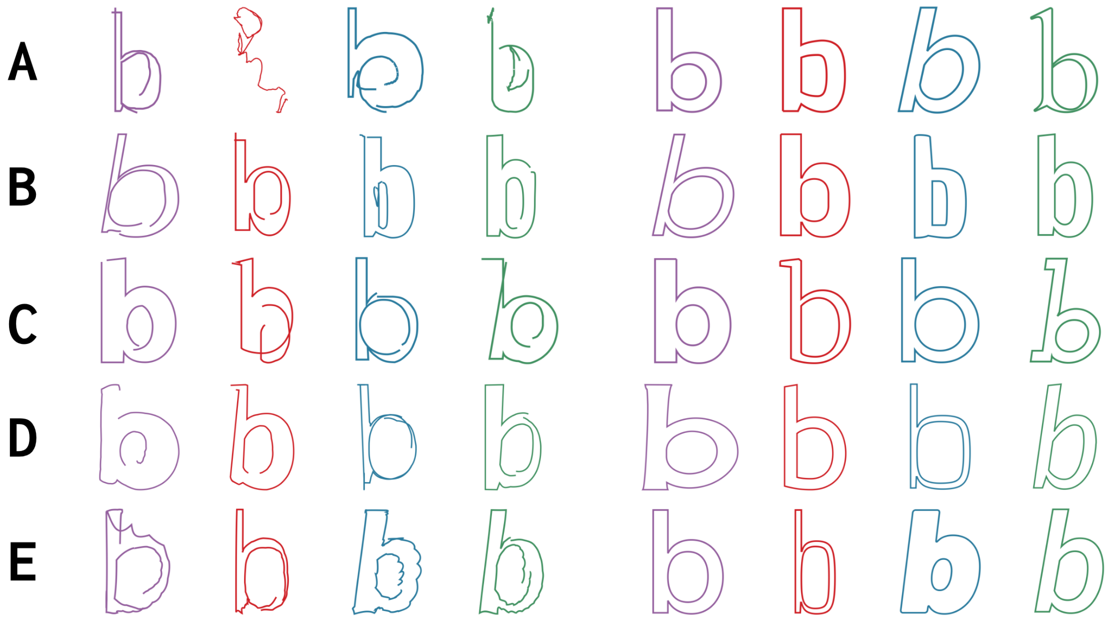


Figure 3-1: Randomly sampled input-output pairs from the SVG models trained on the five encodings described in Table 3.2. Decoded outputs were generated at temperature 0.3. Decoded outputs are found on the left, while ground truth inputs are on the right.

Figure 3-1.

We evaluate results quantitatively by TODO: eval!!!!.

Chapter 4

Font glyph generation

4.1 Dataset collection

4.2 Model architecture

4.2.1 Single-class

4.2.2 Multi-class

4.3 Training

4.4 Results

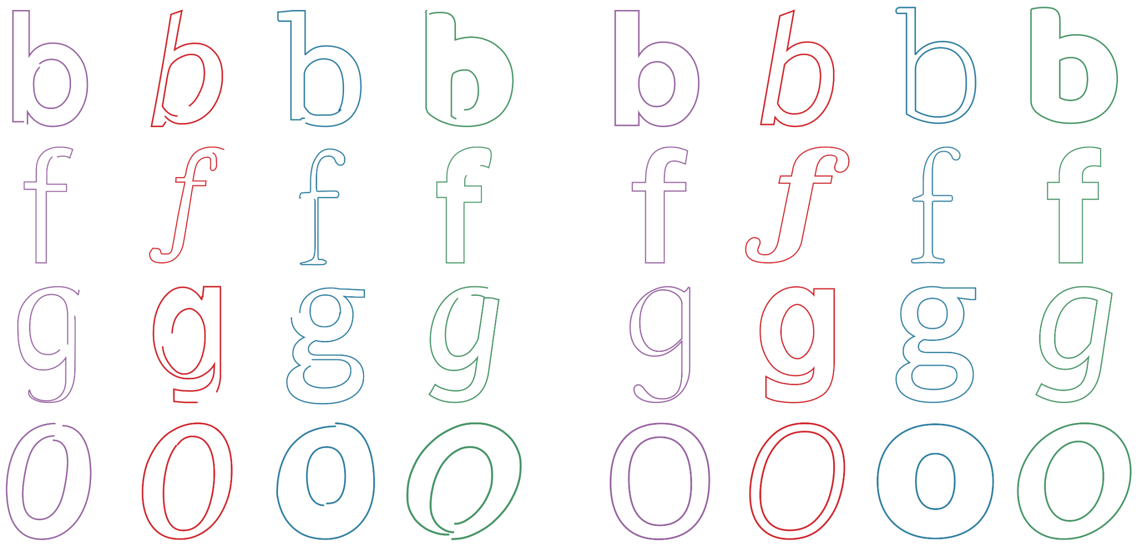


Figure 4-1: TODO: caption

Chapter 5

Evaluation

5.1 Qualitative results

5.1.1 Common modes of failure

- discussion of failures - vector interpolation

5.2 Quantitative results

Chapter 6

Conclusion

Appendix A

Algorithms

TODO: math for transforming arcs into cubics

Appendix B

Encoding experiment training

TODO: training graphs from encoding experiment

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, Y. Vyas, J. Boyd-Graber, H. Daumé III, and L. Davis, "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, A. Recasens, K. Zhong, H. Pfister, F. Durand, and A. Oliva, "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, P. Chen, V. Cheung, R. Duan, I. Goodfellow, D. Kingma, J. Ho, R. Houthooft, T. Salimans, J. Schulman, I. Sutskever, and W. Zaremba. (2016). Generative models, [Online]. Available: `https://blog.openai.com/generative-models/`.

[9] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "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] D. Ha and D. Eck, “A neural representation of sketch drawings,” *ArXiv preprint arXiv:1704.03477*, 2017.

[14] A. Grasso, C. Lilley, D. Jackson, J. Ferraiolo, P. Dengler, J. Watt, C. McCormack, J. Fujisawa, E. Dahlström, and D. Schepers, “Scalable vector graphics (SVG) 1.1 (second edition),” W3C, W3C Recommendation, Aug. 2011, http://www.w3.org/TR/2011/REC-SVG11-20110816/.