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. 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.

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

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

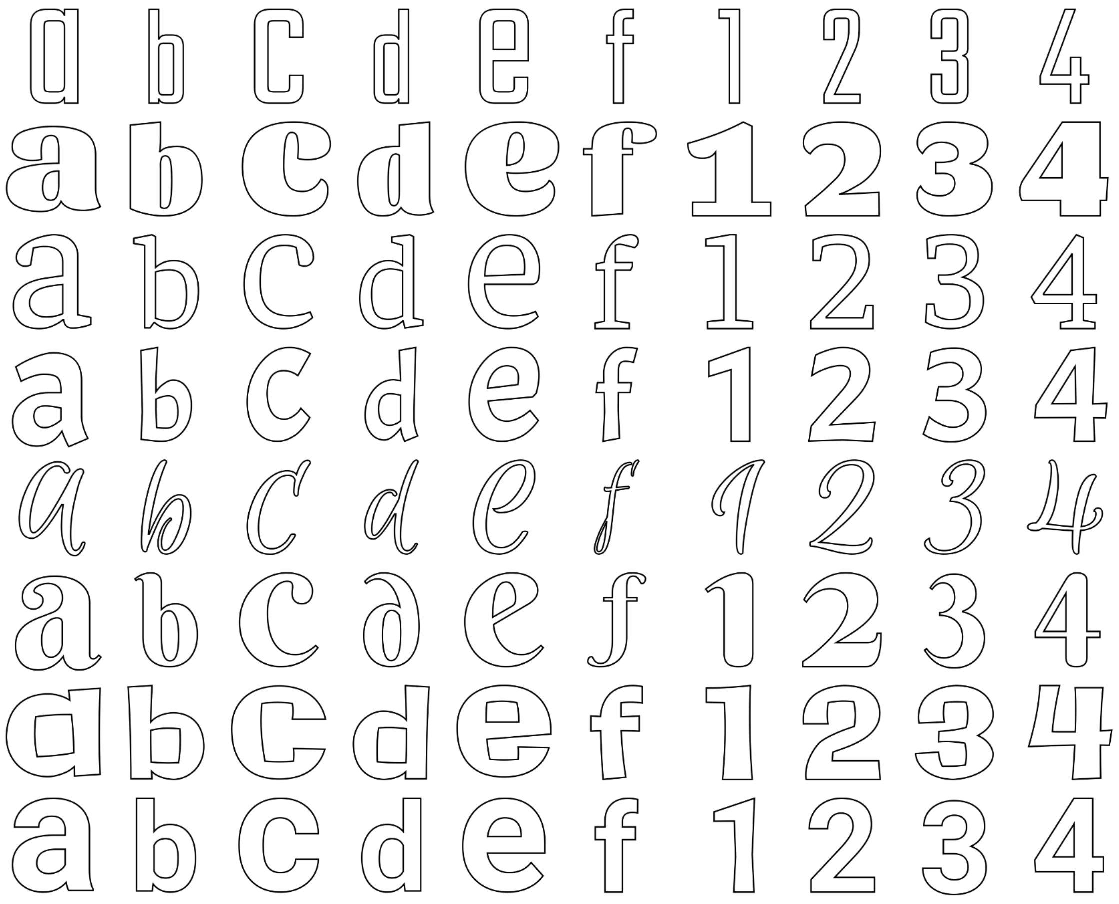
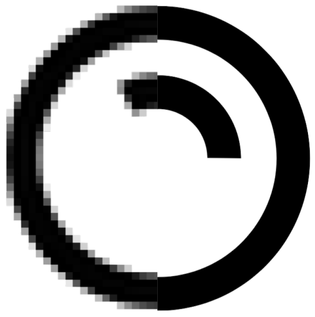
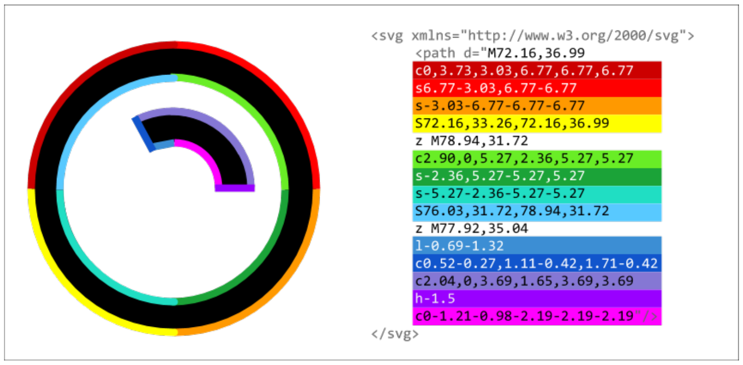


Figure 2-1: A sample of the types of font glyphs used in our dataset. Lowercase letters “a” through “f” are shown, as well as digits 1 through 4. Note the variety of styles represented: the dataset includes serif, sans serif, display, and handwriting style fonts.



(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.

(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.

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 that describes a variational lower bound, where Kullback-Leibler (KL) divergence $\mathcal{D}$ accounts for the similarity between our $Q_\phi(z|x)$ and the true $P(z)$ and the reconstruction loss accounts for similarity between input and output:

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

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 problems of polyline generation and procedural 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, Kulkarni, Babuschkin, Ali Eslami, and Vinyals uses a reinforcement-learning approach to train an agent to draw sketches.

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 an 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, Zhao, and Xiao 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, we simplify the problem space to focus only on `path`s (since `path`s 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].

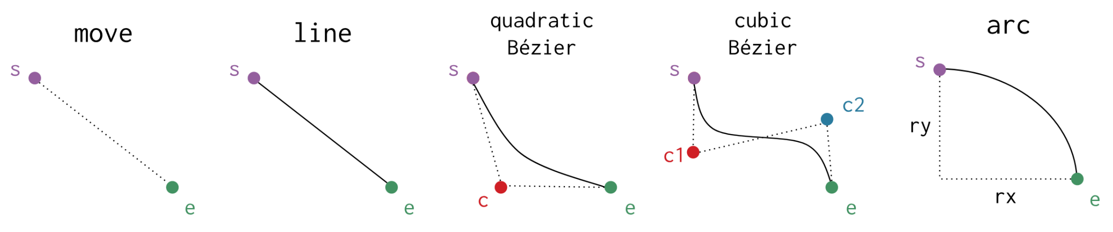


Figure 3-1: A visualization of the five commands in the SVG `path`

Table 3.1: A description of possible SVG path commands. For simplicity, we omit the relative coordinate variants of the commands, which specify relative (dx, dy) coordinates instead of absolute (x, y) for all pairs of 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	Description & Code
Move	moves the pen to a specified point (x, y) `M x y`
Line	draws a line from the current (start) point to the end point (x, y) `L x y`
Quadratic Bézier	draws a quadratic curve according to given control point (cx, cy) from the current point to the end point (x, y) `Q cx cy, x y`
Cubic Bézier	draws a cubic 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) `C cx1 cy1, cx2 cy2, x y`
Arc	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 `A rx ry t fl fs x y`

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 **??**.

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, view-boxes, 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:

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 [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 up (p_u), pen down (p_d), 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 the details of this feature transformation process and how its tweaks affect modeling performance.

Chapter 4

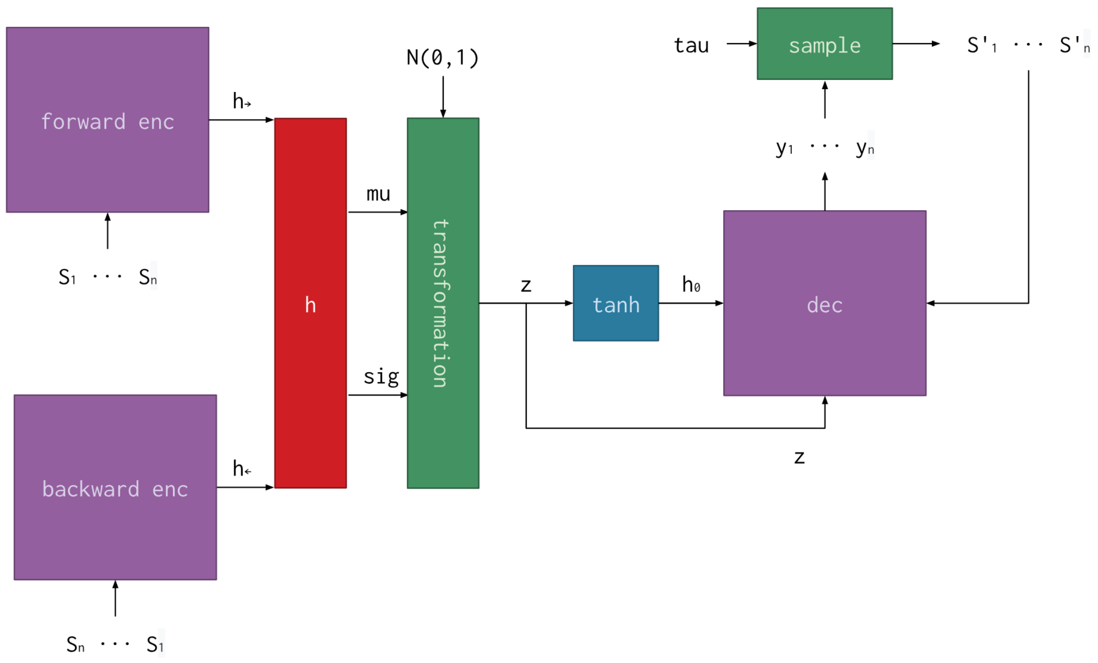
Model architecture

For our end-to-end SVG generation method, we build upon the work of 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 to four probability distributions instead of two: three Gaussian Mixture Models (GMMs) represent the three coordinates in the feature vectors, and one categorical distribution is used for modeling the pen state. 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 forward and one for 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 $S = (S_1, S_2, \cdots, S_n)$, 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 pen 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 pen location GMM models. 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. By default, each GMM contains 20 normal distributions, and each distribution is parameterized by a μ_x, σ_x, μ_y, σ_y, ρ_{xy} and has a mixture weight π. 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 $\{p_d, p_u, p_e\}$. A temperature multiplier τ is also used when sampling from GMMs and from the pen state distribution to adjust the randomness of the output. 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.

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. A breakdown of font types for the 877 font families can be found in Table 5.1.

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.

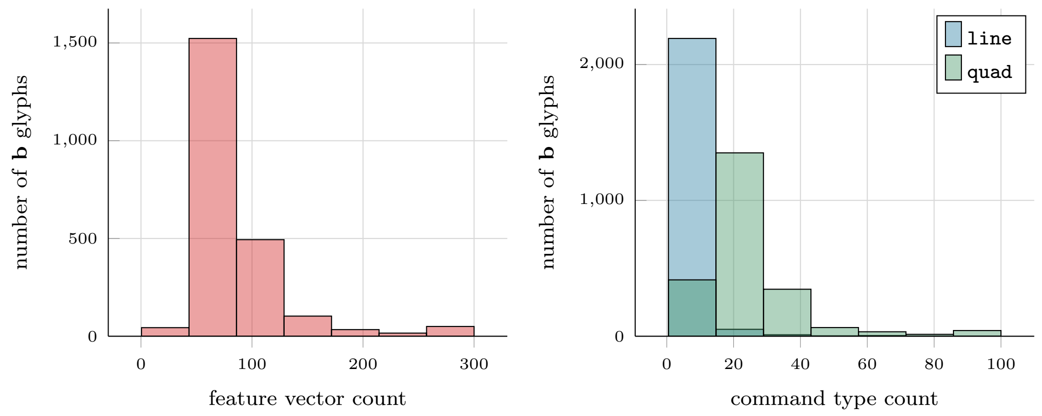
Serif	Sans serif	Display	Handwriting	Monospace
180	249	296	135	17

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

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

The statistics for the glyph **b** across all font faces are shown in Figure 5-1. The remaining glyph statistics can be found in Appendix B.

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



5.2 Training

We train the model on single-class datasets for 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, 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 slowly over time. The size of the latent space vector z is set to 128, and we use recurrent dropout to reduce overfitting. Cost graphs and details about trained models can be found in Appendix C.

5.3 Results

Here, we report quantitative results as well as a qualitative evaluation of model performance.

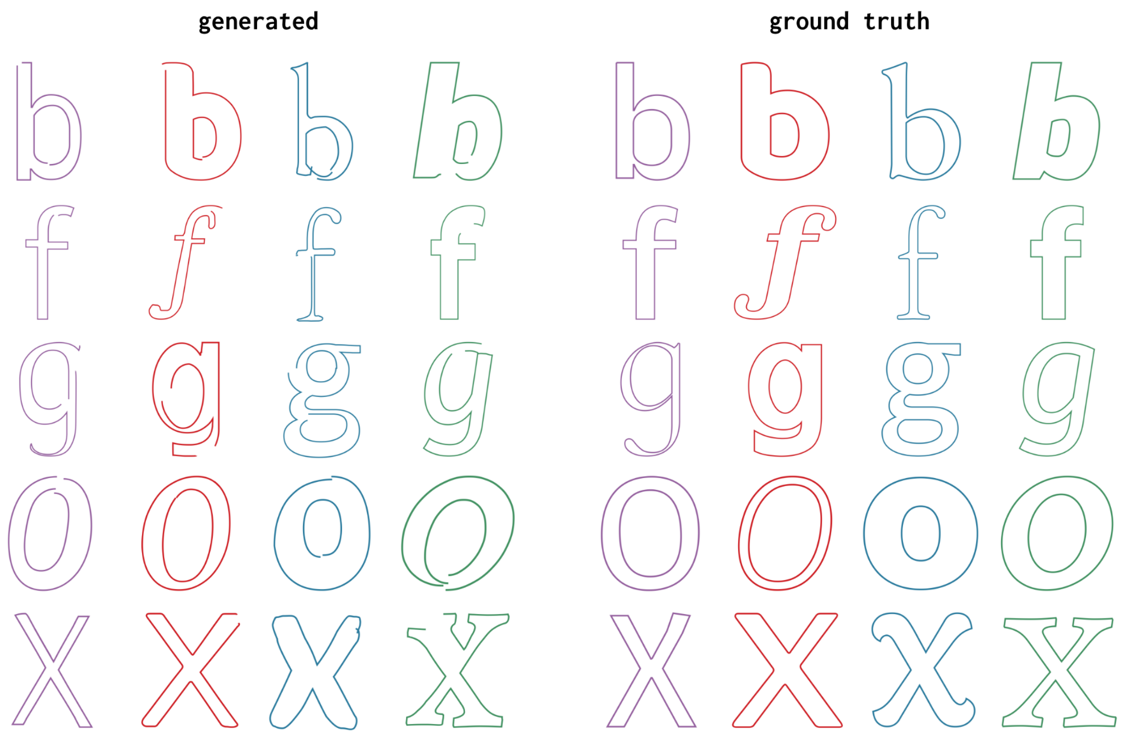


Figure 5-2: Selected glyphs conditionally generated by the trained single-class model. Ground truth inputs on the right are fed into the encoder and decoded into the generated outputs on the left.

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 increases 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 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

Table 5.2: Modified Hausdorff distance for models trained on each glyph on a test set of N images.

Glyph	Mean	Std. dev.	Kurtosis	N
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

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.

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 latent space is efficiently used and that the KL regularization term in the loss function has sufficient power.

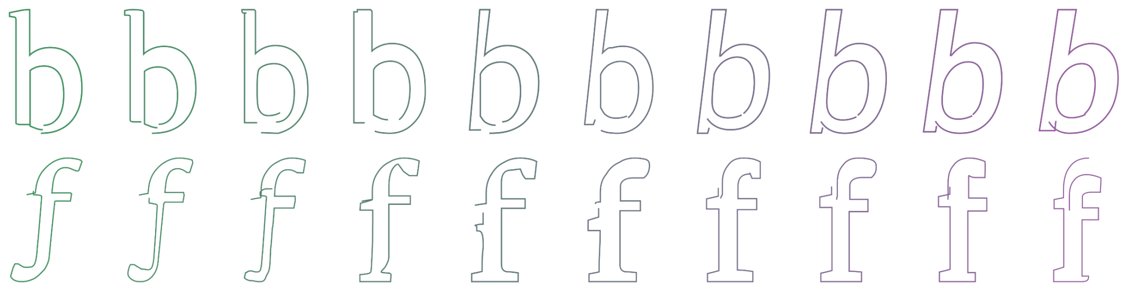


Figure 5-3: The latent space abstractly encodes both style and method—it must represent both how to draw a given glyph as well as the identify different types of glyphs. The decoded output for input glyphs are on the far sides of the figure, and we spherically interpolate between their latent vectors.

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

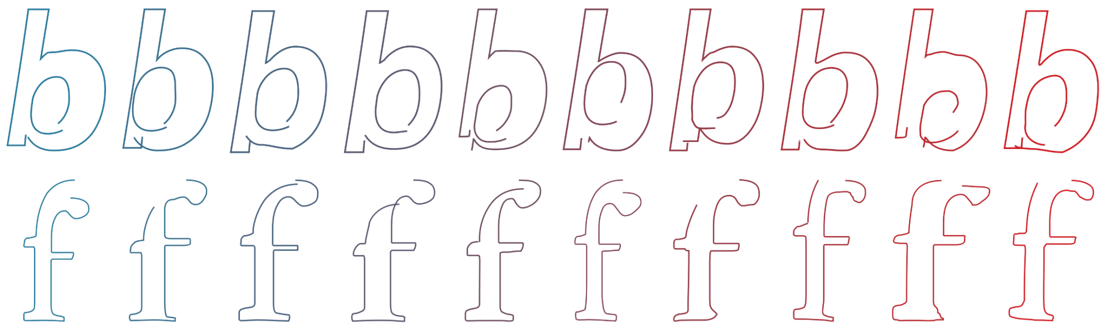


Figure 5-4: To demonstrate how temperature affects decoding, we decode the same latent vector at different temperature settings. At the far left, $\tau = 0.1$, and at the far right, $\tau = 1.0$, with τ increasing by 0.1 for every intermediate image.

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

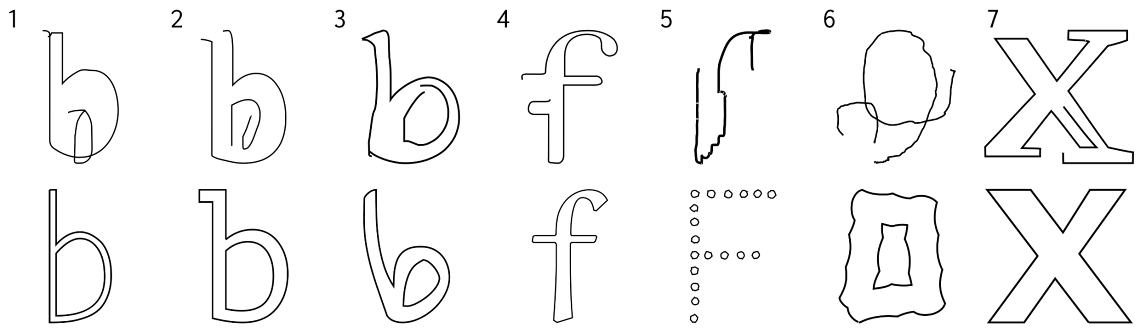


Figure 5-5: The model generally makes a set of common mistakes, as shown here. It struggles with positioning of disconnected components (1), closing paths (2, 4), understanding uncommon styles (3, 5, 6), and sometimes confuses styles (7).

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.

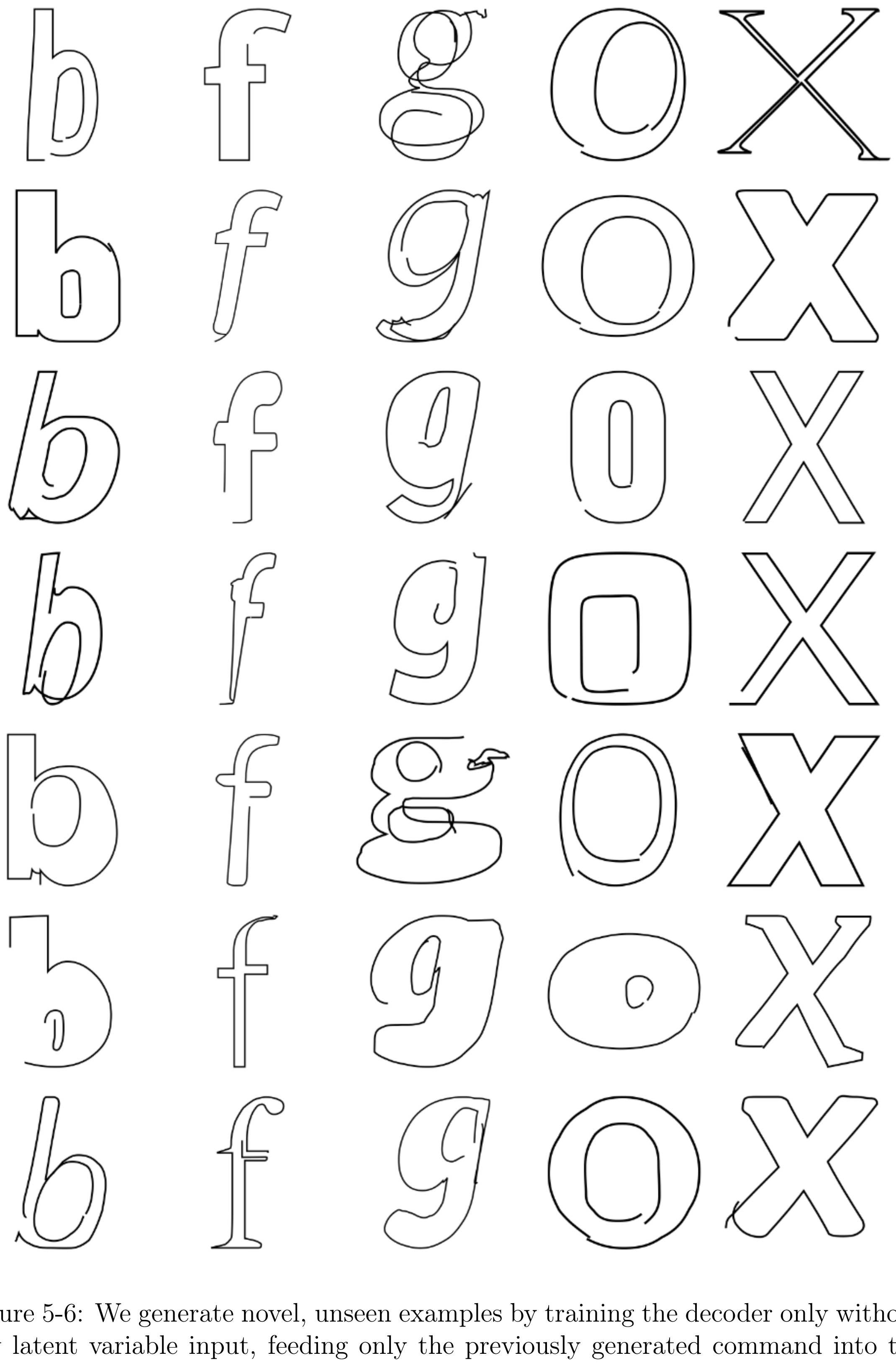


Figure 5-6: We generate novel, unseen examples by training the decoder only without any latent variable input, feeding only the previously generated command into the decoder cells.

Chapter 6

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?

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

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 Figure A-1.

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 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`

All models are trained using the same architecture as described in Section **??**, 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 set of feature vectors 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. SVGs whose total number of path commands is greater than 250 are pruned. Graphs depicting loss during the training process can be found in Figure C-2, and test costs for each selected model are reported in Tabel C.2.

6.2 Results

For each of the models, the model iteration with the best validation loss is selected for evaluation.

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. Evaluation is run on a set of N images for each encoding, and quantitative results can be found in Table 6.2.

Table 6.2: Modified Hausdorff distance for models trained on the **b** dataset with each encoding on a test set of N images.

Encoding	Mean	Std. dev.	Kurtosis	N
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

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. Still, 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. TODO: 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.

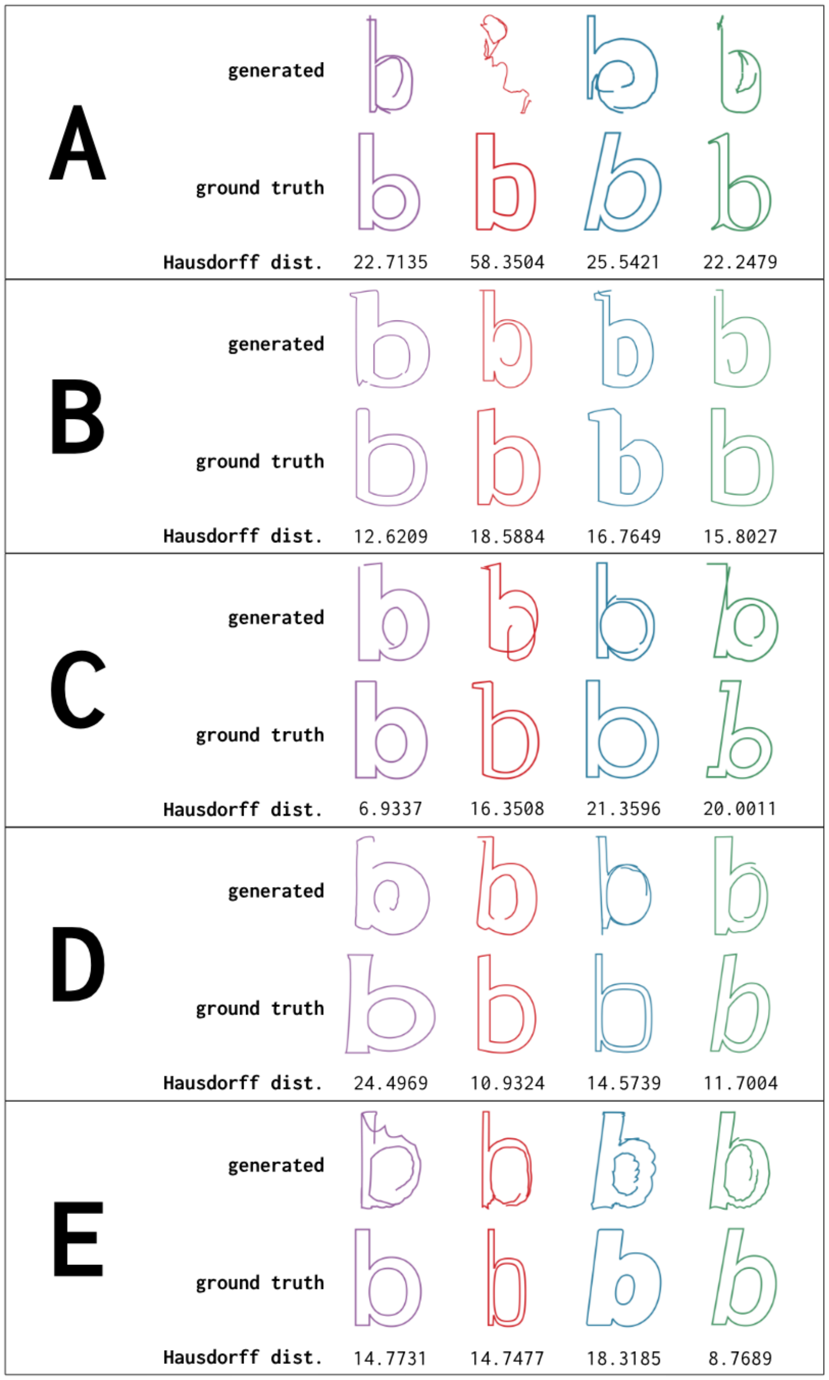

A
generated
ground truth
Hausdorff dist. 22.7135 58.3504 25.5421 22.2479
B
generated
ground truth
Hausdorff dist. 12.6209 18.5884 16.7649 15.8027
C
generated
ground truth
Hausdorff dist. 6.9337 16.3508 21.3596 20.0011
D
generated
ground truth
Hausdorff dist. 24.4969 10.9324 14.5739 11.7004
E
generated
ground truth
Hausdorff dist. 14.7731 14.7477 18.3185 8.7689

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 temperature 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

7.1 Model architecture

7.2 Results

7.2.1 Quantitative results

7.2.2 Qualitative results

Chapter 8

Conclusion

TODO: put appendix back in

Appendix A

Algorithms

Figure A-1: Code for generating encodings described in Table 6.1

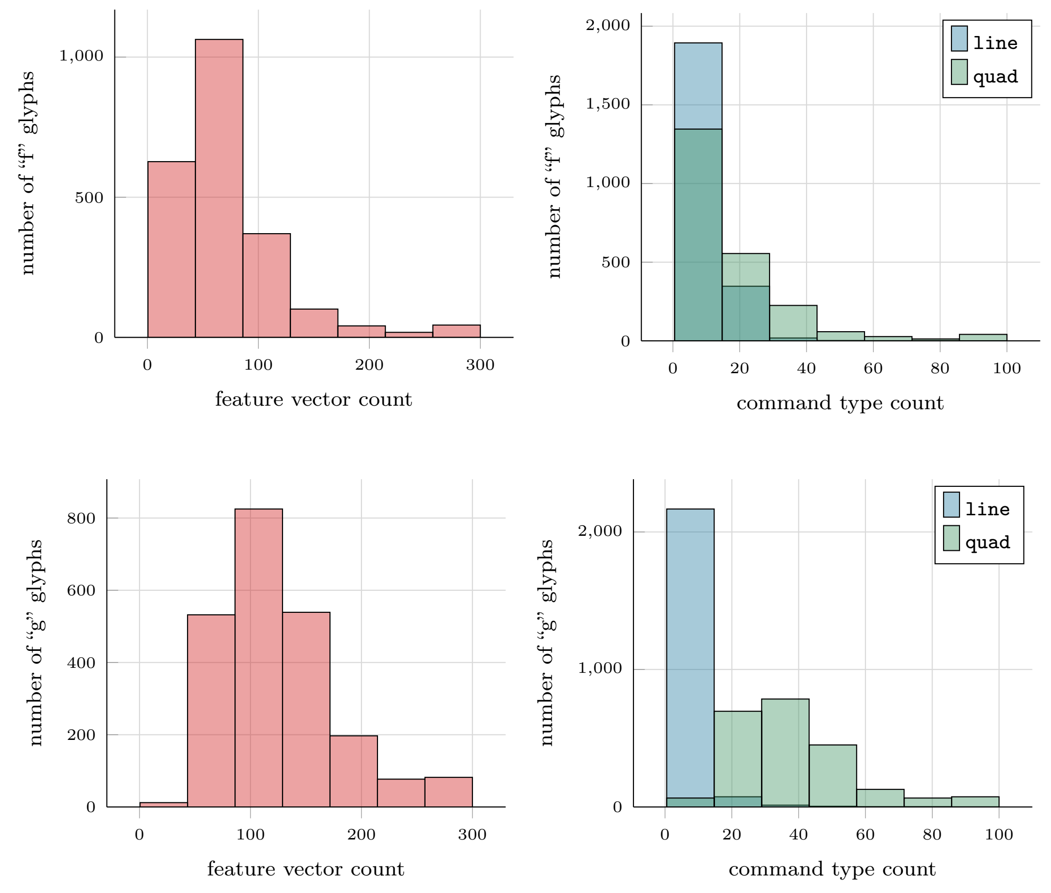
```
"""
vec[0] = d1.x
vec[1] = d1.y
vec[2] = d2.x
vec[3] = d2.x
vec[4] = d3.x
vec[5] = d3.x
vec[6] = pen_down
vec[7] = pen_up
vec[8] = end
"""
def rotate(pt, theta):
    new_x = pt.real * math.cos(theta) - pt.imag * math.sin(theta)
    new_y = pt.real * math.sin(theta) + pt.imag * math.cos(theta)
    return Point(new_x, new_y)

def enc_A(bez, next_state):
    result = np.zeros(NUM_DIM)
    set_vec_state(result, next_state)
    set_d1(result, bez.end - bez.start)
    set_d2(result, bez.c1 - bez.start)
    set_d3(result, bez.c2 - bez.start)
    return result
```

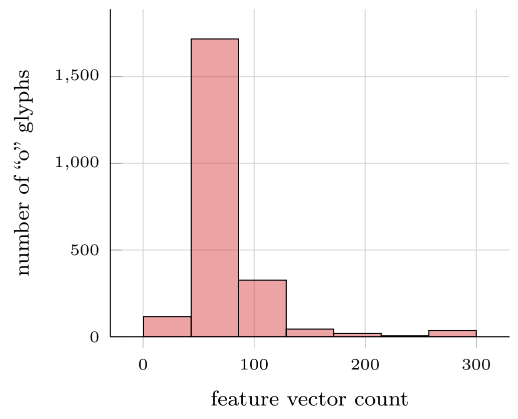
```python
def enc_B(bez, next_state):
    result = np.zeros(NUM_DIM)
    set_vec_state(result, next_state)
    set_d1(result, bez.c1 - bez.start)
    set_d2(result, bez.c2 - bez.c1)
    set_d3(result, bez.end - bez.c2)
    return result

def enc_C(bez, next_state):
    result = np.zeros(NUM_DIM)
    set_vec_state(result, next_state)
    disp = bez.end - bez.start
    theta = math.atan2(disp.y, disp.x)
    set_d1(result, disp)
    set_d2(result, rotate(bez.c1 - bez.start, theta))
    set_d3(result, rotate(bez.end - bez.c2, theta))
    return result

def enc_D(bez, next_state):
    result = np.zeros(NUM_DIM)
    set_vec_state(result, next_state)
    disp = bez.end - bez.start
    theta = math.atan2(disp.y, disp.x)
    set_d1(result, bez.end)
    set_d2(result, rotate(bez.c1 - bez.start, -theta))
    set_d3(result, rotate(bez.end - bez.c2, -theta))
    return result

def enc_E(bez, next_state):
    result = np.zeros(NUM_DIM)
    set_vec_state(result, next_state)
    set_d1(result, bez.end)
    set_d2(result, bez.c1)
    set_d3(result, bez.c2)
    return result
```

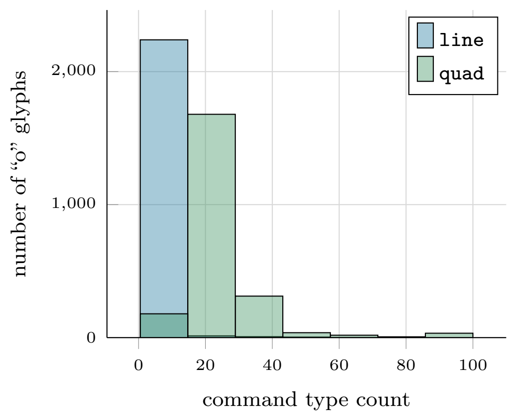
Appendix B

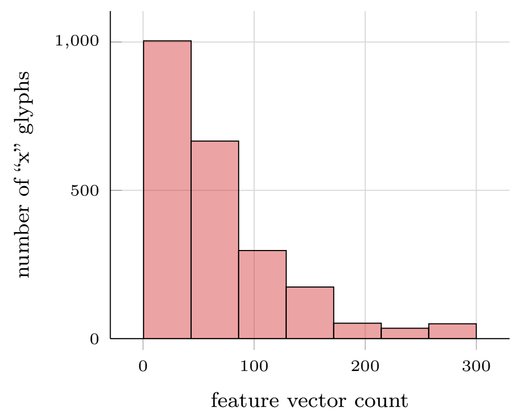
Dataset statistics

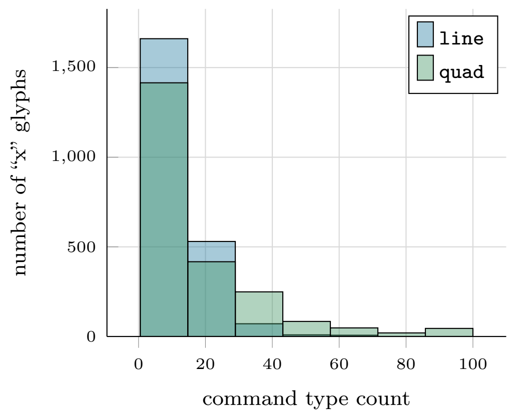
Figure B-1: Dataset statistics for the remaining glyphs on which our model was trained across all 2552 font faces in the Google Fonts dataset.

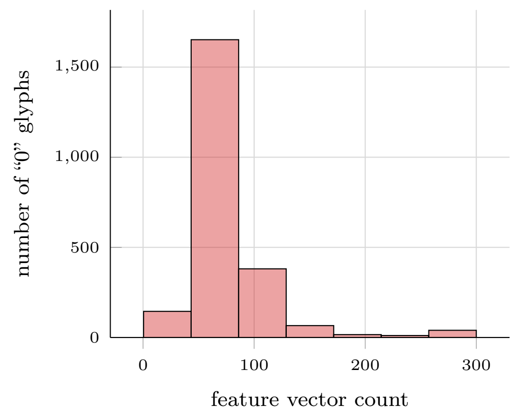


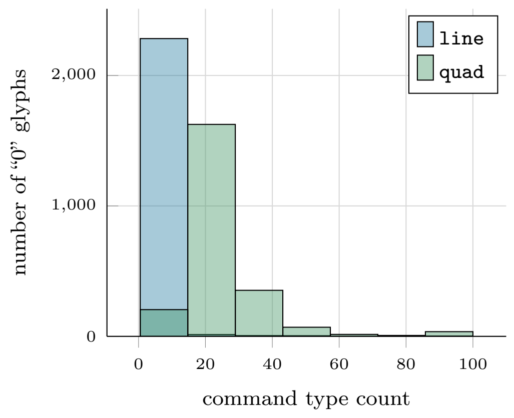

number of “o” glyphs
1,500
1,000
500
0
0
100
200
300
feature vector count

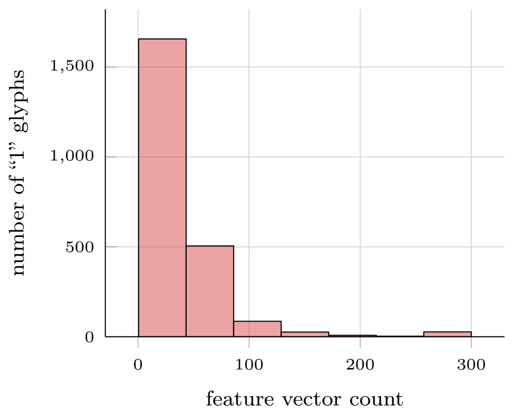

number of “o” glyphs
2,000
1,000
0
0
20
40
60
80
100
command type count
line
quad

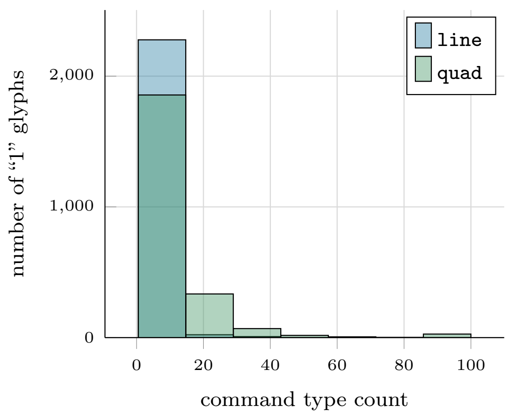

number of “x” glyphs
1,000
500
0
0
100
200
300
feature vector count

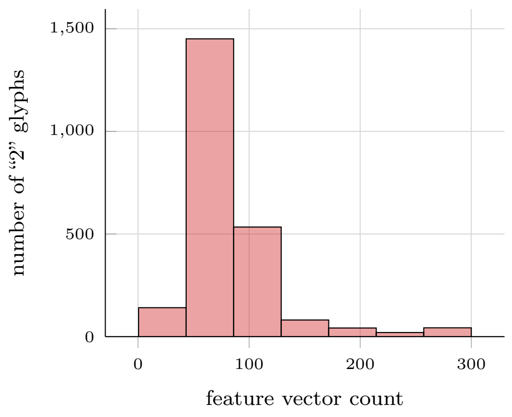

number of “x” glyphs
1,500
1,000
500
0
0
20
40
60
80
100
command type count
line
quad

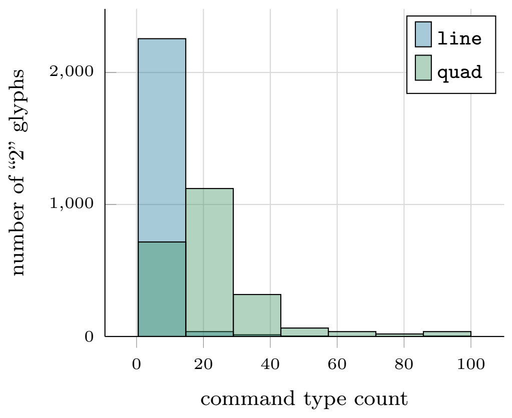

number of “0” glyphs
1,500
1,000
500
0
0
100
200
300
feature vector count

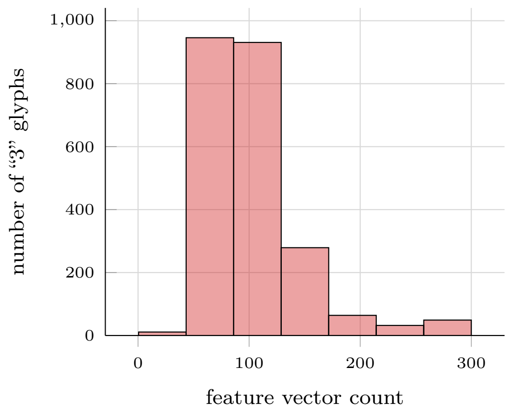

number of “0” glyphs
2,000
1,000
0
0
20
40
60
80
100
command type count
line
quad

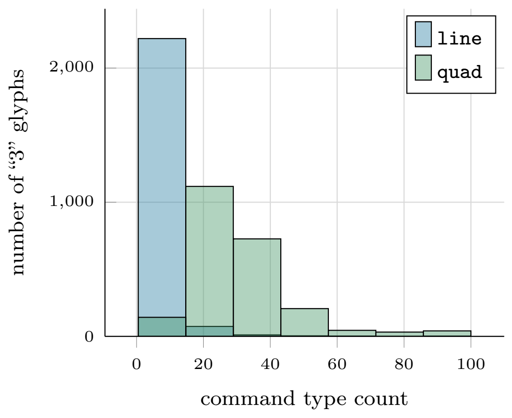

number of “1” glyphs
0
500
1,000
1,500
0
100
200
300
feature vector count

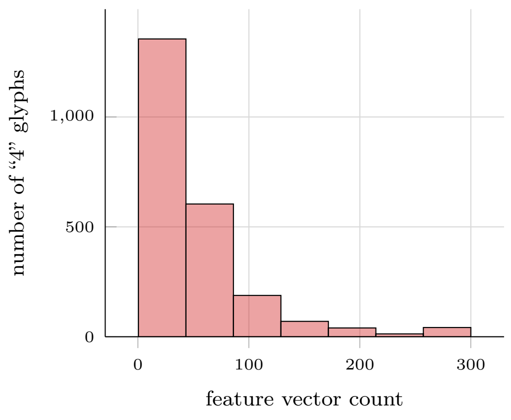

number of “1” glyphs
0
1,000
2,000
0
20
40
60
80
100
command type count
line
quad


number of “2” glyphs
0
500
1,000
1,500
0
100
200
300
feature vector count

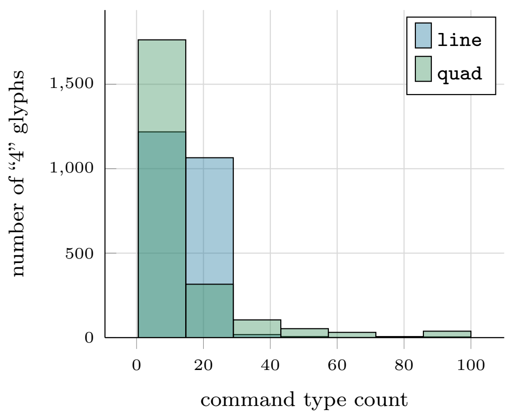

number of “2” glyphs
0
1,000
2,000
0
20
40
60
80
100
command type count
line
quad


number of “3” glyphs
0
200
400
600
800
1,000
0
100
200
300
feature vector count

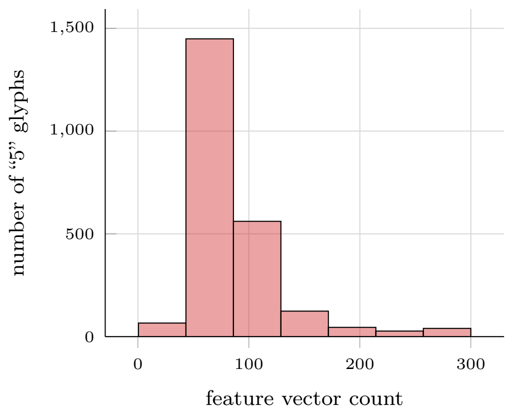

number of “3” glyphs
0
1,000
2,000
0
20
40
60
80
100
command type count
line
quad



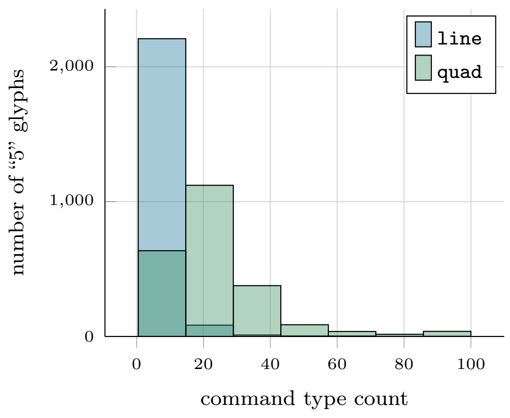
number of "4" glyphs
0
500
1,000
0
100
200
300
feature vector count



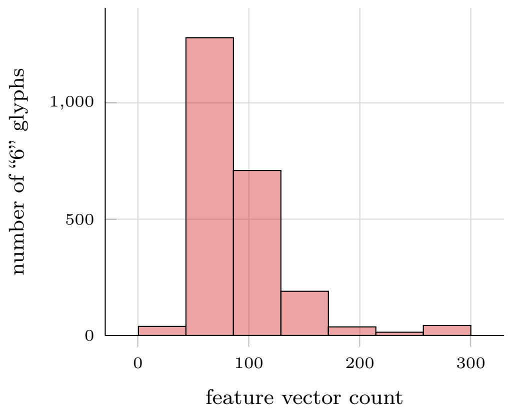
line
quad
number of "4" glyphs
0
500
1,000
1,500
0
20
40
60
80
100
command type count



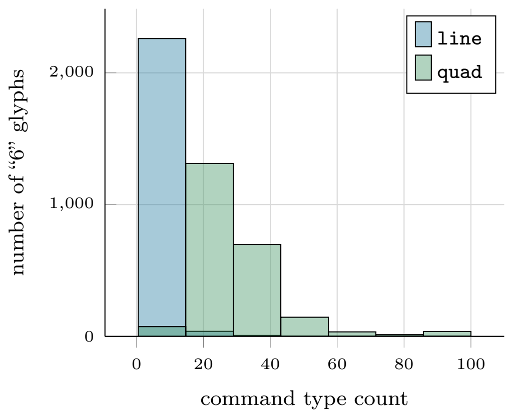
number of "5" glyphs
0
500
1,000
1,500
0
100
200
300
feature vector count



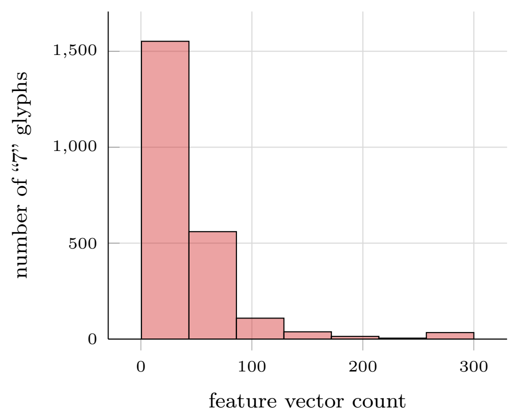
line
quad
number of "5" glyphs
0
1,000
2,000
0
20
40
60
80
100
command type count

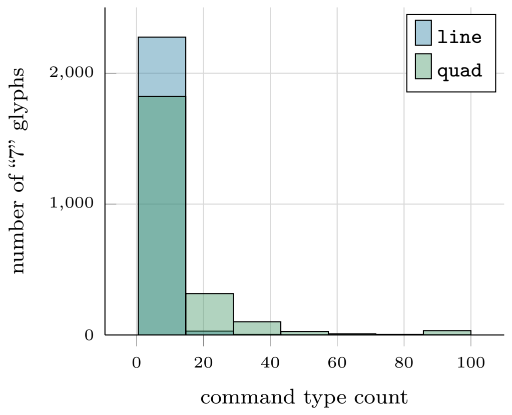


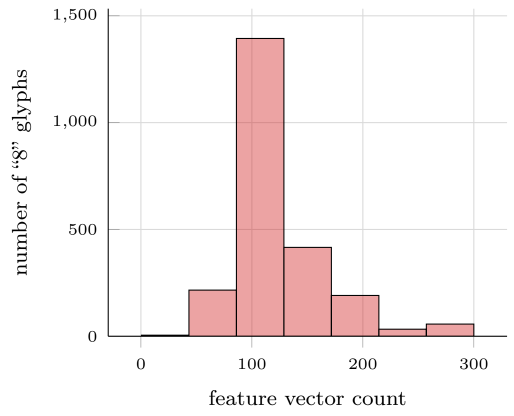
number of "6" glyphs
0
500
1,000
0
100
200
300
feature vector count

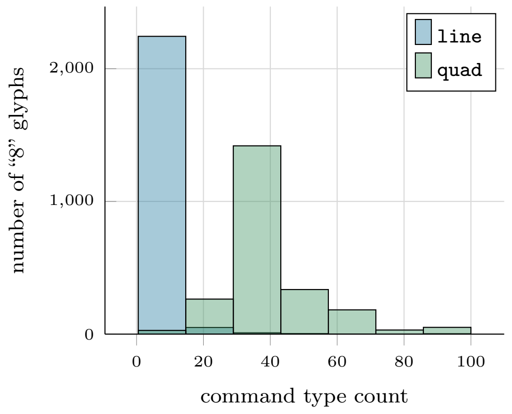


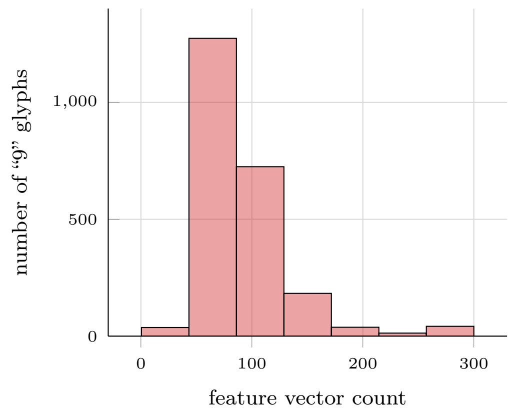
line
quad
number of "6" glyphs
0
1,000
2,000
0
20
40
60
80
100
command type count

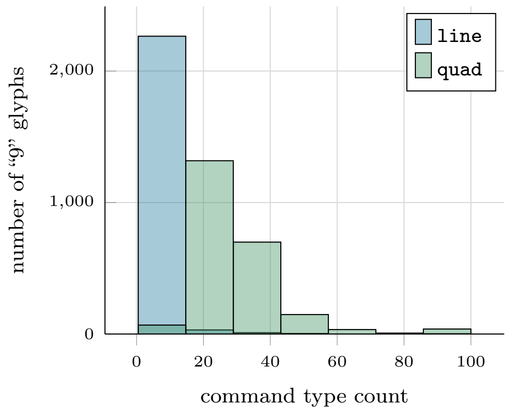

number of “7” glyphs
1,500
1,000
500
0
0
100
200
300
feature vector count


number of “7” glyphs
2,000
1,000
0
0
20
40
60
80
100
command type count
line
quad


number of “8” glyphs
1,500
1,000
500
0
0
100
200
300
feature vector count


number of “8” glyphs
2,000
1,000
0
0
20
40
60
80
100
command type count
line
quad


number of “9” glyphs
1,000
500
0
0
100
200
300
feature vector count


number of “9” glyphs
2,000
1,000
0
0
20
40
60
80
100
command type count
line
quad

Appendix C

Training

Table C.1: Training results and validation costs for single-glyph models described in Chapter 5. Models were run against the validation set every 5k iterations, and the model with the best validation cost is reported. We also report the overall, KL, and reconstruction costs of the selected model for each encoding on the test set.

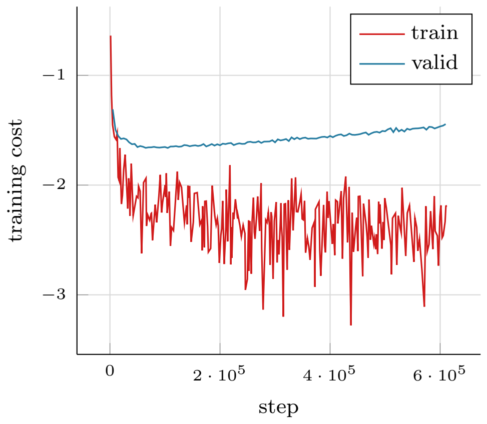
Glyph	Iterations trained	Best model iteration	Test cost	Test KL cost	Test R cost
b	610k	105k	-1.830	0.5585	-1.836
f	573k	205k	-1.669	0.4151	-1.673
g	471k	65k	-2.075	0.7274	-2.082
o	438k	280k	-1.733	0.2448	-1.736
x	566k	60k	-1.055	0.2946	-1.058

Table C.2: Training results and validation costs for models in encoding experiment described in Section 6.1. Each model was trained for 125k iterations on the **b** dataset partitioned into 1920 training, 240 test, and 240 validation examples, with the default parameters described in Chapter 5. Models were run against the validation set every 5k iterations, and the model with the best validation cost is reported and used to evaluate its encoding.

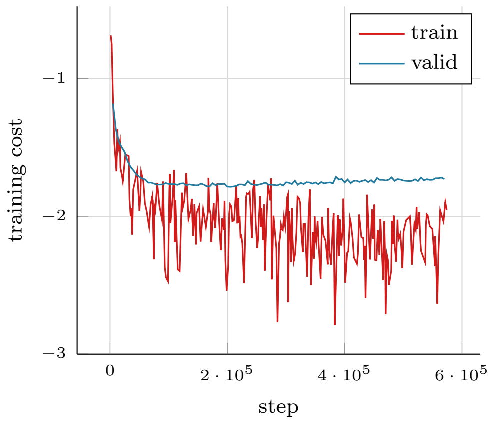
Encoding	Best model iteration	Test cost	Test KL cost	Test R cost
A	80k	-1.428	0.3350	-1.431
B	110k	-1.724	0.5614	-1.729
C	110k	-1.478	0.4631	-1.483
D	115k	-2.182	0.4391	-2.186
E	25k	-1.466	0.5928	-1.472

Figure C-1: Training and validation costs for the models trained in Chapter 5.

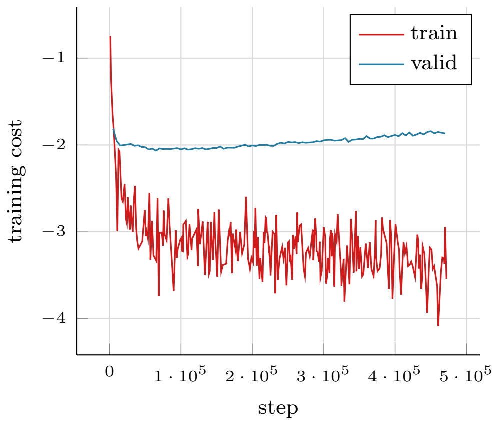
(a) Glyph **b**



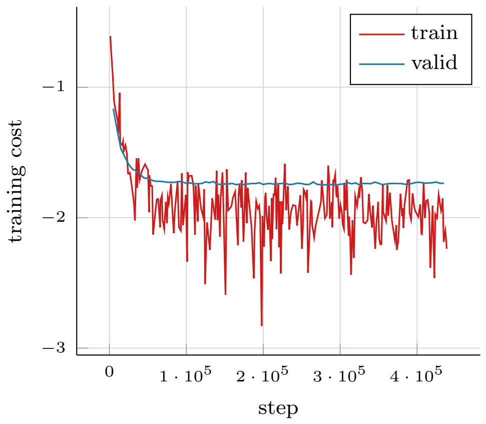
(b) Glyph **f**



(c) Glyph **g**



(d) Glyph **o**



(e) Glyph **x**

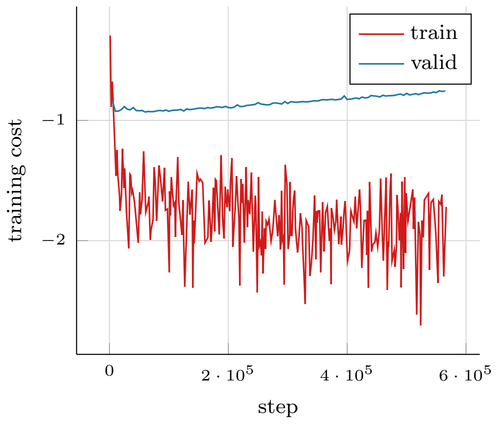
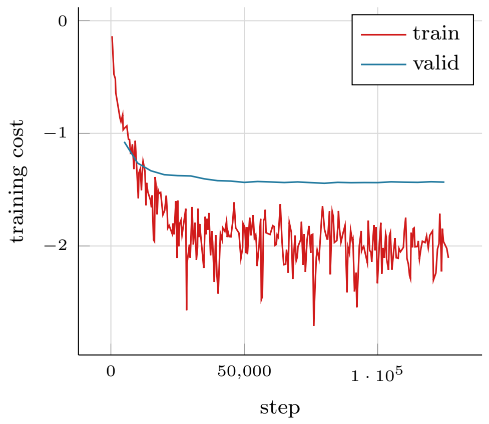
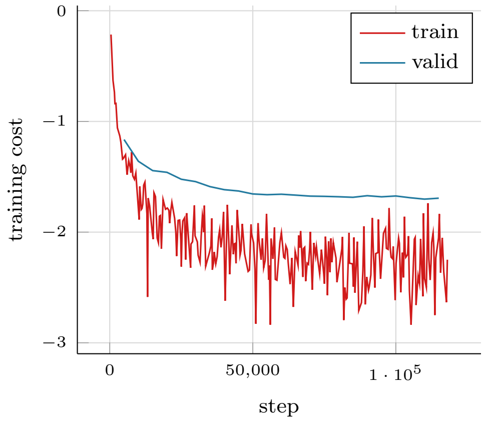


Figure C-2: Training and validation costs for encoding models listed in Table 6.1.

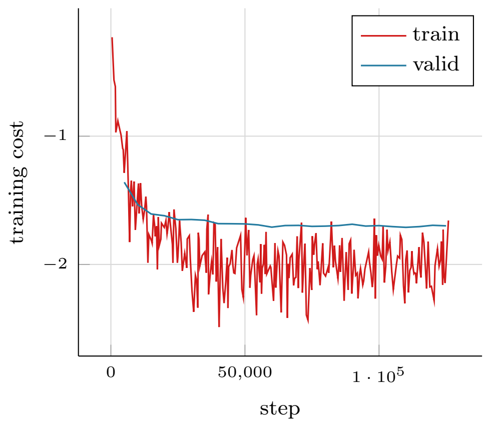
(a) Encoding A



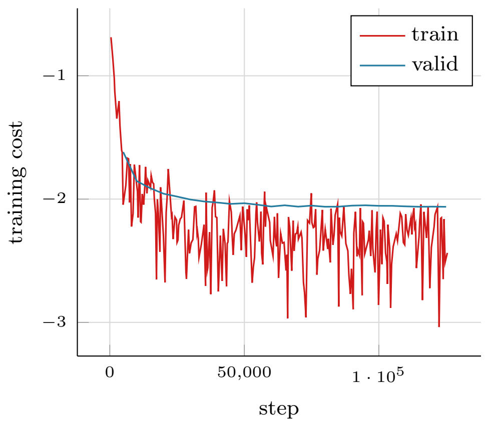
(b) Encoding B



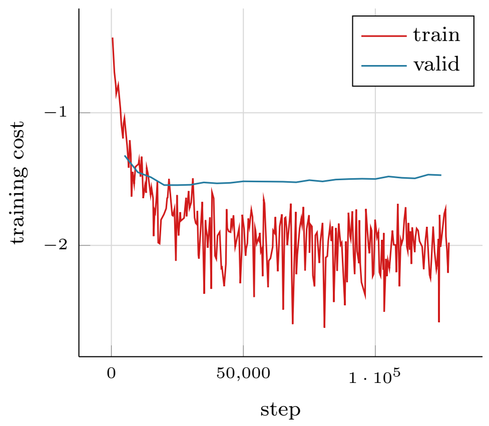
(c) Encoding C



(d) Encoding D



(e) Encoding E



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.