A Neural Representation of Sketch Drawings

David Ha
Google Brain
hadavid@google.com

Douglas Eck
Google Brain
deck@google.com

Abstract

We present `sketch-rnn`, a recurrent neural network (RNN) able to construct stroke-based drawings of common objects. The model is trained on thousands of crude human-drawn images representing hundreds of classes. We outline a framework for conditional and unconditional sketch generation, and describe new robust training methods for generating coherent sketch drawings in a vector format.

1 Introduction

Recently, there have been major advancements in generative modelling of images using neural networks as a generative tool. Generative Adversarial Networks (GANs) [7], Variational Inference (VI) [19], and Autoregressive (AR) [24] models have become popular tools in this fast growing area. Most of the work, however, has been targeted towards modelling rasterized images represented as a two dimensional grid of pixel values. While these models are currently able to generate realistic, low resolution pixel images, a key challenge for many of these models is to generate images with coherent structure. For example, these models may produce amusing images of cats with three or more eyes, or dogs with multiple heads [7]. In this paper we investigate a much lower-dimensional vector-based representation inspired by how people draw.

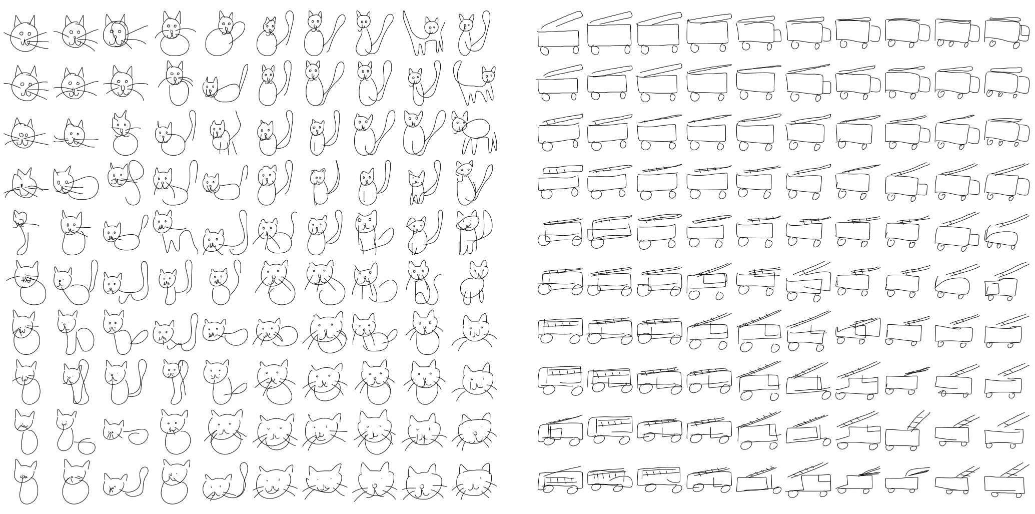
Figure 1: Latent space of cats and firetrucks generated by `sketch-rnn`.

As humans, we do not understand the world as a grid of pixels, but rather develop abstract concepts to represent what we see. From a young age, we develop the ability to communicate what we see by drawing on paper with a pencil or crayon. In this way we learn to represent an image based on a short sequence of strokes. For example, an archetypical child-drawn house – i.e. a triangle on top of a square with door and window added – can be achieved with a few line strokes. Drawings like this may not resemble reality as captured by a photograph, but they do tell us something about how people represent and reconstruct images of the world around them. Objects such as man, woman, eyes, face, cat, dog, etc. can be clearly communicated with simple drawings. Arguably even emotions can be conveyed via line drawings made from only a few strokes. Finally, abstract visual communication is a key part of how humans communicate with each other, with many writing systems composed of symbols that originated from simple drawings known as pictograms.

Our goal is to train machines to learn to draw and generalize abstract concepts in a manner similar to humans. As a first step towards this goal, in this work we train neural networks to learn from a dataset of crude hand drawn sketches. The sketches are represented in a vector format, not as pixels, and are encoded as a list of discrete pen strokes. By training a generative recurrent neural network to model discrete pen strokes, we can observe the limits of its ability to learn a generative representation of an object that is both compact (only a few pen strokes) and in some ways abstract in that these stroke sequences capture many different strategies for drawing the same object. For example, see Figure 1 (Left), where the model learns several very different ways to draw a cat.

This paper makes the following contributions: We outline a framework for both unconditional and conditional generation of vector images composed of a sequence of lines. Our recurrent neural network-based generative model is capable of producing sketches of common objects in a vector format. We develop a training procedure unique to vector images to make the training more robust. In the conditional generation model, we explore the latent space developed by the model to represent a vector image. We demonstrate that enforcing a prior on the latent space helps the model generate more coherent images. We also discuss potential creative applications of our methodology. We are working towards making available a large dataset of simple hand drawings to encourage further development of generative models, and we will release an implementation of our model as an open source project called `sketch-rnn`.

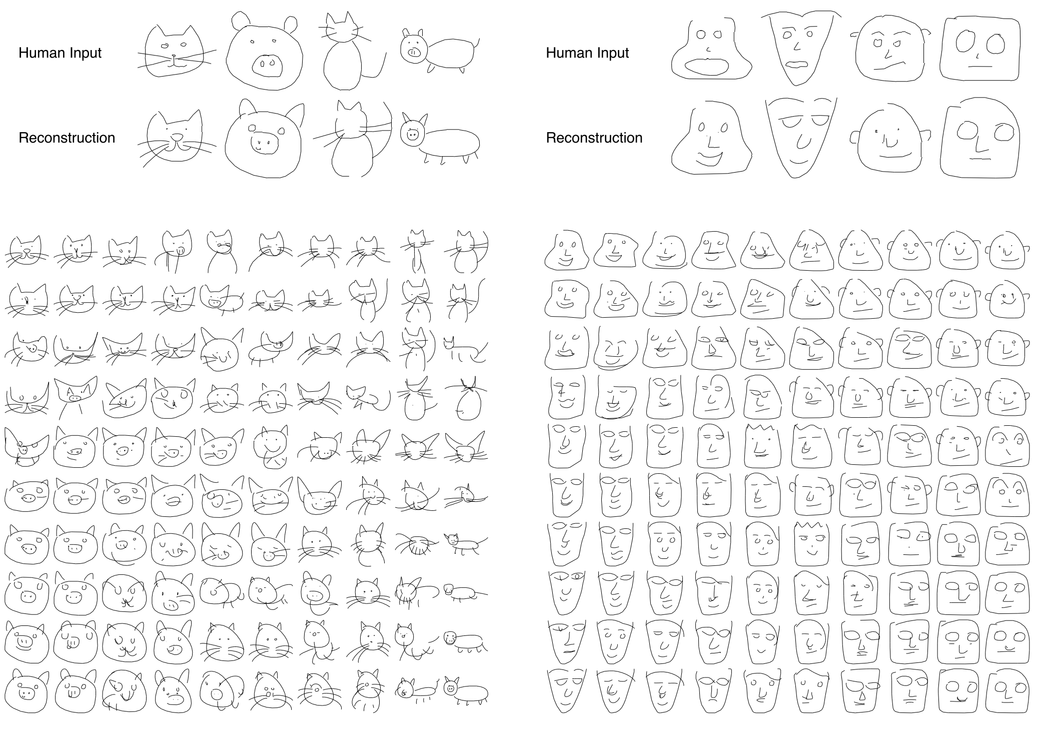

Figure 2: Example input sketches and `sketch-rnn` generated reproductions (Top), Latent space interpolation between the four reproduced sketches (Bottom).

2 Related Work

There is a long history of work related to algorithms that mimic painters. One such work is Portrait Drawing by Paul the Robot [28, 30], where an underlying algorithm controlling a mechanical robot arm sketches lines on a canvas with a programmable artistic style to mimic a given digitized portrait of a person. Reinforcement Learning based-approaches [30] have been developed to discover a set of paint brush strokes that can best represent a given input photograph. These prior works generally attempt to mimic digitized photographs, rather than develop generative models of vector images.

Neural Network-based approaches have been developed for generative models of images, although the majority of neural network-related research on image generation deal with pixel images [7, 13, 15, 18, 24, 29]. There has been relatively little work done on vector image generation using neural networks. Graves' original work on handwriting generation with recurrent neural networks [8] laid the groundwork for utilizing mixture density networks [2] to generate continuous data points. Recent works of this approach attempted to generate vectorized Kanji characters unconditionally [9] and conditionally [31] by modelling Chinese characters as a sequence of pen stroke actions.

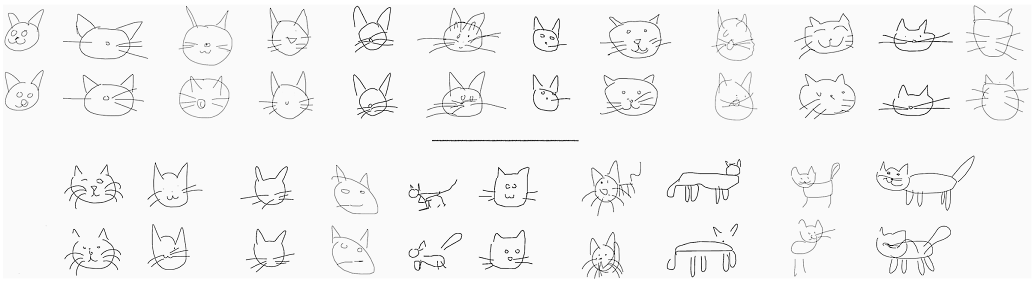
Figure 3: Human sketches of cats, and their generated reconstructions below each human sketch.

In addition to unconditionally generating sketches, we also explore encoding existing sketches into a latent space of embedding vectors. We can then conditionally generate sketches based on such latent vectors, like the examples in Figure 3. Previous work [3] outlined a methodology to combine Sequence-to-Sequence models with a Variational Autoencoder to model natural English sentences in latent vector space. A related work [20], utilizes probabilistic program induction, rather than neural networks, to perform one-shot modelling of the Omniglot dataset containing vector images of simple symbols.

One of the factors limiting research development in the space of generative vector drawings is the lack of publicly available datasets. Previously, the Sketch dataset [6], consisting of 20K vector sketches, was used to explore feature extraction techniques. A subsequent work, the Sketchy dataset [25], provided 70K vector sketches along with corresponding pixel images for various classes. This allowed for a larger-scale exploration of human sketches. ShadowDraw [22] is an interactive system that predicts what a finished drawing looks like based on a set of incomplete brush strokes from the user while the sketch is being drawn. ShadowDraw used a dataset of 30K raster images combined with extracted vectorized features. In this work, we use a much larger dataset of vector sketches that will be made publicly available.

3 Methodology

3.1 Dataset

We constructed a dataset from sketch data obtained from The Quickdraw A.I. Experiment [14], an online demo where the users are asked to draw objects belonging to a particular object class in less than 20 seconds. We have selected 75 classes from the raw dataset to construct the `quickdraw-75` dataset. Each class consists of a training set of 70K samples, in addition to 2.5K samples each for validation and test sets.

In our experiments, we attempt to model the entire dataset, in addition to modelling individual classes or multiple classes selected out of the 75 classes. We are currently looking at ways to release the full version of this dataset to the research community.

3.2 Data Format

```
[ 0,    -5,   1,    0,    0 ]
[ 3,    -23,  1,    0,    0 ]
[ 8,    -14,  1,    0,    0 ]
[ 27,   -36,  1,    0,    0 ]
[ 22,   -19,  1,    0,    0 ]
[ 53,   -29,  1,    0,    0 ]
[ 21,   -5,   1,    0,    0 ]
[ 30,   -1,   1,    0,    0 ]
[ 25,   6,    1,    0,    0 ]
[ 28,   12,   1,    0,    0 ]
[ 33,   29,   1,    0,    0 ]
[ 33,   54,   1,    0,    0 ]
[ 16,   38,   1,    0,    0 ]
[ -85,  0,    1,    0,    0 ]
[ -117, -14,  1,    0,    0 ]
[ -49,  0,    1,    0,    0 ]
[ -56,  10,   0,    1,    0 ]
[ 60,   -80,  1,    0,    0 ]
[ 23,   -4,   1,    0,    0 ]
[ 84,   0,    1,    0,    0 ]
[ 45,   -9,   1,    0,    0 ]
[ 20,   0,    0,    1,    0 ]
[ -177, 42,   1,    0,    0 ]
[ 182,  5,    1,    0,    0 ]
[ 39,   6,    0,    1,    0 ]
[ -160, -77,  1,    0,    0 ]
```

```
[ -19,  55,   1,    0,    0 ]
[ -21,  44,   1,    0,    0 ]
[ 0,    6,    0,    1,    0 ]
[ 86,   -105, 1,    0,    0 ]
[ -2,   113,  0,    1,    0 ]
[ 41,   -117, 1,    0,    0 ]
[ 0,    111,  0,    1,    0 ]
[ -176, -48,  1,    0,    0 ]
[ -35,  0,    1,    0,    0 ]
[ -30,  -7,   1,    0,    0 ]
[ -27,  0,    1,    0,    0 ]
[ -22,  6,    1,    0,    0 ]
[ -5,   10,   1,    0,    0 ]
[ 4,    18,   1,    0,    0 ]
[ 14,   9,    1,    0,    0 ]
[ 26,   6,    1,    0,    0 ]
[ 51,   -2,   1,    0,    0 ]
[ 7,    3,    0,    1,    0 ]
[ -70,  -41,  1,    0,    0 ]
[ -12,  1,    1,    0,    0 ]
[ -3,   4,    1,    0,    0 ]
[ 2,    6,    1,    0,    0 ]
[ 10,   1,    1,    0,    0 ]
[ 9,    -6,   1,    0,    0 ]
[ 0,    -3,   1,    0,    0 ]
[ -8,   -3,   0,    1,    0 ]
```

```
[ -22,  25,   1,    0,    0 ]
[ 11,   -1,   1,    0,    0 ]
[ 12,   -7,   0,    1,    0 ]
[ 91,   36,   1,    0,    0 ]
[ 0,    22,   1,    0,    0 ]
[ 3,    9,    1,    0,    0 ]
[ 6,    8,    1,    0,    0 ]
[ 5,    1,    1,    0,    0 ]
[ 14,   -12,  1,    0,    0 ]
[ 13,   -36,  0,    1,    0 ]
[ 171,  13,   1,    0,    0 ]
[ 4,    30,   1,    0,    0 ]
[ 5,    10,   1,    0,    0 ]
[ 4,    2,    1,    0,    0 ]
[ 6,    -1,   1,    0,    0 ]
[ 6,    -7,   1,    0,    0 ]
[ 6,    -14,  1,    0,    0 ]
[ 2,    -15,  0,    1,    0 ]
[ 17,   -44,  1,    0,    0 ]
[ 21,   0,    1,    0,    0 ]
[ 14,   -6,   1,    0,    0 ]
[ 11,   -17,  1,    0,    0 ]
[ -1,   -9,   1,    0,    0 ]
[ -14,  -1,   1,    0,    0 ]
[ -45,  9,    0,    1,    0 ]
[ 0,    0,    0,    0,    1 ]
```

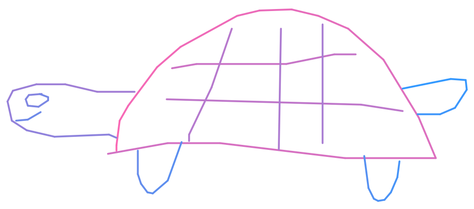

Figure 4: A sample sketch, as a sequence of $(\Delta x, \Delta y, p_1, p_2, p_3)$ points and in rendered form. In the rendered sketch, the line color corresponds to the sequential stroke ordering.

We use a data format that represents a sketch as a set of pen stroke actions. This representation is an extension of the handwriting format used in [8]. Our format extends the binary pen stroke event into a multi-state event. In this data format, the initial absolute coordinate of the drawing is located at the origin.

Figure 4 shows an example. A sketch is a list of points, and each point is a vector consisting of 5 elements: $(\Delta x, \Delta y, p_1, p_2, p_3)$. The first two elements are the offset distance in the x and y directions of the pen from the previous point. The last 3 elements represents a binary one-hot vector of 3 possible states. The first pen state, p_1, indicates that the pen is currently touching the paper, and that a line will be drawn connecting the next point with the current point. The second pen state, p_2, indicates that the pen will be lifted from the paper after the current point, and that no line will be drawn next. The final pen state, p_3, indicates that the drawing has ended, and subsequent points, including the current point, will not be rendered.

3.3 `sketch-rnn`

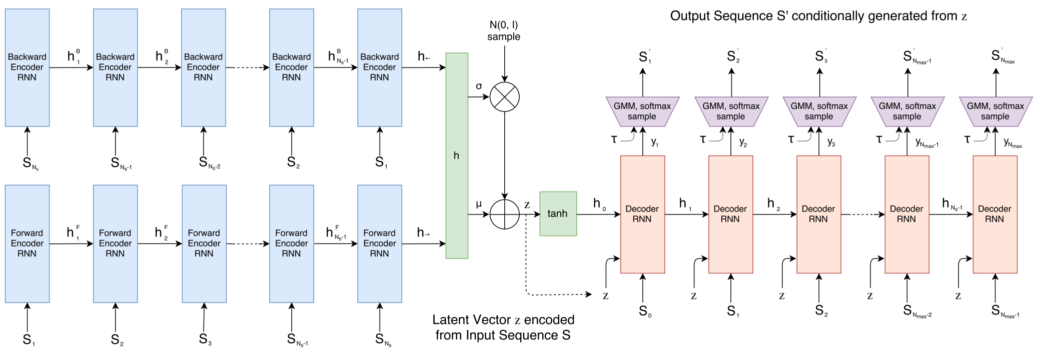

Figure 5: Schematic diagram of `sketch-rnn`.

Our model is a Sequence-to-Sequence Variational Autoencoder (VAE), similar to the architecture described in [3, 19]. Our encoder is a bidirectional RNN [26] that takes in a sketch as an input, and outputs a latent vector of size N_z. Specifically, we feed the sketch sequence, S, and also the same sketch sequence in reverse order, $S_{\texttt{reverse}}$, into two encoding RNNs that make up the bidirectional RNN, to obtain two final hidden states:

$$\begin{aligned} h_{\rightarrow} &= \texttt{encode}_{\rightarrow}(S) \\ h_{\leftarrow} &= \texttt{encode}_{\leftarrow}(S_{\texttt{reverse}}) \\ h &= [\, h_{\rightarrow} \,;\, h_{\leftarrow} \,] \end{aligned} \tag{1}$$

We take this final concatenated hidden state, h, and project it into two vectors μ and $\hat{\sigma}$, each of size N_z, using a fully connected layer. We convert $\hat{\sigma}$ into a non-negative standard deviation parameter σ using an exponential operation. We use μ and σ, along with $\mathcal{N}(0, I)$, a vector of IID Gaussian variables of size N_z, to construct a random vector, $z \in \mathbb{R}^{N_z}$, as in the approach for a Variational Autoencoder [19]:

$$\begin{aligned}
\mu &= W_\mu h + b_\mu \\
\hat{\sigma} &= W_\sigma h + b_\sigma \\
\sigma &= \exp\left(\frac{\hat{\sigma}}{2}\right) \\
z &= \mu + \sigma \odot \mathcal{N}(0, I)
\end{aligned} \tag{2}$$

Under this encoding scheme, the latent vector z is not a deterministic output for a given input sketch, but a random vector conditioned on the input sketch.

Our decoder is an autoregressive RNN that samples output sketches conditional on a given latent vector z. The initial hidden states h_0, and optional cell states c_0 (if applicable) of the decoder RNN is the output of a single layer network:

$$[\ h_0\ ;\ c_0\ ] = \tanh(W_z z + b_z) \tag{3}$$

At each step i of the decoder RNN, we feed the previous point, S_{i-1} and the latent vector z in as a concatenated input x_i. Note that S_0 is defined as $(0, 0, 1, 0, 0)$. The output at each time step are the parameters for a probability distribution of the next data point S_i. In Equation 4, we model $(\Delta x, \Delta y)$ as a Gaussian Mixture Model (GMM) with M normal distributions as in [2, 8], and (q_1, q_2, q_3) as a categorical distribution to model the ground truth data (p_1, p_2, p_3), where $(q_1 + q_2 + q_3 = 1)$ as done in [9] and [31]. Unlike Graves [8], our generated sequence is conditioned from a latent code z sampled from our encoder, which is trained end-to-end alongside the decoder.

$$p(\Delta x, \Delta y) = \sum_{j=1}^{M} \Pi_j\ \mathcal{N}(\Delta x, \Delta y \mid \mu_{x,j}, \mu_{y,j}, \sigma_{x,j}, \sigma_{y,j}, \rho_{xy,j}),\ \text{where}\ \sum_{j=1}^{M} \Pi_j = 1 \tag{4}$$

$\mathcal{N}(x, y|\mu_x, \mu_y, \sigma_x, \sigma_y, \rho_{xy})$ is the probability distribution function for a Bivariate Normal distribution. Each of the M Bivariate Normal distributions consist of five parameters: $(\mu_x, \mu_y, \sigma_x, \sigma_y, \rho_{xy})$, where μ_x and μ_y are the means, σ_x and σ_y are the standard deviations, and ρ_{xy} is the correlation parameter of each Bivariate Normal distribution. An additional vector Π of length M, also a categorical distribution, are the mixture weights of the Gaussian Mixture Model. Hence the size of the output vector y is $5M + M + 3$, which includes the 3 logits needed to generate (q_1, q_2, q_3).

The next hidden state of the RNN, generated with its forward operation, projects into the output vector y_i using a fully-connected layer:

$$\begin{aligned}
x_i &= [\ S_{i-1}\ ;\ z\ ] \\
[\ h_i\ ;\ c_i\ ] &= \texttt{forward}(x_i,\ [\ h_{i-1}\ ;\ c_{i-1}\ ]) \\
y_i &= W_y h_i + b_y,\ \text{where}\ \ y_i \in \mathbb{R}^{6M+3}
\end{aligned} \tag{5}$$

The vector y_i is broken down into the parameters of the probability distribution of the next data point:

$$[\ (\hat{\Pi}_1\ \mu_x\ \mu_y\ \hat{\sigma}_x\ \hat{\sigma}_y\ \hat{\rho}_{xy})_1\ ...\ (\hat{\Pi}_1\ \mu_x\ \mu_y\ \hat{\sigma}_x\ \hat{\sigma}_y\ \hat{\rho}_{xy})_M\ (\hat{q}_1\ \hat{q}_2\ \hat{q}_3)\ ] = y_i \tag{6}$$

As in [8], we apply $\exp$ and $\tanh$ operations to ensure the standard deviation values are non-negative, and that the correlation value is between -1 and 1:

$$\begin{aligned}
\sigma_x &= \exp(\hat{\sigma}_x) \\
\sigma_y &= \exp(\hat{\sigma}_y) \\
\rho_{xy} &= \tanh(\hat{\rho}_{xy})
\end{aligned} \tag{7}$$

The probabilities for the categorical distributions are calculated using the outputs as logit values:

$$
\begin{aligned}
q_k &= \frac{\exp(\hat{q}_k)}{\sum_{j=1}^{3} \exp(\hat{q}_j)}, k \in \{1,\ 2,\ 3\} \\
\Pi_k &= \frac{\exp(\hat{\Pi}_k)}{\sum_{j=1}^{M} \exp(\hat{\Pi}_j)}, k \in \{1,\ \ldots\ ,\ M\}
\end{aligned}
\tag{8}
$$

A key challenge is to train our model to know when to stop drawing. Because the probabilities of the three pen stroke events are highly unbalanced, the model becomes more difficult to train. The probability of a p_1 event is much higher than p_2, and the p_3 event will only happen once per drawing. The approach developed in [9] and later followed by [31] was to use different weightings for each pen event when calculating the losses, such as a hand-tuned weighting of $(1, 10, 100)$. We find this approach to be inelegant and inadequate for our dataset of diverse image classes.

We develop a simpler, more robust approach that works well for a broad class of sketch drawing data. In our approach, all sequences are generated to a length of $N_{\max}$ where $N_{\max}$ is the length of the longest sketch in our training dataset. In principle $N_{\max}$ can be considered a hyper parameter. As the length of S is usually shorter than $N_{\max}$, we set S_i to be $(0, 0, 0, 0, 1)$ for $i > N_s$. We discuss the training in detail in the next section.

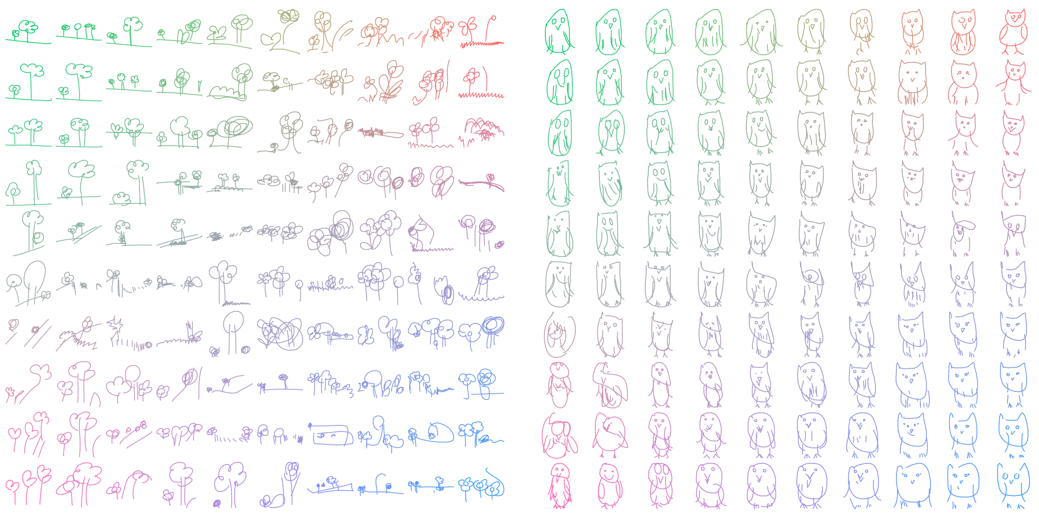
Figure 6: Latent space interpolation between four generated gardens and owls. We also interpolate between four distinct colors for visual effect.

After training, we can sample sketches from our model. During the sampling process, we generate the parameters for both GMM and categorical distributions at each time step, and sample an outcome S'_i for that time step. Unlike the training process, we feed the sampled outcome S'_i, rather than the ground truth S_i, as the input for the next time step. We continue to sample until $p_3 = 1$, or when we have reached $i = N_{\max}$. Like the encoder, the sampled output is not deterministic, but a random sequence, conditioned on the input latent vector z. We can control the level of randomness we would like our samples to have during the sampling process by introducing a temperature parameter τ:

$$
\begin{aligned}
q_k &= \frac{\exp(\frac{\hat{q}_k}{\tau})}{\sum_{j=1}^{3} \exp(\frac{\hat{q}_j}{\tau})}, j \in \{1,\ 2,\ 3\} \\
\Pi_k &= \frac{\exp(\frac{\hat{\Pi}_k}{\tau})}{\sum_{j=1}^{M} \exp(\frac{\hat{\Pi}_j}{\tau})}, k \in \{1,\ \ldots,\ M\} \\
\sigma_x^2 &\rightarrow \sigma_x^2 \tau \\
\sigma_y^2 &\rightarrow \sigma_y^2 \tau
\end{aligned}
\tag{9}
$$

We can scale the softmax parameters of the categorial distribution and also the σ parameters of the Bivariate Normal distribution by a temperature parameter τ, to control the level of randomness in our samples. τ is typically set between 0 and 1. In the limiting case as $\tau \rightarrow 0$, our model becomes deterministic and samples will consist of the most likely point in the probability density function. Figure 7 illustrates of effect of sampling sketches with various temperature parameters.

3.4 Unconditional Generation

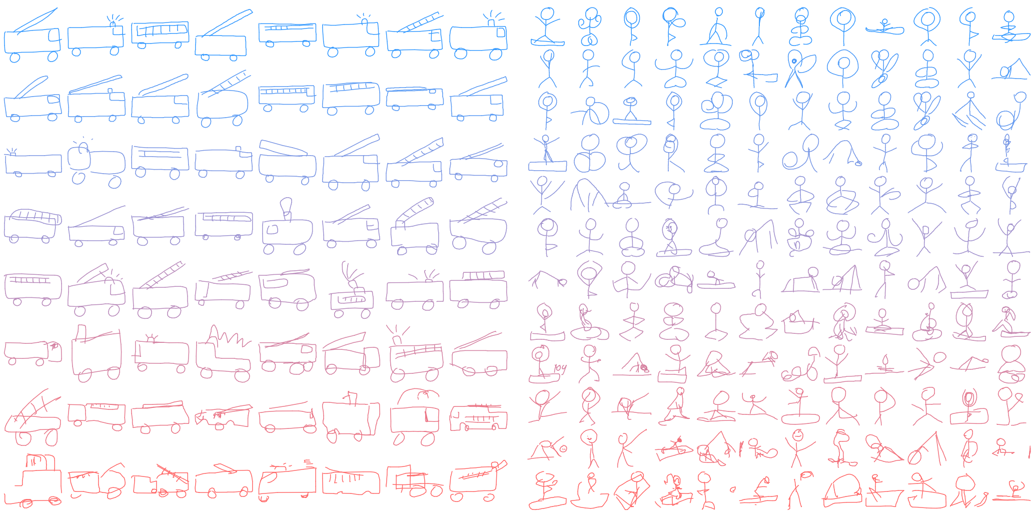

Figure 7: Unconditional sketch generation of firetrucks and yoga positions with varying τ.

As a special case, we can also train our model to generate sketches unconditionally, where we only train the decoder RNN module, without any input or latent vectors. By removing the encoder, the decoder RNN as a standalone model is an autoregressive model without latent variables. In this use case, the initial hidden states and cell states of the decoder RNN are initialized to zero. The inputs x_i of the decoder RNN at each time step is only S_{i-1} or S'_{i-1}, as we do not need to concatenate a latent vector z. In Figure 7, we sample various sketch images generated unconditionally by varying the temperature parameter from $\tau = 0.2$ at the top in blue, to $\tau = 0.9$ at the bottom in red.

3.5 Training

Our training procedure follows the approach of the Variational Autoencoder [19], where the loss function is the sum of two terms: the Reconstruction Loss and the Kullback-Leibler divergence loss. We train our model to optimize this two-part loss function. The Reconstruction Loss term, described in Equation 10, maximizes the log-likehood of the generated probability distribution to explain the training data S. We can calculate this reconstruction loss, L_R, using the generated parameters of the pdf and the training data S. L_R is composed of the sum of the log loss of the offset terms $(\Delta x, \Delta y)$, denoted as L_s, and the log loss of the pen state terms (p_1, p_2, p_3):

$$\begin{aligned}
L_s &= -\frac{1}{N_{\max}} \sum_{i=1}^{N_s} \log\Big(\sum_{j=1}^{M} \Pi_{j,i}\, \mathcal{N}(\Delta x_i, \Delta y_i \mid \mu_{x,j,i}, \mu_{y,j,i}, \sigma_{x,j,i}, \sigma_{y,j,i}, \rho_{xy,j,i}) \Big) \\
L_p &= -\frac{1}{N_{\max}} \sum_{i=1}^{N_{\max}} \sum_{k=1}^{3} p_{k,i} \log(q_{k,i}) \qquad (10) \\
L_R &= L_s + L_p
\end{aligned}$$

Note that we discard the pdf parameters modelling the $(\Delta x, \Delta y)$ points beyond N_s when calculating L_s, while L_p is calculated using all of the pdf parameters modelling the (p_1, p_2, p_3) points until $N_{\max}$. Both terms are normalized by the total sequence length $N_{\max}$. We found this methodology of loss calculation to be more robust and allows the model to easily learn when it should stop drawing, unlike the earlier mentioned method of assigning importance weightings to p_1, p_2, and p_3.

The Kullback-Leibler (KL) divergence loss term measures the difference between the distribution of our latent vector z, to that of an IID Gaussian vector with zero mean and unit variance. Optimizing for this loss term allows us to minimize this difference. We use the result in [19], and calculate the KL loss term, L_{KL}, normalized by number of dimensions N_z of the latent vector:

$$L_{KL} = -\frac{1}{2N_z}\Big(1 + \hat{\sigma} - \mu^2 - \exp(\hat{\sigma})\Big) \qquad (11)$$

The loss function in Equation 12 is a weighted sum of the reconstruction loss term L_R and the KL loss term L_{KL}:

$$Loss = L_R + w_{KL} L_{KL} \qquad (12)$$

There is a tradeoff between optimizing for one term over the other. As $w_{KL} \to 0$, our model approaches a pure autoencoder, sacrificing the ability to enforce a prior over our latent space while obtaining better reconstruction loss metrics. Note that for unconditional generation, where our model is the standalone decoder, there will not be a L_{KL} term as we only optimize for L_R.

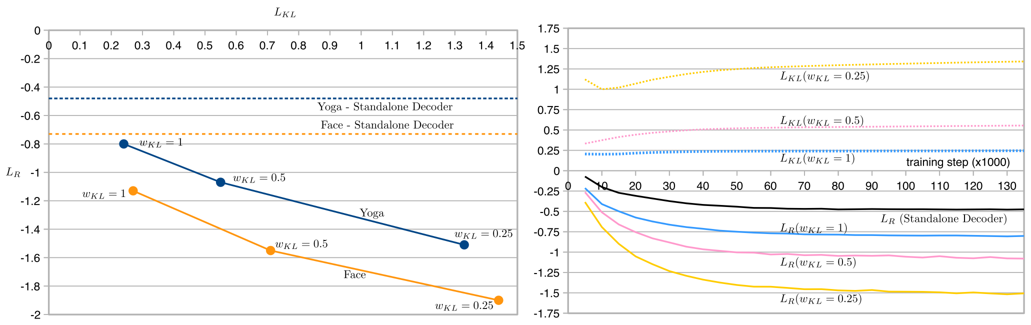

Figure 8: Tradeoff between L_R and L_{KL}, for two models trained on single class datasets (Left). Validation Loss Graph for models trained on the Yoga dataset using various w_{KL}. (Right)

Figure 8 illustrates the tradeoff between different settings of w_{KL} and the resulting L_R and L_{KL} metrics on the test set, along with the L_R metric on a standalone decoder RNN for comparison. As the unconditional model does not receive any prior information about the entire sketch it needs to generate, the L_R metric for the standalone decoder model serves as an upper bound for various conditional models using a latent vector. In our experiments, we explore more closely the qualitatively implications of this tradeoff for sketch generation.

While the loss function in Equation 12 can be used during training, we find that annealing the KL term in the loss function (Equation 13) produced better results. This modification is only used for model training, and the original loss function in Equation 12 is still used to evaluate validation and test sets, and for early stopping.

$$\begin{aligned} \eta_{\texttt{step}} &= 1 - (1 - \eta_{\texttt{min}})R^{\texttt{step}} \\ Loss_{\texttt{train}} &= L_R + w_{KL}\ \eta_{\texttt{step}}\ \max(L_{KL}, KL_{\texttt{min}}) \end{aligned} \tag{13}$$

We find that annealing the KL Loss term generally results in better losses. Annealing the L_{KL} term in the loss function directs the optimizer to first focus more on the reconstruction term in Equation 10, which is the more difficult loss term of the model to optimize for, before having to deal with optimizing for the KL loss term in Equation 11, a far simpler expression in comparison. This approach has been used in [3, 15, 18]. Our annealing term $\eta_{\texttt{step}}$ starts at $\eta_{\texttt{min}}$ (typically 0 or 0.01) at training step 0, and converges to 1 for large training steps. R is a term close to, but less than 1.

If the distribution of z is close enough to $\mathcal{N}(0, I)$, we can sample sketches from the decoder using randomly sampled z from $\mathcal{N}(0, I)$ as the input. In practice, we find that going from a larger L_{KL} value ($L_{KL} > 1.0$) to a smaller L_{KL} value of ~ 0.3 generally results in a substantial increase in the quality of sampled images using randomly sampled $z \sim \mathcal{N}(0, I)$. However, going from $L_{KL} = 0.3$ to L_{KL} values closer to zero does not lead to any further noticeable improvements. Hence we find it useful to put a floor on L_{KL} in the loss function by enforcing $\max(L_{KL}, KL_{\texttt{min}})$ in Equation 13.

The $KL_{\texttt{min}}$ term inside the $\max$ operator is typically set to a small value such as 0.10 to 0.50. This term will encourage the optimizer to put less focus on optimizing for the KL Loss term L_{KL} once it is low enough, so we can obtain better metrics for the reconstruction loss term L_R. This approach is similar to the approach described in [18] as *free bits*, where they apply the $\max$ operator separately inside each dimension of the latent vector z.

4 Experiments

We conduct several experiments with `sketch-rnn` for both conditional and unconditional vector image generation. In our experiments, we train our model either on an individual image class, multiple image classes, or all 75 image classes. The class information is not part of the input into the model. We train the models on various settings for w_{KL} and record the breakdown of losses.

The `sketch-rnn` model treats the RNN cell as an abstract component. In our experiments, we use the Long Short-Term Memory (LSTM) [11], as the encoder RNN. For the decoder RNN, we use the HyperLSTM, as this type of RNN cell excels at sequence generation tasks [10]. The ability for HyperLSTM to spontaneously augment its own weights enables it to adapt to many different regimes in a large diverse dataset. Our encoder and decoder RNNs consist of 512 and 2048 nodes respectively.

In our model, we use $M = 20$ mixture components for the decoder RNN. The latent vector z has $N_z = 128$ dimensions. We apply Layer Normalization [1] to our model, and during training apply recurrent dropout [27] with a keep probability of 90%. We train the model with batch sizes of 100 samples, using Adam [17] with a learning rate of 0.0001 and gradient clipping of 1.0. All models are trained with $KL_{\texttt{min}} = 0.20$, $R = 0.99999$. During training, we perform simple data augmentation by multiplying the offset columns $(\Delta x, \Delta y)$ by two IID random factors chosen uniformly between 0.90 and 1.10. Unless mentioned otherwise, all experiments are conducted with $w_{KL} = 1.00$.

4.1 Experimental Results

In our experiments, we train `sketch-rnn` on datasets consisting of individual classes. To experiment with a diverse set of image classes with varying complexities, we select the cat, pig, face, firetruck, garden, owl, mermaid, mosquito, and yoga classes out of the 75 classes. We also construct datasets consisting of multiple classes, such as (cat, pig), and (crab, face, pig, rabbit), in addition to the dataset with all 75 classes. The results for test set evaluation on various datasets are displayed in Table 1.

The relative loss numbers are consistent with our expectations. We see that the reconstruction loss term L_R decreases as we relax the w_{KL} parameter controlling the weight for the KL Loss term, and meanwhile the KL Loss term L_R increases as a result. The reconstruction loss term for the conditional model is strictly less than the unconditional, standalone decoder model.

Dataset	$w_{KL} = 1.00$		$w_{KL} = 0.50$		$w_{KL} = 0.25$		Decoder Only
	L_R	L_{KL}	L_R	L_{KL}	L_R	L_{KL}	L_R
cat	-0.98	0.29	-1.33	0.70	-1.46	1.01	-0.57
pig	-1.14	0.22	-1.37	0.49	-1.52	0.80	-0.82
cat, pig	-1.02	0.22	-1.24	0.49	-1.50	0.98	-0.75
crab, face, pig, rabbit	-0.91	0.22	-1.04	0.40	-1.47	1.17	-0.67
face	-1.13	0.27	-1.55	0.71	-1.90	1.44	-0.73
firetruck	-1.24	0.22	-1.26	0.24	-1.78	1.10	-0.90
garden	-0.79	0.20	-0.81	0.25	-0.99	0.54	-0.62
owl	-0.93	0.20	-1.03	0.34	-1.29	0.77	-0.66
mermaid	-0.64	0.23	-0.91	0.54	-1.30	1.22	-0.37
mosquito	-0.67	0.30	-1.02	0.66	-1.41	1.54	-0.34
yoga	-0.80	0.24	-1.07	0.55	-1.51	1.33	-0.48
everything	-0.61	0.21	-0.79	0.46	-1.02	0.98	-0.41

Table 1: Loss figures (L_R and L_{KL}) for various w_{KL} settings.

More difficult classes, such as (mermaid, mosquito, everything), generally have higher reconstruction loss numbers compared to the easier classes, such as (pig, face, firetruck). However, these numbers can only be used approximately to compare different classes, as the maximum number of strokes parameter $N_{\max}$ is different for the various datasets, which will affect the definition of L_R in our framework. For instance, the pig dataset has $N_{\max} = 182$, while the everything dataset has $N_{\max} = 200$.

In Figure 8 (Right), we plot validation-set loss graphs for on the yoga class for models with various w_{KL} settings. As L_R decreases, the L_{KL} term tends to increase due to the tradeoff between reconstruction loss L_R and KL loss L_{KL}.

4.2 Conditional generation

4.2.1 Conditional Reconstruction

In addition to the loss figures in Table 1, we qualitatively assess the reconstructed sketch S' given an input sketch S. In Figure 9, we sample several reconstructions at various levels of temperature τ using a model trained on the single cat class, starting at 0.01 on the left and linearly increasing to 1.0 on the right. The reconstructed cat sketches have similar properties to the input image, and occasionally add or remove details such as a whisker, a mouth, a nose, or the orientation of the tail.

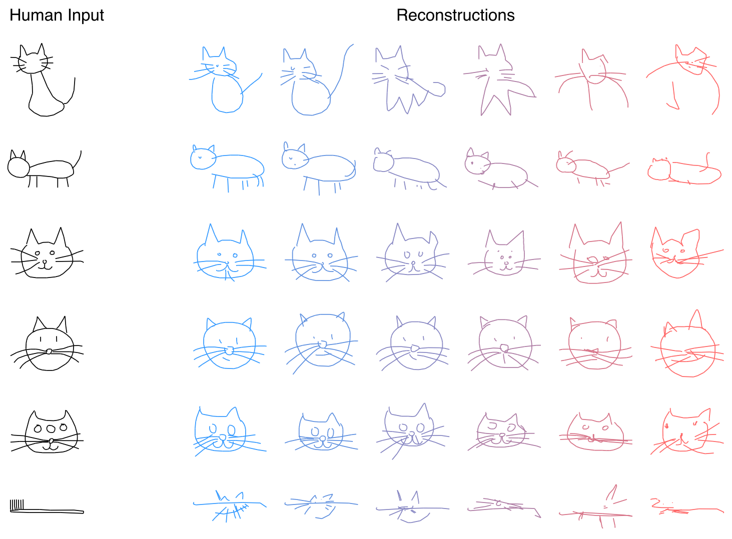

Figure 9: Conditional generation of cats.

When presented with a non-standard image of a cat, such as a cat's face with three eyes, the reconstructed cat only has two eyes. If we input a sketch from another image class, such a toothbrush, the model seemingly generate sketches with similar orientation and properties as the toothbrush input image, but with some cat-like features such as cat ears, whiskers or feet.

We perform a similar experiment with a model trained on the pig class in Figure 10. We see that the model is capable of generating pig faces or entire pigs. However, when presented with a sketch of a pig with many more legs than a normal pig, the reconstructed images generally look like a similar pig with two to four legs. The reconstructions of the eight-legged pig seems to contain extra artifacts on the pig's body or a pair of eyes that was not in the original drawing, as the decoder RNN may be trying to mimic the input image's stroke count. When the input is an image of a truck, the generated drawing looks like a pig with similar features as the truck.

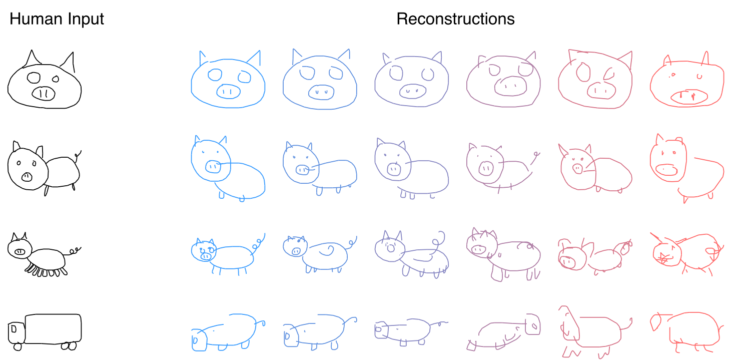

Figure 10: Conditional generation of pigs.

4.2.2 Latent Space Interpolation

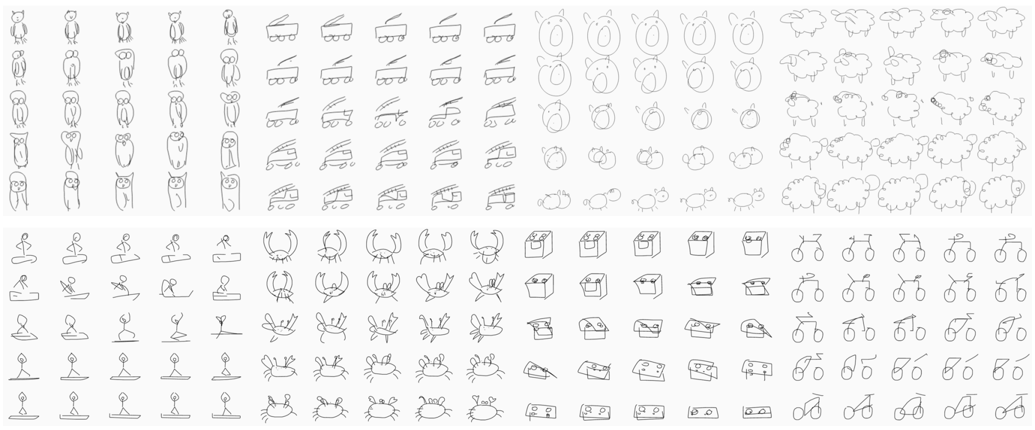

Figure 11: Example of conditional generated sketches with single class models. Latent space interpolation from left to right, and then top to bottom.

As we enforce a Gaussian prior on the latent space, we expect fewer *gaps* in the space between two encoded latent vectors. By interpolating between latent vectors, we can visualize how one image morphs into another image by visualizing the reconstructions of the interpolations. We see some examples of interpolating within models trained on single-class datasets in Figure 11.

To demonstrate the latent space interpolation effect, we trained various models on a dataset consisting of both the cat and pig classes, and we used two images from the test set as inputs - a cat face and a full pig. In Figure 12, we demonstrate the reconstructed images of the model with various settings of w_{KL}, and also reconstruct the spherically interpolated [29] latent vectors between the two encoded latent vectors.

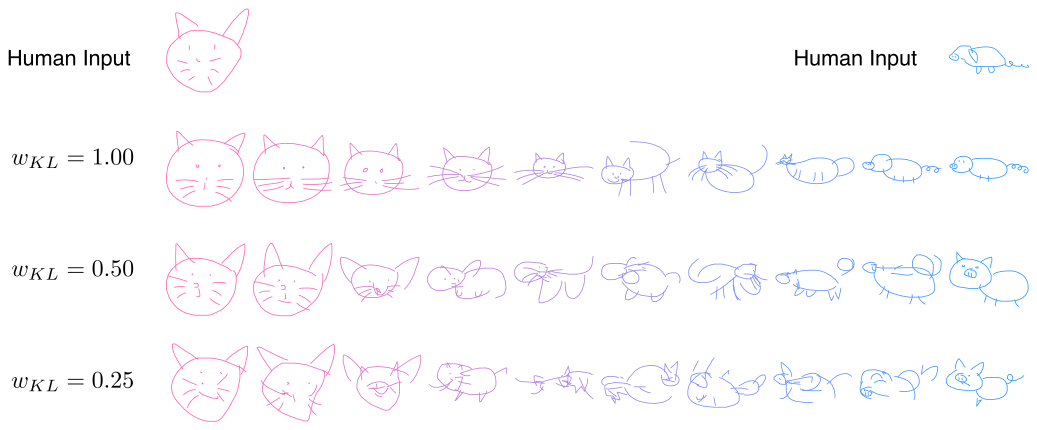

Figure 12: Latent space interpolation between cat and pig using models with various w_{KL} settings.

Models trained with higher w_{KL} settings, resulting in lower L_{KL} losses, appear to produce more meaningful interpolated latent vectors. The reconstructions of the interpolated latent space from models trained with lower w_{KL} settings are less meaningful in comparison, and the reconstructed images are arguably no better, despite having a much lower L_R.

We would like to question the relative importance of the reconstruction loss term L_R, relative to the KL loss term L_{KL}, when our goal is to produce higher quality image reconstructions.

4.2.3 Which Loss Controls Image Coherency?

While our reconstruction loss term L_R optimizes for the log-likelihood of the set of strokes that make up a sketch, this metric alone does not give us any guarantee that a model with a lower L_R number will produce higher quality reconstructions compared to a model with a higher L_R number.

For example, imagine a simple sketch of an face, ☺, where most of the data points of S are be used to represent the head, and only a minority of points represent facial features such as the eyes and mouth. It is possible to reconstruct the face with incoherent facial features, and yet still score a lower L_R number compared to another reconstruction with a coherent and similar face, if the edges around the incoherent face are generated more precisely.

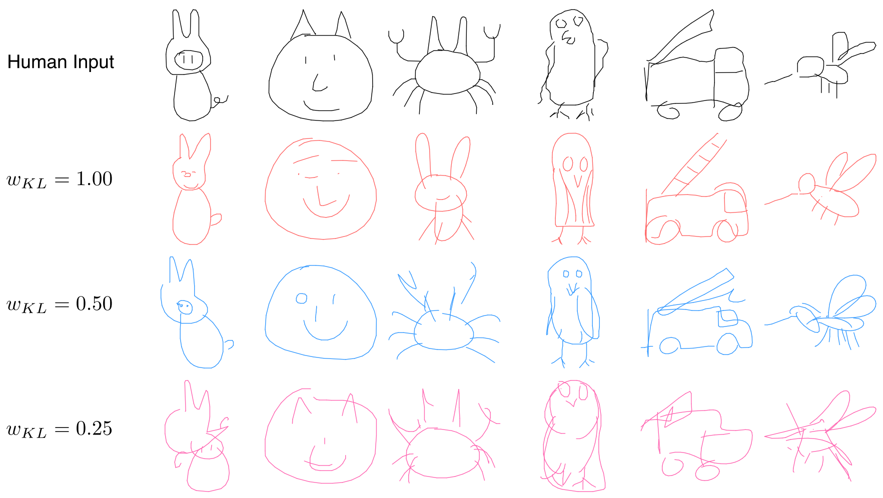

Figure 13: Reconstructions of sketch images using models with various w_{KL} settings.

In Figure 13, we compare the reconstructed images generated using models trained with various w_{KL} settings. In the first three examples from the left, we train our model on a dataset consisting of four image classes (crab, face, pig, rabbit). We deliberately sketch input drawings that contain features of two classes, such as a rabbit with a pig mouth and pig tail, a person with animal ears, and a rabbit with crab claws. We see that the model trained using higher w_{KL} weights, tend to generate sketches with features of a single class that look more coherent, despite having lower L_{KL} numbers. For instance, the model with $w_{KL} = 1.00$ omit pig features, animal ears, and crab claws from its reconstructions. In contrast, the model with $w_{KL} = 0.25$, with higher L_{KL}, but lower L_R numbers tries to keep both inconsistent features, while generating sketches that look less coherent.

In the last three examples in Figure 13, we repeat the experiment on models trained on single-class images, and see similar results even when we deliberately choose input samples from the test set with noisier lines.

As we saw from Figure 12, models with better KL loss terms also generate more meaningful reconstructions from the interpolated space between two latent vectors. This suggests the latent vector for models with lower L_{KL} control more meaningful parts of the drawings, such as controlling whether the sketch is an animal head only or a full animal with a body, or whether to draw a cat head or a pig head. Altering such latent vectors can allow us to directly manipulate these animal features. Conversely, altering the latent codes of models with higher L_{KL} results in scattered movement of individual line segments, rather than alterations of meaningful conceptual features of the animal.

This result is consistent with incoherent reconstructions seen in Figure 13. With a lower L_{KL}, the model is likely to generate coherent images given any random z. Even with a non-standard, or noisy, input image, the model will still encode a z that produces coherent images. For models with lower L_{KL} numbers, the encoded latent vectors contain conceptual features belonging to the input image, while for models with higher L_{KL} numbers, the latent vectors merely encode information about specific line segments. This observation suggests that when using `sketch-rnn` on a new dataset, we should first try different w_{KL} settings to evaluate the tradeoff between L_R and L_{KL}, and then choose a setting for w_{KL} (and $KL_{\texttt{min}}$) that best suit our requirements.

4.2.4 Sketch Drawing Analogies

The interpolation example in Figure 12 suggests the latent vector z encode conceptual features of a sketch. Can we use these features to augment other sketches without such features – for example, adding a body to a cat's head? Indeed, we find that sketch drawing analogies are possible for models trained with low L_{KL} numbers. Given the smoothness of the latent space, where any interpolated vector between two latent vectors results in a coherent sketch, we can perform vector arithmetic using the latent vectors encoded from different sketches and explore how the model organizes the latent space to represent different concepts in the manifold of generated sketches.

As an experiment, we train a model on a dataset of cat and pig classes, and encode sketches of cat heads, pig heads, full cats, full pigs into latent vectors z. We then perform vector arithmetic on $z_{\texttt{cat head}}$, $z_{\texttt{pig head}}$, $z_{\texttt{full cat}}$, and $z_{\texttt{full pig}}$.

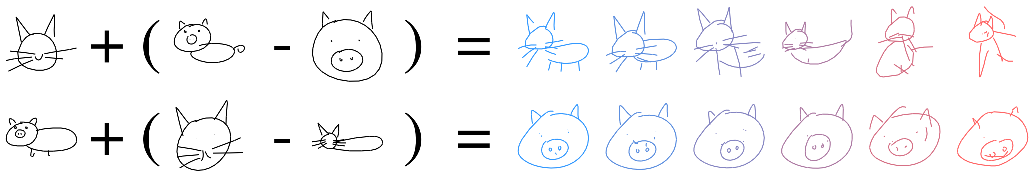

Figure 14: Sketch Drawing Analogies.

In Figure 14, we construct two new latent vectors. The first latent vector, $\tilde{z}_1$, is constructed by adding the difference of two vectors $z_{\texttt{full pig}} - z_{\texttt{pig head}}$ to $z_{\texttt{cat head}}$. Using $\tilde{z}_1$ as in input to the RNN decoder, we sample images of varying temperature τ and obtain sketches of full cats. In the second example, the second latent vector, $\tilde{z}_2$, is constructed by adding the latent vector of a full pig to the difference between the latent vectors a cat head and a full cat, resulting in sketches of pig heads.

4.2.5 Multi-Sketch Drawing Interpolation

In addition to interpolating between two sketches, we can also visualize the interpolation between four sketches in latent space to gain further insight from the model.

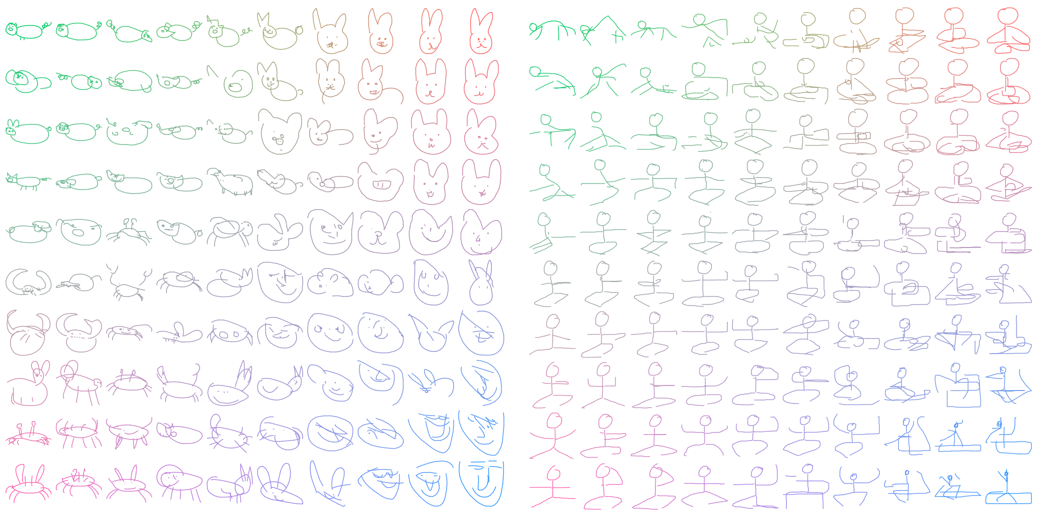

Figure 15: Interpolation of generated pig, rabbit, crab and face (Left), Latent space of generated yoga positions (Right).

The left side of Figure 15 visualizes the interpolation between a full pig, a rabbit's head, a crab, and a face, using a model trained on these four classes. In certain parts of the space between a crab and a face is a rabbit's head, and we see that the ears of the rabbit becomes the crab's claws. Applying the model on the yoga class, it is interesting to see how one yoga position slowly transitions to another via a set of interpolated yoga positions generated by the model. For visual effect, we also interpolate between four distinct colors, and color each sketch using a unique interpolated color.

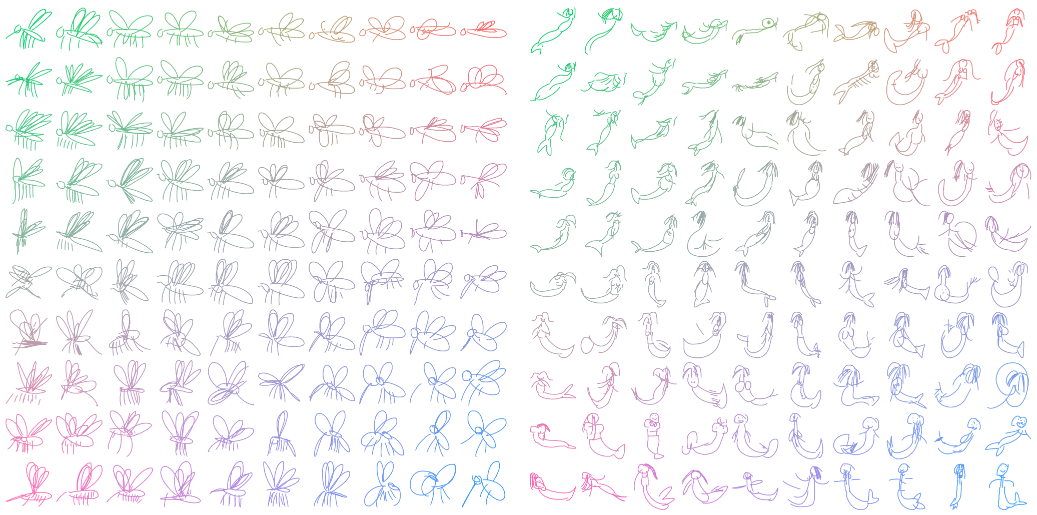

Figure 16: Latent space of generated mosquitoes and mermaids.

We also construct latent space interpolation examples for the mosquito class and the mermaid class, in Figure 16. We see that the model can interpolate between concepts such as style of wings, leg counts, and orientation.

Figure 17 is a delightful example where we use the model trained on the cat class to encode four different latent vectors from four different sketch drawings of chairs as inputs to obtain four sketches

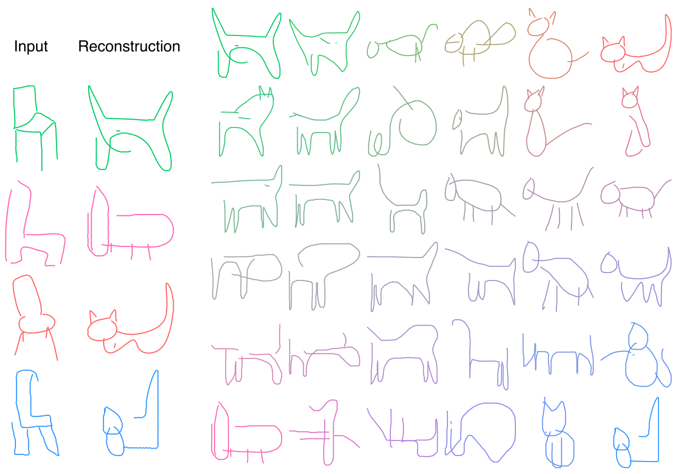

Figure 17: Latent space of generated cats conditioned on sketch drawings of chairs.

of cat-styled chairs. We can then interpolate and visualize the latent subspace of these four chair-like cats using the same model.

4.3 Unconditional generation

As a special case of `sketch-rnn`, we can use the decoder RNN as a standalone model to generate sketches without any inputs or latent vectors. This unconditional model is generally easier to train, as we only need to optimize for L_R. Furthermore, we do not need to train the complementary encoder system to a generate a meaningful vector space for the decoder. For unconditional generation with `sketch-rnn`, we find that lower L_R scores correspond directly with higher quality sketches.

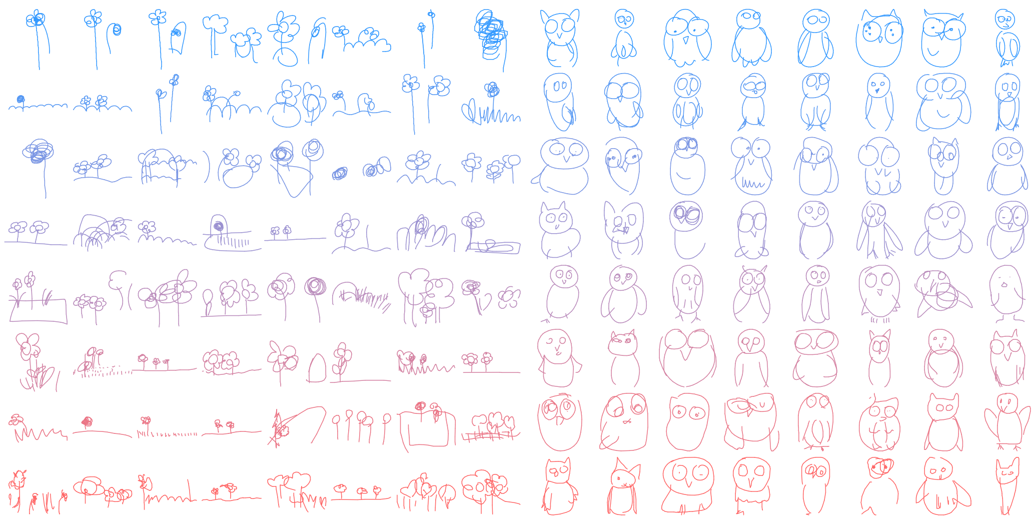

Figure 18: Unconditional generated sketches of gardens and owls.

Figure 18 displays a grid of samples from models trained on individual image classes with various temperature settings, varying from $\tau = 0.2$ in the blue region to $\tau = 0.9$ in the red region. Sketches generated using lower temperatures tend to be simpler, and constructed with smoother and more deterministic lines. Occasionally when the temperature is too low, the model stays within a single mixture of the GMM and simply draws repeated spirals until $N_{\max}$ points of the drawing have been sampled and it is forced to stop. With higher temperatures, the model can escape more easily from one mixture to another mixture of the GMM, leading to more diverse generated sketches.

4.3.1 Predicting Different Endings of Incomplete Sketches

We can use `sketch-rnn` to finish an incomplete sketch. By using the decoder RNN as a standalone model, we can generate a sketch that is conditional on the previous points. To do this, we use the decoder RNN to first encode an incomplete sketch into a hidden state h. Afterwards, we generate the remaining points of the sketch using h as the initial hidden state.

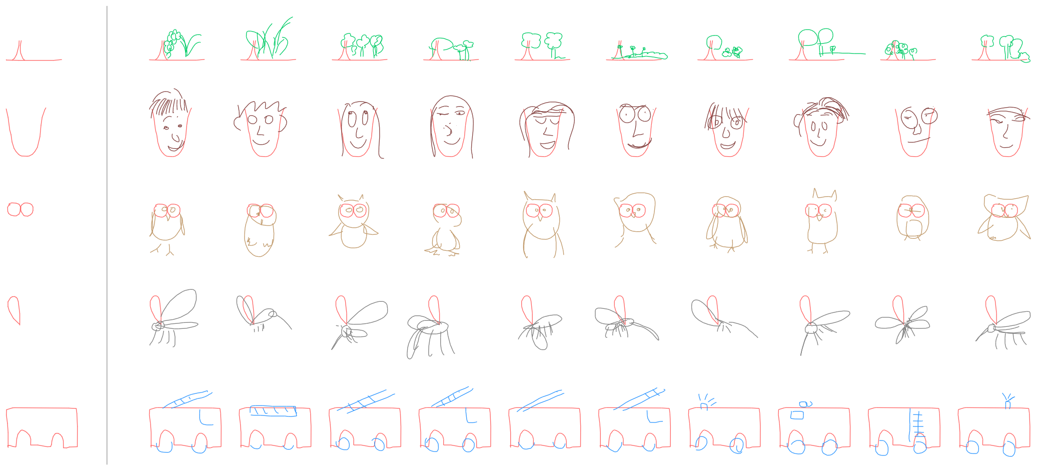

Figure 19: `sketch-rnn` predicting possible endings of various incomplete sketches (the red lines).

In Figure 19, we perform this experiment by training the standalone decoder on individual classes, and sample several different endings for the same incomplete sketch, using a constant temperature setting of $\tau = 0.8$. In the garden example, the model sketches other plants above the ground area that are consistently spaced, and occasionally draws the crown area on top of our tree trunk. When we draw the outline of a face, the model populates the sketch with remaining facial features such as eyes, nose and hair which are coherently spaced relative to the initial outline. The model also generates various mosquitoes at different orientations given the same starting wing, and various types of owls and firetrucks given the same set of eyes or truck body.

5 Discussion

5.1 Limitations

Although `sketch-rnn` can model a large variety of sketch drawings, there are several limitations in the current approach we wish to highlight. For most single-class datasets, `sketch-rnn` is capable of modelling sketches up to around 300 data points. The model becomes increasingly difficult to train beyond this length. For our dataset, we applied the Ramer–Douglas–Peucker algorithm [5] to simplify the strokes of the sketch data to less than 200 data points while still keeping most of the important visual information of each sketch.

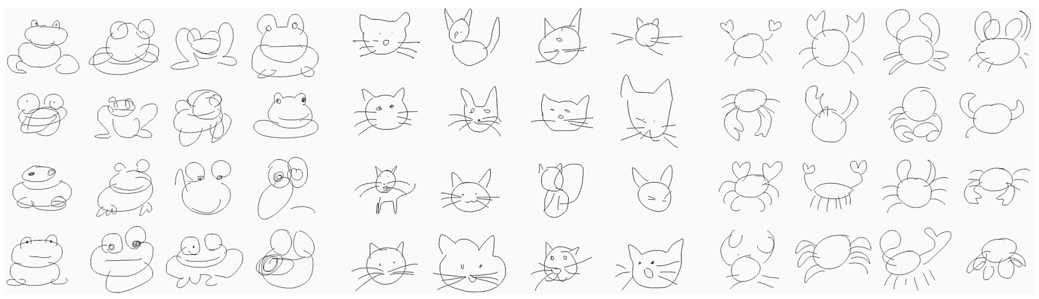

Figure 20: Unconditional generated sketches of frogs, cats, and crabs at $\tau = 0.8$.

For more complicated classes of images, such as mermaids or lobsters, the reconstruction loss metrics are not as good compared to simpler classes such as ants, faces or firetrucks. The models trained on these more challenging image classes tend to draw smoother, more circular line segments that do not resemble individual sketches, but rather resemble an averaging of many sketches in the training set. We can see some of this artifact in the frog class, in Figure 20. This smoothness may be analogous to the blurriness effect produced by a Variational Autoencoder [19] that is trained on pixel images. Depending on the use case of the model, smooth circular lines can be viewed as aesthetically pleasing and a desirable property.

While both conditional and unconditional models are capable of training on datasets consisting of several classes, such as (cat, pig), and (crab, face, pig, rabbit), `sketch-rnn` is not able to effectively model all 75 classes in the dataset. In Figure 21, we sample sketches using an unconditional model trained on all 75 classes, and a model trained on 4 classes. The samples generated from the 75-class model are incoherent, with individual sketches displaying features from multiple classes. The four-class unconditional model usually generates samples of a single class, but occasionally also combines features from multiple classes. For example, we can find some interesting generated sketches of crabs with the body of a face.

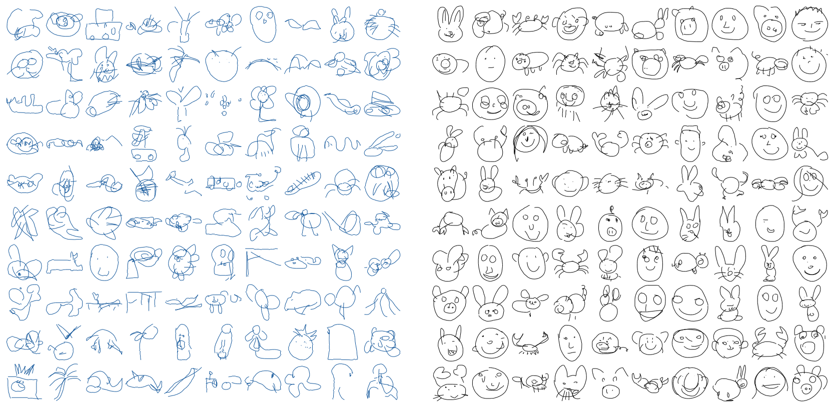

Figure 21: Unconditional generations from model trained on all classes (Left), From model trained on crab, face, pig and rabbit classes (Right).

5.2 Applications and Future Work

We believe `sketch-rnn` will find many creative applications. A `sketch-rnn` model trained on even a small number of classes can assist the creative process of an artist by suggesting many possible ways of finishing a sketch, helping the artist expand her imagination. A creative designer may find value in exploring the latent space between different image classes to find interesting intersections and relationships between two very different objects. Pattern designers can conditionally generate a large number of similar, but unique designs, for textile or wallpaper prints.

A more sophisticated model trained on higher quality sketches may find its way into educational applications that can help teach students how to draw. Even with the simple sketches in `quickdraw-75`, the authors of this work have become much more proficient at drawing animals, insects, and various sea creatures after conducting these experiments. A related application is to encode a poorly sketched drawing to generate more aesthetically looking reproductions by using a model trained with a high w_{KL} setting and sampling with a low temperature τ, as described in Section 4.2.3. In the future, we can also investigate augmenting the latent vector in the direction that maximizes the aesthetics of the drawing by incorporating user-rating data into the training process.

It will be interesting to explore various methods to improve the quality of the generated sketches. The choice of a Gaussian as the prior distribution of our Variational Autoencoder may be too simple to model a large dataset, and we may get better gains by incorporating more advanced variational inference models such a Ladder-VAEs [15] or Inverse Autoregressive Flow [18]. The addition of a GAN-style discriminator, or a critic network as a post processing step may lead to more realistic line styles and better reconstruction samples. This type of approach has been tested on VAE-GAN [21] hybrids on pixel images, and has recently been tested on text generation [12].

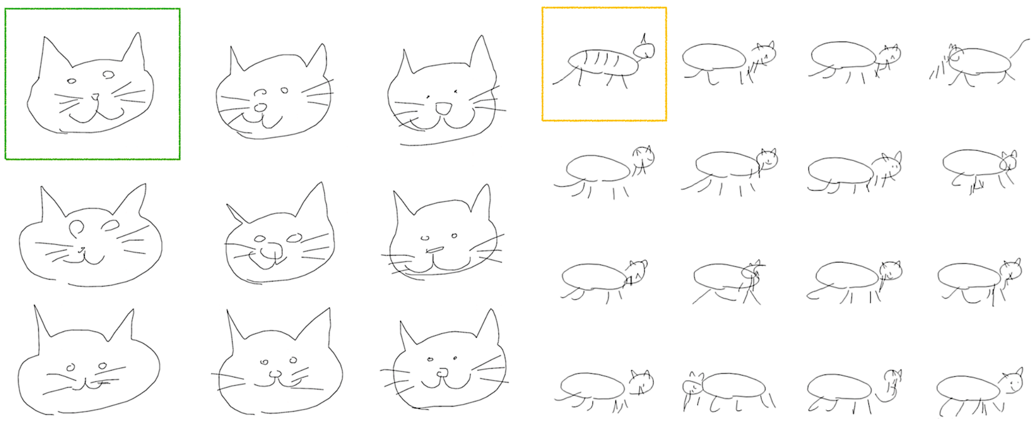
Figure 22: Generating similar, but unique sketches based on a single human sketch in the box.

Combining hybrid variations of sequence-generation models with unsupervised, cross-domain pixel image generation models, such as Image-to-Image models [4, 16, 23], is another exciting direction that we can explore. We can already combine this model with supervised, cross-domain models such as Pix2Pix [13], to occasionally generate photo realistic cat images from generated sketches of cats. The opposite direction of converting a photograph of a cat into an unrealistic, but similar looking sketch of a cat composed of a minimal number of lines seems to be a more interesting problem.

6 Conclusion

In this work, we develop a methodology to model sketch drawings using recurrent neural networks. `sketch-rnn` is able to generate possible ways to finish an existing, but unfinished sketch drawing. Our model can also encode existing sketches into a latent vector, and generate similar looking sketches conditioned on the latent space. We demonstrate what it means to interpolate between two different sketches by interpolating between its latent space. After training on a particular class of image, our model is able to correct features of an input it deems to be incorrect, and output a similar image with corrected properties. We also show that we can manipulate attributes of a sketch by augmenting the latent space, and demonstrate the importance of enforcing a prior distribution on the latent vector for coherent vector image generation.

7 Acknowledgements

We thank Ian Johnson, Jonas Jongejan, Martin Wattenberg, Mike Schuster, Thomas Deselaers, Ben Poole, Kyle Kastner, Junyoung Chung and Kyle McDonald for their help with this project. This work was done as part of the Google Brain Residency program (`g.co/brainresidency`).

References

[1] J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. *NIPS*, 2016.

[2] C. M. Bishop. Mixture density networks. *Technical Report*, 1994.

[3] S. R. Bowman, L. Vilnis, O. Vinyals, A. M. Dai, R. Józefowicz, and S. Bengio. Generating Sentences from a Continuous Space. *CoRR*, abs/1511.06349, 2015.

[4] H. Dong, P. Neekhara, C. Wu, and Y. Guo. Unsupervised Image-to-Image Translation with Generative Adversarial Networks. *ArXiv e-prints*, Jan. 2017.

[5] D. H. Douglas and T. K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10(2):112–122, Oct. 1973.

[6] M. Eitz, J. Hays, and M. Alexa. How Do Humans Sketch Objects? *ACM Trans. Graph. (Proc. SIGGRAPH)*, 31(4):44:1–44:10, 2012.

[7] I. Goodfellow. NIPS 2016 Tutorial: Generative Adversarial Networks. *ArXiv e-prints*, Dec. 2017.

[8] A. Graves. Generating sequences with recurrent neural networks. *arXiv:1308.0850*, 2013.

[9] D. Ha. Recurrent Net Dreams Up Fake Chinese Characters in Vector Format with TensorFlow. `http://blog.otoro.net/`, 2015.

[10] D. Ha, A. M. Dai, and Q. V. Le. HyperNetworks. In *ICLR*, 2017.

[11] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 1997.

[12] Z. Hu, Z. Yang, X. Liang, R. Salakhutdinov, and E. P. Xing. Controllable Text Generation. *ArXiv e-prints*, Mar. 2017.

[13] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros. Image-to-Image Translation with Conditional Adversarial Networks. *ArXiv e-prints*, Nov. 2016.

[14] J. Jongejan, H. Rowley, T. Kawashima, J. Kim, and N. Fox-Gieg. The Quick, Draw! - A.I. Experiment. *https://quickdraw.withgoogle.com/*, 2016.

[15] C. Kaae Sønderby, T. Raiko, L. Maaløe, S. Kaae Sønderby, and O. Winther. Ladder Variational Autoencoders. *ArXiv e-prints*, Feb. 2016.

[16] T. Kim, M. Cha, H. Kim, J. Lee, and J. Kim. Learning to Discover Cross-Domain Relations with Generative Adversarial Networks. *ArXiv e-prints*, Mar. 2017.

[17] D. Kingma and J. Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015.

[18] D. P. Kingma, T. Salimans, and M. Welling. Improving variational inference with inverse autoregressive flow. *CoRR*, abs/1606.04934, 2016.

[19] D. P. Kingma and M. Welling. Auto-Encoding Variational Bayes. *ArXiv e-prints*, Dec. 2013.

[20] B. M. Lake, R. Salakhutdinov, and J. B. Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, Dec. 2015.

[21] A. B. L. Larsen and S. K. Sønderby. Generating Faces with Torch. `http://torch.ch/blog/2015/11/13/gan.html`, 2015.

[22] Y. J. Lee, C. L. Zitnick, and M. F. Cohen. Shadowdraw: Real-time user guidance for freehand drawing. In *ACM SIGGRAPH 2011 Papers*, SIGGRAPH '11, pages 27:1–27:10, New York, NY, USA, 2011. ACM.

[23] M.-Y. Liu, T. Breuel, and J. Kautz. Unsupervised Image-to-Image Translation Networks. *ArXiv e-prints*, Mar. 2017.

[24] S. Reed, A. van den Oord, N. Kalchbrenner, S. Gómez Colmenarejo, Z. Wang, D. Belov, and N. de Freitas. Parallel Multiscale Autoregressive Density Estimation. *ArXiv e-prints*, Mar. 2017.

[25] P. Sangkloy, N. Burnell, C. Ham, and J. Hays. The Sketchy Database: Learning to Retrieve Badly Drawn Bunnies. *ACM Trans. Graph.*, 35(4):119:1–119:12, July 2016.

[26] M. Schuster, K. K. Paliwal, and A. General. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 1997.

[27] S. Semeniuta, A. Severyn, and E. Barth. Recurrent dropout without memory loss. *arXiv:1603.05118*, 2016.

[28] P. Tresset and F. Fol Leymarie. Portrait drawing by paul the robot. *Comput. Graph.*, 37(5):348–363, Aug. 2013.

[29] T. White. Sampling Generative Networks. *ArXiv e-prints*, Sept. 2016.

[30] N. Xie, H. Hachiya, and M. Sugiyama. Artist agent: A reinforcement learning approach to automatic stroke generation in oriental ink painting. In *ICML*. icml.cc / Omnipress, 2012.

[31] X. Zhang, F. Yin, Y. Zhang, C. Liu, and Y. Bengio. Drawing and Recognizing Chinese Characters with Recurrent Neural Network. *CoRR*, abs/1606.06539, 2016.

A Appendix

A.1 Dataset Details

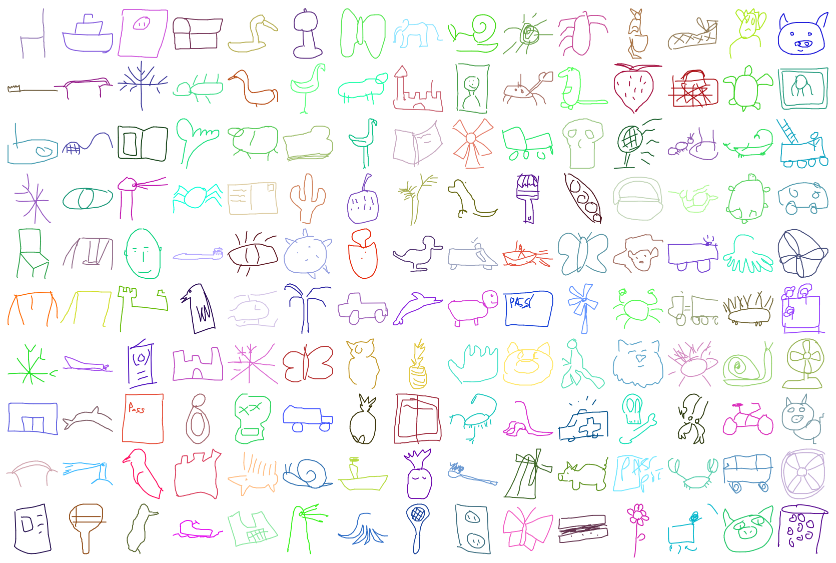

Figure 23: Example sketch drawings from `quickdraw-75` dataset.

The `quickdraw-75` dataset consists of 75 classes. Each class consists of 70K training samples and 2.5K validation and test samples. Stroke simplification using the Ramer–Douglas–Peucker algorithm [5] with a parameter of $\epsilon = 2.0$ has been applied to simplify the lines. The maximum sequence length is 200 data points. The data was originally recorded in pixel-dimensions, so we normalized the offsets $(\Delta x, \Delta y)$ using a single scaling factor. This scaling factor was calculated to adjust the offsets in the training set to have a standard deviation of 1. For simplicity, we do not normalize the offsets $(\Delta x, \Delta y)$ to have zero mean, since the means are already relatively small.

alarm clock	ambulance	angel	ant	barn
basket	bee	bicycle	book	bridge
bulldozer	bus	butterfly	cactus	castle
cat	chair	couch	crab	cruise ship
dolphin	duck	elephant	eye	face
fan	fire hydrant	firetruck	flamingo	flower
garden	hand	hedgehog	helicopter	kangaroo
key	lighthouse	lion	map	mermaid
octopus	owl	paintbrush	palm tree	parrot
passport	peas	penguin	pig	pineapple
postcard	power outlet	rabbit	radio	rain
rhinoceros	roller coaster	sandwich	scorpion	sea turtle
sheep	skull	snail	snowflake	speedboat
spider	strawberry	swan	swing set	tennis racquet
the mona lisa	toothbrush	truck	whale	windmill

Table 2: `quickdraw-75` dataset class list.