A. Appendix

A.1. Dataset details

We collected a total of $14M$ font characters ($\mu = 226,000$, $\sigma = 22,625$ per class) in a common format (SFD), while retaining only characters whose unicode id corresponded to the classes `0-9`, `a-z`, `A-Z`. Filtering by unicode id is imperfect because many icons intentionally declare an id such that equivalent characters can be rendered in that font style (e.g.: 七 sometimes declares the unicode id normally reserved for `7`).

We then convert the SFD icons into SVG. The SVG format can be composed of many elements (`square`, `circle`, etc). The most expressive of these is the `path` element whose main attribute is a sequence of commands, each requiring a varying number of arguments (`lineTo`: 1 argument, `cubicBezierCurve`: 3 arguments, etc.). An SVG can contain multiple elements but we found that SFD fonts can be modelled with a single `path` and a subset of its commands (`moveTo`, `lineTo`, `cubicBezierCurve`, `EOS`). This motivates our method to model SVGs as a single sequence of commands.

In order to aid learning, we filter out characters with over 50 commands. We also found it crucial to use relative positioning in the arguments of each command. Additionally, we re-scale the arguments of all icons to ensure that most real-values in the dataset will lie in similar ranges. This process preserves size differences between icons. Finally, we standardize the command ordering within a path such that each shape begins and ends at its top-most point and the curves always start by going clockwise. We found that setting this prior was important to remove any ambiguity regarding where the SVG decoder should start drawing from and which direction (information which the image encoder would not be able to provide).

When rendering the SVGs as 64x64 pixel images, we pick a render region that captures tails and descenders that go under the character's baseline. Because relative size differences between fonts are still present, this process is also imperfect: zoom out too little and many characters will still go outside the rendered image, zoom out too much and most characters will be too small for important style differences to be salient.

Lastly, we convert the SVG path into a vector format suitable for training a neural network model: each character is represented by a sequence of commands, each consisting of tuples with: 1) a one-hot encoding of command type (`moveTo`, `lineTo`, etc.) and 2) a normalized representation of the command's arguments (e.g.: x, y positions). Note that for this dataset we only use 4 commands (including EOS), but this representation can easily be extended for any SVG icon that use any or all of the commands in the SVG path language.

A.2. Details of network architecture

Our model is composed of two separate substructures: a convolutional variational autoencoder and an autoregressive SVG decoder. The model was implemented and trained with Tensor2Tensor [58].

The image encoder is composed of a sequence of blocks, each composed of a convolutional layer, conditional instance normalization (CIN) [9, 43], and a ReLU activation. Its output is a z representation of the input image. At training time, z is sampled using the reparameterization trick [30, 49]. At test time, we simply use $z = \mu$. The image decoder is an approximate mirror image of the encoder, with transposed convolutions in place of the convolutions. All convolutional-type layers have `SAME` padding. CIN layers were conditioned on the icon's class.

Operations	Kernel, Stride	Output Dim
Conv-CIN-ReLU	5, 1	64x64x32
Conv-CIN-ReLU	5, 1	32x32x32
Conv-CIN-ReLU	5, 1	32x32x64
Conv-CIN-ReLU	5, 2	16x16x64
Conv-CIN-ReLU	4, 2	8x8x64
Conv-CIN-ReLU	4, 2	4x4x64
Flatten-Dense	-	64

Table 1: Architecture of convolutional image encoder containing $416,672$ parameters.

Operations	Kernel, Stride	Output Dim
Dense-Reshape	-	4x4x64
ConvT-CIN-ReLU	4, 2	8x8x64
ConvT-CIN-ReLU	4, 2	16x16x64
ConvT-CIN-ReLU	5, 1	16x16x64
ConvT-CIN-ReLU	5, 2	32x32x64
ConvT-CIN-ReLU	5, 1	32x32x32
ConvT-CIN-ReLU	5, 2	64x64x32
ConvT-CIN-ReLU	5, 1	64x64x32
Conv-Sigmoid	5, 1	64x64x1

Table 2: Architecture of convolutional image decoder containing $516,865$ parameters.

The SVG decoder consists of 4 stacked LSTMs cells with hidden dimension 1024, trained with feed-forward dropout [53], as well as recurrent dropout [62, 51] at 70% keep-probability. The decoder's topmost layer consists of a Mixture Density Network (MDN) [3, 15]. It's hidden state is initialized by conditioning on z. At each time-step, the LSTM receives as input the previous time-step's sampled MDN output, the character's class and the z representation. The total number of parameters is $34,875,272$.

A.3. Training details

The optimization objective of the image VAE is the log-likelihood reconstruction loss and the KL loss applied to z with KL-beta 4.68. We use Kingma et al's [29] trick with 4.8 free bits (0.15 per dimension of z). The model is trained with batch size 64.

The SVG decoder's loss is composed of a softmax cross-entropy loss between the one-hot command-type encoding, added to the MDN loss applied to the real-valued arguments. We found it useful to scale the softmax cross-entropy loss by 10 when training with this mixed loss. We trained this model with teacher forcing and used batch size 128.

All models are initialized with the method proposed by He et al. (2015) [17] and trained with the Adam optimizer with $\epsilon = 10^{-6}$ [26].

A.4. Visualization details

The dimensionality reduction algorithm used for visualizing the latent space is UMAP [39]. We fit the activations z of 1M examples from the dataset into 2 UMAP components, using the cosine similarity metric, with a minimum distance of 0.5, and 50 nearest neighbors. After fitting, we discretize the 2D space into 50 discrete buckets and decode the average z in each grid cell with the image decoder.

A.5. Samples from generated font sets

Figures 1, 4, 5, 7 and 9 contained selected examples to highlight the successes and failures of the model as well as demonstrate various results. To provide the reader with a more complete understanding of the model performance, below we provide additional samples from the model highlighting successes (Figure 10) and failures (Figure 11). As before, results shown are selected best out of 10 samples.

Figure 10: **More examples of randomly generated fonts**. Details follow figure 3.

Figure 11: **Examples of poorly generated fonts**. Details follow figure 3. Highly stylized characters clustered in a high-variance region of the latent space z. Samples from this region generate poor quality SVG fonts. See Section 4.5 for details.