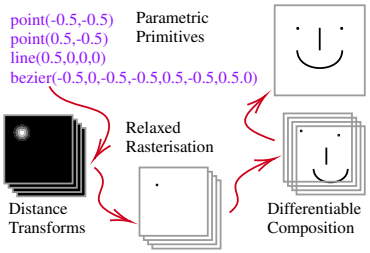
Differentiable Drawing and Sketching

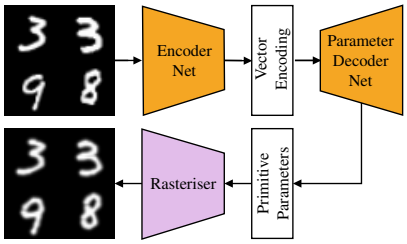
Daniela Mihai*
The University of Southampton
Southampton, UK
adm1g15@ecs.soton.ac.uk

Jonathon Hare*
The University of Southampton
Southampton, UK
jsh2@ecs.soton.ac.uk



(a) Overview of the proposed approach to drawing. The final image is differentiable with respect to the primitive parameters.

(b) Seurat's 'Une baignade à Asnières' reduced to straight lines instead of points by gradient descent through the rasteriser.



(c) Encoder-Decoder-Rasteriser model for learning to map images to primitives. Orange blocks have learnable parameters.

Figure 1: With a differentiable rasteriser (a), it is possible to optimise primitives (b), and build end-to-end learnable models (c).

Abstract

We present a bottom-up differentiable relaxation of the process of drawing points, lines and curves into a pixel raster. Our approach arises from the observation that rasterising a pixel in an image given parameters of a primitive can be reformulated in terms of the primitive's distance transform, and then relaxed to allow the primitive's parameters to be learned. This relaxation allows end-to-end differentiable programs and deep networks to be learned and optimised, and provides several building blocks that allow control over how a compositional drawing process is modelled. We emphasise the bottom-up nature of our proposed approach, which allows for drawing operations to be composed in ways that can mimic the physical reality of drawing rather than being tied to, for example, approaches in modern computer graphics. With the proposed approach we demonstrate how sketches can be generated by directly optimising against photographs and how auto-encoders can be built to transform rasterised handwritten digits into vectors without supervision. Extensive experimental results highlight the power of this approach under different modelling assumptions for drawing tasks.

1. Introduction and Motivation

This paper proposes a differentiable relaxation of the rasterisation process, which we ultimately demonstrate allows us to build end-to-end learnable machines that can perform both image generation and inference tasks. More concretely, we demonstrate that we can build machines that turn digital raster images into parametric representations of continuous paths, and back into rasterised images again. Our approach is not constrained by any particular modelling assumptions about how images should be composed beyond the functions used being differentiable. This allows us to closely model the physical act of drawing with a pen on paper for example. We believe that the proposed approach will ultimately have many applications in future approaches to computer vision tasks related to topics including sketch retrieval, recognition, and generation, as well as topics related to understanding and analysing handwriting, and even to more general topics around understanding visual communication.

When humans use drawing, sketching and writing to communicate they rarely do so by filling in pixels on a grid. Most methods (with some notable exceptions) of producing physically realised forms of drawing and writing by hand involve manipulating an instrument (a pen, paintbrush, pastel, etc) to mark a surface (paper, for example). In the digital world, this process is often approximated with vector graphics, in which paths are 'stroked' and then most often rasterised onto a pixel grid to produce digital images that can be displayed on a monitor or reproduced in hard copy.

To date, modelling the act of drawing with techniques such as deep neural networks has been relatively limited

*Authors contributed equally.

because the process of rasterisation using traditional approaches is not differentiable. The vast majority of recent work on image generation has operated on the principle of trying to optimise outputs broadly at the pixel level utilising tools such as transpose convolutions which operate on raster representations. There are of course exceptions to this statement, where researchers have attempted to more closely consider the underlying process that humans use to draw and write [*e.g.* 13], or to circumvent the non-differentiability of rasterisation [*e.g.* 38] using learning. These techniques, as well as a contemporaneous approach to relaxing modern vector graphics [16] (taking a complementary, but different approach to ours) which was published during the production of this manuscript, are described and discussed in section 2.

Our contributions are as follows: 1. We present a bottom-up differentiable approach (see fig. 1a) to generating pixel rasters from parameterised vector primitives by reformulating and relaxing the rasterisation problem. This is coupled with a set of formulations that allow different approaches to composition. A full exposition of our approach is in section 3. 2. We demonstrate that primitives can be optimised by minimising a loss against an existing raster image (cf. fig. 1b), and show how different losses inform the result. Details are in section 4. 3. We create a range of parameterisations of primitives in end-to-end learnable autoencoder architectures (see fig. 1c) for handwritten characters and objectively compare performance. See section 5 for details. We provide a PyTorch implementation of our approach, which allows others to experiment further. Code is available at `https://bit.ly/2PHtt5v`.

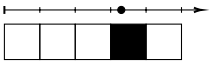
2. Related Work

Drawing, and in particular sketching, has been a means of conveying concepts, objects and stories since ancient times. There is a long history of sketch research in computer vision and human-computer interaction dating back to the 1960s [28]. Sketch applications have become increased in recent years due to the rapid development of deep-learning techniques that can successfully tackle tasks such as sketch recognition [36], generation [38, 8, 24], sketch-based retrieval [4, 23, 5], semantic segmentation [32, 35], grouping[15], parsing[25] and abstraction[20]. Xu et al. [34] offer a recent and detailed survey of free-hand sketch research and applications, focusing on contemporary deep-learning techniques. Our long-term goal for the work presented in this paper is to be able to train models to learn how to produce the parameters of drawing primitives based on visual inputs with only limited supervision. Internally within our models, we want to bridge the gap between input and output rasters, and internal vector representations.

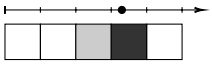
There is a body of recent literature describing models that operate purely on vector stroke data (that is, the process of actually drawing the vectors into an image is not part of the learning machinery). This includes recurrent generative models for sketch data [*e.g.* 8], generative models utilising GANs for sketch generation in vector format [*e.g.* 1], and reinforcement learning [*e.g.* 39, 33]. Another line of work within sketch generation uses Bayesian Program Learning, rather than deep networks, to represent the act of drawing as a probabilistic generative model [13].

With respect to models that turn raster images into vectors, there is considerable classical literature looking at the problem of 'stroke-based rendering' where the objective is to turn raster images into a sequence of strokes [*e.g.* 30, 9, 29] for artistic or visual communication purposes. A good overview of these can be found in the tutorial by Hertzmann [10], which breaks these approaches into Voronoi (broadly based on Lloyd's algorithm), or 'trial and error' approaches which try to minimise a loss based on heuristic tests. The approach presented in section 4 is clearly of relevance to this field of research, but a differentiable rasteriser allows for a potentially more principled or flexible approach to modelling the drawing process or the loss that is optimised. A number of models have been proposed that incorporate drawing into learning machinery. Until very recently, the process of rasterisation and rendering was thought to be non-differentiable, so two approaches were used to circumvent this problem: firstly there were models that use reinforcement learning to learn drawing actions through a traditional (non-differentiable) renderer [*e.g.* 6, 19], and, secondly there were approaches that 'learn' renderers (typically formulated as networks of transposed convolutions, or convolutions and upsampling operations) that can take vector inputs and produce raster outputs [38, 41, 21]. Of the latter, the work by Zheng et al. [38] is most similar to ours in its intent to work with sketches, and to utilise encoder models to produce accurate stroke parameters from raster images; our models in section 5 are however fully end-to-end learnable, unlike Zheng et al.'s model in which the renderer network is trained separately. Models with learned rasterisers are also inherently inflexible in the sense that they have to be trained for every type of stroke parameterisation they can work with.

Recent approaches to differentiable 3D rendering have garnered attention in the computer vision community [*e.g.* 17, 11], and indeed it is the work of Liu et al. [17] that originally helped inform the approach we detail in section 3. Most recently, during the development of our approach, Li et al. [16] presented a differentiable relaxation that takes advantage of how anti-aliasing is performed in modern computer graphics systems using multi-sampling, by providing differentiable relaxations. We consider this to be a top-down approach to the problem because it does not change the underlying rendering model. Conversely, we consider our approach to be bottom-up because we explicitly allow the rendering model to be flexibly defined in a way that is appropriate to the task.



(a) Nearest-neighbour rasterisation: the closest pixel is shaded by the by flooring the point ordinate. There is no useful gradient information.



(b) Anti-aliased rasterisation: the closest two pixels are shaded proportionally by to the distance from the point to those two pixels.

(c) Rasterisation using eq. (3) with $\sigma^2 = 1$. Every pixel in the image will have a (small) gradient with respect to the point ordinate.

Figure 2: Different rasterisation functions illustrated in one-dimension.

3. Differentiable relaxations of rasterisation

In this section we discuss the problem of drawing, or rasterising points, lines and curves defined in a continuous world space $\mathcal{W}$ into an image space $\mathcal{I}$. Our objective is to present a formalisation that allows us to ultimately define rasterisation functions that are differentiable with respect to their world space parameters (*e.g.* the (co)ordinate of a point, or (co)ordinates of the beginning and end of a line segment).

3.1. 1D Rasterisation

We first consider the problem of rasterising a one-dimensional point $p \in \mathcal{W}$ where $\mathcal{W} = \mathbb{R}$. Concretely, the process of rasterisation of the point p can be defined by a function, $f(n; p)$, that computes a value (typically $[0, 1]$) for every pixel in the image space $\mathcal{I}$, whose position is given by $n \in \mathcal{I}$. Such a function represents a scalar field over the space of possible values of n. Commonly we consider values of n to be non-negative integers from the lattice or grid, $\mathbb{Z}^1_{0+}$, defining a pixel in the image.

Simple closest-pixel rasterisation functions. If we assume that the 0th pixel covers the domain $[0, 1)$ in the world space of a point p, and that the 1st pixel covers $[1, 2)$, etc. Nearest-neighbour rasterisation then maps the real-valued point, p, to an image by rounding down:

$$f(n; p) = \begin{cases} 1 & \text{if } \lfloor p \rfloor = n \\ 0 & \text{otherwise .} \end{cases} \tag{1}$$

This process is illustrated in fig. 2a. An alternative rasterisation scheme, illustrated in fig. 2b is to interpolate over the two closest pixels. Assuming that a pixel has maximal value when the point being rasterised lies at its midpoint, then:

$$f(n; p) = \begin{cases} 1.5 - p + \lfloor p - 0.5 \rfloor & \text{if } \lfloor p - 0.5 \rfloor = n \\ 0.5 + p - \lceil p - 0.5 \rceil & \text{if } \lceil p - 0.5 \rceil = n \\ 0 & \text{otherwise .} \end{cases} \tag{2}$$

These functions (extended to 2D) are actually implicitly used in many computer graphics systems, but rarely in the form we have written them. Most graphics subroutines approach the rasterisation problem from the perspective of directly determining which pixels in n should have a colour associated with them given p as this is more efficient if the objective is just to draw the primitive p.

Differentiable Relaxations. Ideally, we would like to be able to define a rasterisation function that is differentiable with respect to p. This would allow p to be optimised with respect to some objective. The rasterisation function given by eq. (1) is piecewise differentiable with respect to p, but the gradient is zero almost everywhere which is not useful. Although eq. (2) has some gradient in the two pixels nearest to p, overall it has the same key problem: the gradient is zero almost everywhere.

We would like to define a rasterisation function that has gradient for all (or at least a large proportion of) possible values of n. This function should be continuous and differentiable almost everywhere. The anti-aliased rasterisation approach gives some hint as to how this could be achieved: the function could compute a value for every n based on the distance between n and p. Distance metrics have an infinite upper bound, whereas we want our pixel values to be finitely bounded in $[0, 1]$, so inversion and application of a non-linearity are necessary. The properties of the chosen function should give values close to 1 when n and p are *close*, and values near 0 when they are *far apart*.

An obvious choice of non-linearity would be to exponentiate the negative squared distances, and use a scaling factor σ^2 to control the fuzziness of the rasterisation and the size of the point or width of the line stroke:

$$f(n; p) = \exp(-d^2(n, p - 0.5)/\sigma^2) \, . \tag{3}$$

It can be shown that there is a direct linear relationship (see proof in appendix A) between the size of a point or thickness of a line, t, and the value of σ: $\sigma \approx 0.54925t \, \forall \, t > 0$.

3.2. Relaxed Rasterisation in N-dimensions

All of the 1D rasterisation functions previously defined can be trivially extended to rasterise a *point* in two or more dimensions. For example, if the point $\boldsymbol{p}$ was considered to be a vector in the world space $\mathcal{W} = \mathbb{R}^2$ and correspondingly $\boldsymbol{n}$ was a vector in the image space[1], $\mathcal{I} = \mathbb{Z}^2_{0+}$, and the floor and ceiling operators are applied element-wise then all three 1D rasterisation functions hold in two (or more) dimensions.

[1]Note that it is most common to use positive integers to index pixels in the image space, but this isn't a requirement; the image space could be unbounded or real for example.

Line Segments. A *line segment* can be defined by its start coordinate $\boldsymbol{s} = [s_x, s_y]$ and end coordinate $\boldsymbol{e} = [e_x, e_y]$. The normal approaches to rasterising lines in computer graphics [*e.g.* 2, 31] are highly optimised and work by considering just the pixels that intersect the line or are within a few pixels of it. These algorithms typically iterate over the line, setting the underlying pixels values accordingly. To develop a general set of (potentially differentiable) rasterisation functions we need to consider a formalisation of rasterisation as we did in the 1D case where we consider a function that defines a scalar field over the set of all pixel positions, n, in the image given a particular line segment: $f(\boldsymbol{n}; \boldsymbol{s}, \boldsymbol{e})$.

To rasterise a line segment one needs to consider how close a pixel is to the segment. We can efficiently compute the squared Euclidean distance of an arbitrary pixel $\boldsymbol{n}$ to the closest point on the line segment as follows:

$$\begin{aligned} \boldsymbol{m} &= \boldsymbol{e} - \boldsymbol{s} \ , \\ t &= ((\boldsymbol{n} - \boldsymbol{s}) \cdot \boldsymbol{m}) / (\boldsymbol{m} \cdot \boldsymbol{m}) \ , \\ \mathrm{d}^2_{\mathrm{seg}}(\boldsymbol{n}, \boldsymbol{s}, \boldsymbol{e}) &= \begin{cases} ||\boldsymbol{n} - \boldsymbol{s}||_2^2 & \text{if } t \leq 0 \\ ||\boldsymbol{n} - (\boldsymbol{s} + t\boldsymbol{m})||_2^2 & \text{if } 0 < t < 1 \\ ||\boldsymbol{n} - \boldsymbol{e}||_2^2 & \text{if } t \geq 1 \ . \end{cases} \end{aligned} \tag{4}$$

Concretely, $\mathrm{d}^2_{\mathrm{seg}}(\boldsymbol{n}, \boldsymbol{s}, \boldsymbol{e})$ is the squared Euclidean Distance Transform of the line segment. It defines a scalar field in which the value is equal to the squared distance to the closest point on the line segment. This function is piecewise smooth and differentiable with respect to the line segment parameters everywhere for a given $\boldsymbol{n}$.

In the case of nearest-neighbour rasterisation one would ask if the line passes through the pixel in question and only fill it if that were the case:

$$f(\boldsymbol{n}; \boldsymbol{s}, \boldsymbol{e}) = \begin{cases} 1 & \text{if } \mathrm{d}^2_{\mathrm{seg}}(\boldsymbol{n}, \boldsymbol{s}, \boldsymbol{e}) \leq \delta^2 \\ 0 & \text{otherwise} \ . \end{cases} \tag{5}$$

Assuming a 1-1 mapping between the domains of the coordinate system of the image space and world space, then $\delta^2 = 0.5$ would give a rasterisation that mimics the 1-pixel wide line that would be drawn by Bresenham's algorithm [2]. If we replace the calculation of distance to a point in eq. (3) with the minimum distance to the line segment we get a line segment rasteriser that is differentiable with respect to the parameters of the line segment $\boldsymbol{s}$ and $\boldsymbol{e}$:

$$f(\boldsymbol{n}; \boldsymbol{s}, \boldsymbol{e}) = \exp(-\mathrm{d}^2_{\mathrm{seg}}(\boldsymbol{n}, \boldsymbol{s}, \boldsymbol{e}) / \sigma^2) \ . \tag{6}$$

Curves. It is common in computer graphics to utilise parametric curves $C(t, \boldsymbol{\theta})$ where $\boldsymbol{\theta}$ defines the parameters and $0 \leq t \leq 1$. Typically $C(t, \boldsymbol{\theta})$ is polynomial (usually quadratic or cubic in t). The parameters $\boldsymbol{\theta}$ are commonly specified in Bézier (*e.g.* Bézier Curves) or Hermite form (*e.g.* Catmull-Rom splines) as described in appendix B. To rasterise a curve (irrespective of the parameterisation) in a way that is differentiable with respect to the parameters we can follow the same general approach that was taken for line segments: *compute the minimum Squared Euclidean distance between each coordinate $\boldsymbol{n} \in \mathcal{I}$ and the curve*:

$$\begin{aligned} \mathrm{d}^2_{\mathrm{cur}}(\boldsymbol{n}, \boldsymbol{\theta}) = \min_t \quad & ||C(t, \boldsymbol{\theta}) - \boldsymbol{n}||_2^2 \\ \text{s.t.} \quad & 0 \leq t \leq 1 \ . \end{aligned} \tag{7}$$

As in the case of line segments, this distance transform can then be combined with a rasterisation function that works in terms of a distance:

$$f(\boldsymbol{n}; \boldsymbol{\theta}) = \exp(-\mathrm{d}^2_{\mathrm{cur}}(\boldsymbol{n}, \boldsymbol{\theta}) / \sigma^2) \ . \tag{8}$$

The only additional challenge over the rasterisation of line segments is that the computation of the distance map requires solving a constrained minimisation problem and doesn't have a closed form solution. A number of different approaches are possible (see appendix C), however, in practice, we have had success with both a fast polyline approximation[2] and a recursive approach which are both easily vectorised as tensor operations that can be performed efficiently on a GPU.

3.3. Composing Multiple Primitives

To rasterise multiple lines[3] we can consider combining the rasterisations of different line segments into a single image. We denote images produced by rasterising different line segments $\{\boldsymbol{s_1}, \boldsymbol{e_1}\}, \{\boldsymbol{s_2}, \boldsymbol{e_2}\}, \ldots, \{\boldsymbol{s_i}, \boldsymbol{e_i}\}$ into matrices $\boldsymbol{I}^{(1)}, \boldsymbol{I}^{(2)}, \ldots, \boldsymbol{I}^{(n)}$ defined over the same image space $\mathcal{I}$. In the simplest case, where we have binary rasterisations, we might consider that the logical-or of corresponding pixels would produce the desired effect of selecting any pixels that were shaded in the individual rasterisations as being shaded in the final output:

$$c(\boldsymbol{I}^{(1)}, \boldsymbol{I}^{(2)}, \ldots, \boldsymbol{I}^{(n)}) = \boldsymbol{I}^{(1)} \vee \boldsymbol{I}^{(2)} \vee \cdots \vee \boldsymbol{I}^{(n)} \ . \tag{9}$$

We can relax this composition to be differentiable and also allow the pixel values to be non binary (but restricted to $[0, 1]$) as follows:

$$\mathrm{c}_{\mathrm{softor}}(\boldsymbol{I}^{(1)}, \boldsymbol{I}^{(2)}, \ldots, \boldsymbol{I}^{(n)}) = \mathbf{1} - \prod_{i=1}^{n} (\mathbf{1} - \boldsymbol{I}^{(i)}) \ . \tag{10}$$

Effectively if a pixel is 'on' in any of the individual images then this will select it as being 'on' in the output. This approach treats all the input images as a set; the output will be the same irrespective of the order they appear in.

[2] Note that there is a potential for a small change in curve's parameters to cause a large difference in the polyline approximation, although we have not seen this become an issue in practice.

[3] We're considering composing multiple line segments, but everything here also applies to multiple points and curves, as well as combinations of line segments, points and curves, or indeed any other raster.

Undoubtedly many other possible differentiable composition functions exist with alternative properties; we propose and discuss a number of these alternatives in appendix D. The experiments that follow use the above *soft-or* function, with the notable exception of the image in fig. 1b which was generated using the *over* composition (see appendix D) as this is more appropriate for colour images.

3.4. Extended drawing

Clearly, at this point, we now have all the components required to construct a basic drawing system. There are however a number of aspects that haven't been considered, including, for example how to draw in colour. As we focus the remainder of the paper on utilising the approach we have already described, discussion of additional extensions related to drawing and rasterisation can be found in appendix E.

3.5. Advantages and Limitations

The rasterisation process described in this section in principle allows gradients to flow from every pixel in the image to the parameters of a rendered primitive (note however that in practice this is not the case because of finite numeric precision). This is in contrast to the work of Li et al. [16] where the gradients are limited by the size of the filter. The advantage of our method is that optimisation should be easier with more gradient, however of course this does itself also have disadvantages. Computationally, our approach can be entirely implemented as batched tensor operations (this includes computation of distance transforms for all primitives), so all computation can be performed on the GPU making use of all available processing resources, and unlike Li et al. [16]'s approach, does not involve the CPU for rendering. The disadvantage is that memory usage could be very high, particularly for batches of large images with many primitives (in our original envisaged use case of exploring simple sketching and writing this is not a problem however). One interesting idea to explore in the future would be to utilise sparse tensors to reduce storage requirements by not storing pixels contributing to no value or gradient. Another potential criticism of our approach is that the generated images will be very slightly blurry as a result of the relaxation; again, for our envisaged use case this is not a problem, and it is always possible to use the relaxation for learning/optimisation, and then switch to a regular render for generation at inference time. Finally, we draw attention to the fact that our approach is not restricted to 2D, and can be *e.g.* directly applied to 3D data for voxel rasterisation.

4. Direct Optimisation of Primitive Parameters

With the machinery defined in section 3 it is now possible to define a complete system that takes the parameters describing primitives and rasterises those primitives into an image. If a loss function is introduced in the image space, between the complete rasterised image and a fixed target image, it becomes possible to compute gradients with respect to the parameters of the primitives that created the rasterised image. Minimising this loss will adapt the underlying primitives to "shapes" that best fit the target image.

A commonly used 'reconstruction' loss function for images is the mean squared error between the target and the generated image. We can thus formalise the optimisation problem as,

$$\min_{\boldsymbol{\theta}} \|R(\boldsymbol{\theta}) - \boldsymbol{T}\|_2^2 \,, \tag{11}$$

for a target image, $\boldsymbol{T}$ and rasterisation function R defined over the same image space $\mathcal{I}$. The rasterisation function itself is defined as a composition $c(\dots)$ (see section 3.3) over k primitives, themselves rasterised by primitive rasterisation functions, $f^{(i)}$ (*e.g.* eqs. (3), (6) and (8), etc.):

$$R(\boldsymbol{\theta}) = c(f^{(1)}(\boldsymbol{\theta}^{(1)}), f^{(2)}(\boldsymbol{\theta}^{(2)}), \dots, f^{(k)}(\boldsymbol{\theta}^{(k)})) \\ \text{where } \boldsymbol{\theta} = [\boldsymbol{\theta}^{(1)} | \boldsymbol{\theta}^{(2)} | \dots | \boldsymbol{\theta}^{(k)}] \,. \tag{12}$$

If the rasterisation function R is differentiable with respect to $\boldsymbol{\theta}$, then the minimisation problem in eq. (11) can be solved using gradient descent. Note that the problem is in general non-convex, with potentially many local optima[4]. Additionally, the magnitude of gradients can become vanishingly small, which is particularly problematic with fixed-precision arithmetic; this problem can however be overcome as we demonstrate in the following sections.

4.1. Loss Functions

MSE loss is not the only possible choice; in fact, MSE has one significant disadvantage in that if we are drawing in black and white, but optimising a grey-level image, then the loss landscape would be very flat. This is illustrated in table 1 where it can be seen that both configurations of the generated image in the first input to the loss function produce exactly the same MSE value. Human vision does not suffer from the same problem; we can see that the two input images to the loss functions are clearly different. In addition, if we look from far enough away the striped example on the second row, the image and the target would begin to look the same

[4]The optimisation landscape has considerable permutation symmetry. For example: the start and end of a line segment could be swapped with no change to the resultant image; if the composition is non-sequential the order of the rendered primitives could be permuted; etc.

$\mathcal{L} =$	MSE	SSMSE	BlurMSE
$\mathcal{L}$ ()	0.25	0.42	0.13
$\mathcal{L}$ ()	0.25	0.25	0.02

Table 1: Loss functions incorporating scale can overcome limitations of MSE and induce gradients.

to us. To build this phenomenon into the loss function we can incorporate a notion of spatial scale. We utilise two such approaches: BlurMSE, a single-scale version of MSE in which the input, and optionally the target are blurred by a Gaussian filter of a predetermined standard deviation; and the SSMSE, a scale-space variant in which a scale-space (or optionally a scale pyramid) is built for both the input and target, and the loss is accumulated over all levels. Our implementation of the scale-space follows Lowe [18], and constructs a space with octaves defined by a doubling of the standard deviation, and a fixed number of intervals per octave. As can be seen from table 1, the losses incorporating scale produce smaller values for the striped input image on the second row, compared to the half-half image on the first row, thus indicating usable gradient information.

A Gaussian scale-space alone is not necessarily a good measure of how a human perceives an image as it fundamentally only helps capture areas of light and dark, and ensures they are shaded accordingly in the generated image. Many perceptually motivated distance metrics have been proposed in the past, such as the well-known SSIM [40] and its variants. More recently, it has been shown that features from deep convolutional networks can correlate well with human perceptual judgements of image similarity, and this has motivated the development of CNN-based perceptual losses like LPIPS [37]. Because a loss based on deep features would inherently be differentiable, we can utilise it as an objective when optimising primitive parameters that define an image.

4.2. Examples: image based optimisation

To demonstrate the effectiveness of our approach for optimising primitives against a real image we provide a number of example results. All of the generated images in figs. 4 and 5 utilise the 200×266 pixels input image in fig. 3a as the target image to optimise against. Points and pixels are optimised to have a 1-pixel diameter/thickness in the image space. The domain of the world-space is constrained to [-1,1] on the y-axis and scaled proportionally on the x-axis. All generated examples were optimised using Adam [12] with a learning rate of 0.01 for 500 iterations. Figure 4 shows the results from optimising 1000 points and 1000 lines using blurred MSE loss and demonstrates the overall effect that can be achieved. Figure 5 demonstrates the effect of optimising 500 line segments from the same starting point using a range of different loss functions.

It is instructive to compare how the automatically generated sketches compare to an image drawn by a human. Figure 3b is a hand-drawn pen and ink sketch of the same scene as used in the generation of figs. 4 and 5. It is clear that all of the sketches broadly capture the overall structure of the scene and areas of light and dark. However, there are significant differences in the way this is captured. The losses based on MSE (including scale-space and blurred) all display the same trait of capturing the local intensity, although this is much more pronounced in the scale-space and blur variants, which also capture more detail. Changing the number of intervals per octave in the scale-space losses has very little overall effect (subtle changes around the 'balloon'). The perceptual loss using AlexNet captures highly local structure, but overall the resultant image is perhaps the least perceptually similar (or interpretable) of all the images. The perceptual loss using VGG captures a lot of the structure of the image; it is interesting how much of the broad shape information is captured, and how areas of light and dark are also represented. In addition, we can observe that the overall brightness on the right-hand side is lighter than the left, mimicking the human-drawn sketch, even though the raw grey-level values in the input image are similar on both sides. The differences between the two perceptual losses reflect the observation that the VGG variant is closer to traditional notions of perceptual difference when used for optimisation [37]. Related to the observation that the VGG model seems to capture shape information rather well, we wonder if direct optimisation in the way we have performed it might lead to a new way to probe the (lack of) shape bias in different neural architectures [7]. This could ultimately help us move closer to networks that robustly recognise objects from both sketches and photographs.

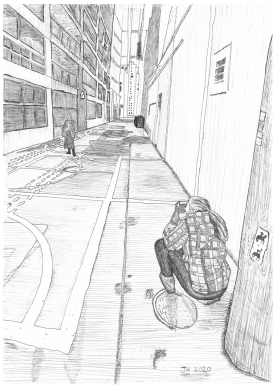
5. Autotracing autoencoders

We next look at models that learn to perform autotracing of handwritten characters with only self-supervision. The structure of our autotracing model, shown in fig. 1c, is similar to that of a standard autoencoder, with two main components: an image encoder that creates a latent encoding, and a parameter decoder that decodes a latent vector to 'stroke data'. This stroke data is then rasterised into the output image. Both the encoder and parameter decoder have learnable parameters, but the rasterisation is entirely fixed.

We next demonstrate a series of decoders which allow for different approaches to drawing. For example, we consider stroke parametrisation functions such as independent straight lines/curves, connected lines/curves through a series of consecutive points, and sets of points with learned connections between them. These models lay the groundwork for future exploration of learned, differentiable models of sketching that are more similar to how humans write/draw that *e.g.* address the challenges set out by Lake et al. [13].

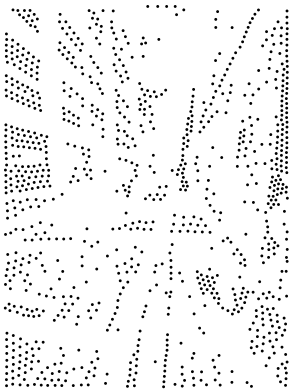
Encoders. For experiments on MNIST [14], we present results using a simple multi-layer perceptron encoder network. For more complex characters of Omniglot [13], a convolutional network is preferred. When comparing against StrokeNet [38] (table 2b), we replicate their VGG-like Encoder. Full model details are provided in appendix F.1.

(a) Photo

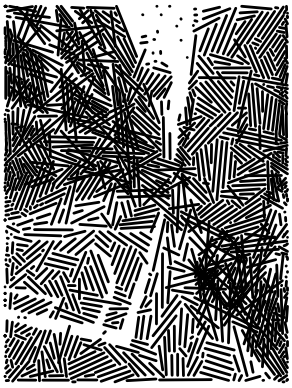


(b) Pen & Ink Sketch

Figure 3: Pictures of the author, by the other author.

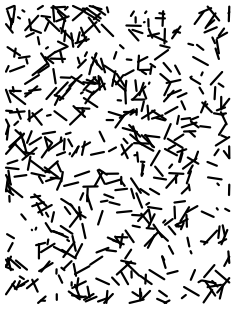


(a) 1000 Points



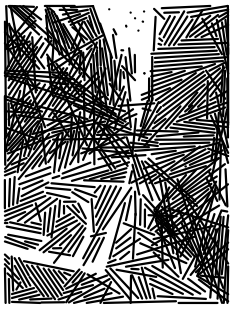
(b) 1000 lines

Figure 4: Optimising against fig. 3a using BlurMSE ($\sigma = 1.0$).



(a) Initialisation

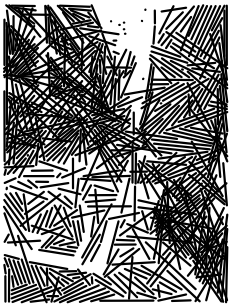
(b) MSE



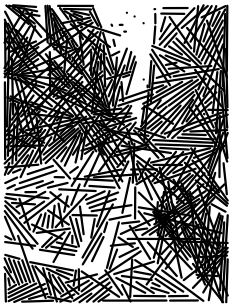
(c) BlurMSE ($\sigma = 1.0$)

(d) BlurMSE ($\sigma = 3.0$)

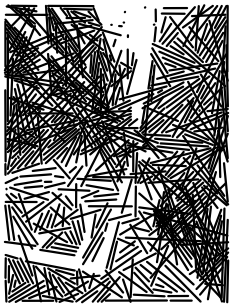
(e) BlurMSE ($\sigma = 5.0$)



(f) SSMSE, 1i/o



(g) SSMSE, 2i/o



(h) SSMSE, 4i/o

(i) LPIPS(AlexNet)

(j) LPIPS(VGG)

Figure 5: Images created by optimising parameters using gradient descent with different losses. Parameters optimised to fit the photo shown in fig. 3a starting from the random lines in fig. 5a. All images using SSMSE use 5 octaves, and 'i/o' abbreviates the number of intervals per octave. Note that regular MSE is just BlurMSE with $\sigma = 0$.

Decoders. Our decoder networks allow for different parametrisations of 'stroke data' that is then used by the rasteriser described in section 3. The decoder transforms a vector encoding of the input image to lists of stroke primitives which aim to reproduce the input image when rasterised. In the simplest case, the latent vector can be decoded to a fixed number of line segments (*LineDecoder*), each defined by their start and end points. Next, we provide *PolyLineDecoder*, for which a stroke is represented as a sequence of consecutive points. Instead of line segments, we can choose to use curves parameterised as Catmull-Rom splines (*CRSDecoder*) or Bézier curves (*BézierDecoder*). In both cases, we can control the number of joined curve segments by specifying how many points (CRS) or segments (Bézier) are used. To allow more flexibility in modelling, we have also explored decoders which incorporate sub-networks to learn to produce a set of 2d points, and the upper-triangular portion of a soft connection matrix between points (optionally including the

Decoder	#P	#S	#L	MSE	Acc.
Line	10	1	5	0.0195	94.06%
PolyLine	16	15	1	0.0225	93.27%
PolyConnect	16	-	-	0.0118	96.47%
CRS	16	14	1	0.0208	94.63%
Bézier	20	1	5	0.0136	96.34%
BézierConnect	16	-	-	0.0116	96.43%

(a) MNIST Test Dataset (baseline unencoded acc. 98.60%).

Model	Steps	#P	#S	#L	Acc.
StrokeNet [38]	3 (SN)	16	14	1	95.25%
StrokeNet [38]	1	16	14	1	97.75%
Ours, CRS	1	16	14	1	97.12%
Ours, Bézier	3 (GRU)	4	1	1	96.97%
Ours, Bézier	1	7	2	2	98.28%
Ours, Bézier	1	43	14	1	97.94%

(b) Scaled MNIST Dataset (baseline unencoded acc. 98.58%).

Table 2: Reconstruction performance of parameterisations, measured by MSE and classification accuracy with a classifier trained on unencoded training sets of the respective datasets. #* indicates the number of (L)ines, (S)egments, and (P)oints. All Scaled MNIST models use the same 'StrokeNet Agent' architecture [38] to map images to primitive parameters.

diagonal). The network producing the connection matrix uses a sigmoid to ensure values are between 0 and 1. To utilise the connection matrix, all possible combinations of lines are rasterised and are multiplied by the appropriate connection weight before composition (*PolyConnect*). In the case of Bézier curves (*BézierConnect*) each point in the connection matrix corresponds to both an end point and its corresponding control point, and when drawing curves, the end point is drawn using the mirror of its control point allowing for smooth multiple-segment curves to be created. Zheng et al. [38] proposed a recurrent model using a visual working memory; the network is presented at each timestep with the features of the target image, together with the current canvas, which is then encoded, concatenated with the input, and transformed to the parameters of a new stroke which is rendered and overlaid on the canvas. We experimented with this approach but found it hard to train and computationally expensive, so we also investigated a simple GRU [3] based RNN which is fed a target image's encoding as its initial hidden state, along with a projection of a zeroed input. The GRU output is projected to a set of Bézier curve parameters for rendering, and also re-projected for input at the next time step. Full details are given in appendix F.2.

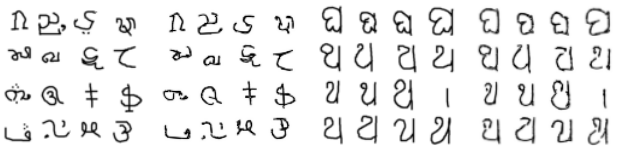
Table 2a shows the effect of different stroke parametrisations on MNIST (reconstructions shown in appendix G). As an objective measure, we compute the classification accuracy of rasterised sketches from the test set using a classifier (baseline accuracy of 98.6%); reconstructions that capture the character should have higher accuracy. *Connect* models, which generate strokes based on a learned connection matrix for the given number of points, perform best due to the flexibility of deciding which points should be joined in a line/curve segment. Following Zheng et al. [38] we perform a similar experiment on their scaled MNIST dataset (see appendix I), and also show results using the pretrained StrokeNet models that are publicly available. The accuracies of all models are high indicating good reconstructions, but we note that MNIST doesn't require complex decoders.

Figure 6 illustrates reconstruction of Omniglot [13]. Note that the test set contains alphabets completely disjoint from training/validation. Some small details of the characters are missing, and it is clear that the models do not always choose to draw stokes in the way a human would, but the performance is generally good (see appendix H).

6. Future Directions

We have presented a derivation of a bottom-up differentiable approach to rasterising vector primitives into images, that allows gradients to flow through every pixel in the image to the underlying primitive's parameters. Our approach allows us to construct end-to-end models of vision that learn primitive parameters directly from raster images. Further, we have demonstrated how effective sketch generation can be achieved with different losses, and how parameterisations can change what a model learns.

Our approach is only a building block towards future applications and research. Our own motivation for designing this approach is to use it to explore writing and visual communication, although there are undoubtedly many potential use-cases. For us, questions to be answered next involve looking at how we might build models that can learn to produce the appropriate number of strokes (and choose between different types of primitive). As part of this, it is clear that reconstruction performance alone should not be the key driver of gradient; the ability to communicate information is more important. Both attention and weak supervision to better mimic humans are also key to this endeavour.



(a) Val (b) Val rec. (c) Test (d) Test rec.

Figure 6: 28 pixel Omniglot validation and test data samples and *Bézier* model (3 segment, 5 line) reconstructions.

References

[1] S Balasubramanian, Vineeth N Balasubramanian, et al. Teaching gans to sketch in vector format. *arXiv preprint arXiv:1904.03620*, 2019. 2

[2] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965. 4

[3] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, Oct. 2014. Association for Computational Linguistics. 8

[4] Jungwoo Choi, Heeryon Cho, Jinjoo Song, and Sang Min Yoon. Sketchhelper: Real-time stroke guidance for freehand sketch retrieval. *IEEE Transactions on Multimedia*, 21(8):2083–2092, 2019. 2

[5] Antonia Creswell and Anil Anthony Bharath. Adversarial training for sketch retrieval. In *European Conference on Computer Vision*, pages 798–809. Springer, 2016. 2

[6] Yaroslav Ganin, Tejas Kulkarni, Igor Babuschkin, S. M. Ali Eslami, and Oriol Vinyals. Synthesizing programs for images using reinforced adversarial learning. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine Learning Research*, pages 1652–1661. PMLR, 2018. 2

[7] Robert Geirhos, Patricia Rubisch, Claudio Michaelis, Matthias Bethge, Felix A. Wichmann, and Wieland Brendel. Imagenet-trained CNNs are biased towards texture; increasing shape bias improves accuracy and robustness. In *International Conference on Learning Representations*, 2019. 6

[8] David Ha and Douglas Eck. A neural representation of sketch drawings. In *International Conference on Learning Representations*, 2018. 2

[9] Aaron Hertzmann. Painterly rendering with curved brush strokes of multiple sizes. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, page 453–460, New York, NY, USA, 1998. Association for Computing Machinery. 2

[10] A. Hertzmann. A survey of stroke-based rendering. *IEEE Computer Graphics and Applications*, 23(4):70–81, 2003. 2

[11] Hiroharu Kato, Yoshitaka Ushiku, and Tatsuya Harada. Neural 3d mesh renderer. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018. 2

[12] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. 6

[13] Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015. 2, 6, 8

[14] Yann LeCun, Corinna Cortes, and Christopher JC Burges. The mnist database of handwritten digits, 1998. *URL http://yann. lecun. com/exdb/mnist*, 10:34, 1998. 6

[15] Ke Li, Kaiyue Pang, Jifei Song, Yi-Zhe Song, Tao Xiang, Timothy M. Hospedales, and Honggang Zhang. Universal sketch perceptual grouping. In *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018. 2

[16] Tzu-Mao Li, Michal Lukáč, Michaël Gharbi, and Jonathan Ragan-Kelley. Differentiable vector graphics rasterization for editing and learning. *ACM Trans. Graph.*, 39(6), Nov. 2020. 2, 5, 13

[17] S. Liu, W. Chen, T. Li, and H. Li. Soft rasterizer: A differentiable renderer for image-based 3d reasoning. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 7707–7716, 2019. 2

[18] David G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, Nov. 2004. 6

[19] John F. J. Mellor, Eunbyung Park, Yaroslav Ganin, Igor Babuschkin, Tejas Kulkarni, Dan Rosenbaum, Andy Ballard, Theophane Weber, Oriol Vinyals, and S. M. Ali Eslami. Unsupervised doodling and painting with improved SPIRAL. *CoRR*, abs/1910.01007, 2019. 2

[20] Umar Riaz Muhammad, Yongxin Yang, Yi-Zhe Song, Tao Xiang, and Timothy M. Hospedales. Learning deep sketch abstraction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018. 2

[21] Reiichiro Nakano. Neural painters: A learned differentiable constraint for generating brushstroke paintings. *CoRR*, abs/1904.08410, 2019. 2

[22] Thomas Porter and Tom Duff. Compositing digital images. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, page 253–259, New York, NY, USA, 1984. Association for Computing Machinery. 14, 15

[23] Patsorn Sangkloy, Nathan Burnell, Cusuh Ham, and James Hays. The sketchy database: learning to retrieve badly drawn bunnies. *ACM Transactions on Graphics (TOG)*, 35(4):1–12, 2016. 2

[24] Patsorn Sangkloy, Jingwan Lu, Chen Fang, Fisher Yu, and James Hays. Scribbler: Controlling deep image synthesis with sketch and color. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5400–5409, 2017. 2

[25] Ravi Kiran Sarvadevabhatla, Isht Dwivedi, Abhijat Biswas, and Sahil Manocha. Sketchparse: Towards rich descriptions for poorly drawn sketches using multi-task hierarchical deep networks. In *Proceedings of the 25th ACM international conference on Multimedia*, pages 10–18, 2017. 2

[26] Thomas W Sederberg and Geng-Zhe Chang. Isolator polynomials. In *Algebraic Geometry and Its Applications*, pages 507–512. Springer, 1994. 13

[27] Erik Sintorn and Ulf Assarsson. Hair self shadowing and transparency depth ordering using occupancy maps. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, I3D '09, page 67–74, New York, NY, USA, 2009. Association for Computing Machinery. 14

[28] Ivan E Sutherland. Sketchpad a man-machine graphical communication system. *Simulation*, 2(5):R–3, 1964. 2

[29] Georges Winkenbach and David H. Salesin. Computer-

generated pen-and-ink illustration. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, page 91–100, New York, NY, USA, 1994. Association for Computing Machinery. 2

[30] Georges Winkenbach and David H. Salesin. Rendering parametric surfaces in pen and ink. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, page 469–476, New York, NY, USA, 1996. Association for Computing Machinery. 2

[31] Xiaolin Wu. An efficient antialiasing technique. *SIGGRAPH Comput. Graph.*, 25(4):143–152, July 1991. 4

[32] X. Wu, Y. Qi, J. Liu, and J. Yang. Sketchsegnet: A rnn model for labeling sketch strokes. In *2018 IEEE 28th International Workshop on Machine Learning for Signal Processing (MLSP)*, pages 1–6, 2018. 2

[33] Ning Xie, Hirotaka Hachiya, and Masashi Sugiyama. Artist agent: A reinforcement learning approach to automatic stroke generation in oriental ink painting. *IEICE TRANSACTIONS on Information and Systems*, 96(5):1134–1144, 2013. 2

[34] Peng Xu, Timothy M Hospedales, Qiyue Yin, Yi-Zhe Song, Tao Xiang, and Liang Wang. Deep learning for free-hand sketch: A survey and a toolbox. *arXiv e-prints*, pages arXiv–2001, 2020. 2

[35] Lumin Yang, Jiajie Zhuang, Hongbo Fu, Kun Zhou, and Youyi Zheng. Sketchgcn: Semantic sketch segmentation with graph convolutional networks. *arXiv preprint arXiv:2003.00678*, 2020. 2

[36] Qian Yu, Yongxin Yang, Feng Liu, Yi-Zhe Song, Tao Xiang, and Timothy M Hospedales. Sketch-a-net: A deep neural network that beats humans. *International journal of computer vision*, 122(3):411–425, 2017. 2

[37] Richard Zhang, Phillip Isola, Alexei A. Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018. 6

[38] Ningyuan Zheng, Yifan Jiang, and Dingjiang Huang. Strokenet: A neural painting environment. In *International Conference on Learning Representations*, 2019. 2, 6, 8, 16, 17, 19

[39] Tao Zhou, Chen Fang, Zhaowen Wang, Jimei Yang, Byungmoon Kim, Zhili Chen, Jonathan Brandt, and Demetri Terzopoulos. Learning to doodle with stroke demonstrations and deep q-networks. In *British Machine Vision Conference 2018, BMVC 2018, Newcastle, UK, September 3-6, 2018*, page 13. BMVA Press, 2018. 2

[40] Zhou Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004. 6

[41] Zhengxia Zou, Tianyang Shi, Shuang Qiu, Yi Yuan, and Zhenwei Shi. Stylized neural painting. *CoRR*, 2020. 2

Appendices

Figure VII: Is this what Georges might choose to paint if he decided to paint with straight lines rather than points? This image was created by taking a photo of the original 'Une baignade à Asnières' from Wikipedia (https://en.wikipedia.org/wiki/Bathers_at_Asnieres) and optimising 500 uniformly coloured lines to fit the image.

A. Relating σ to line thickness.

Consider the 1D rasterisation of a point of size t given by the scaled unit box function $\Pi(x/t)$ and the relaxed rasterisation given by $exp(-d^2(x)/\sigma^2)$ as illustrated in fig. VIII. We want to find a relationship between the value of t and σ^2 when trying to minimise the squared difference of the functions across the entire domain x,

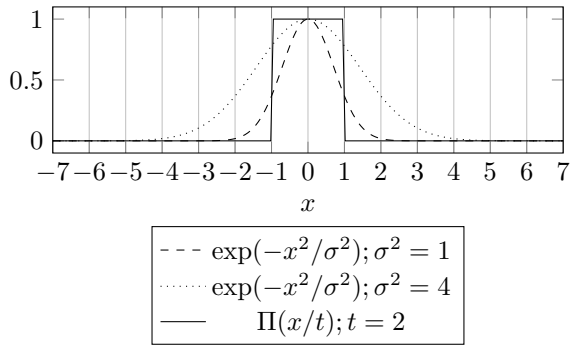


Figure VIII: Illustration of a target point of thickness $t = 2$ pixels and approximations with the exp rasterisation function with different σ^2 values.

$$\begin{aligned} \min_{\sigma^2} \quad & \int_{-\infty}^{\infty} (e^{-x^2/\sigma^2} - \Pi(x/t))^2 \, dx \\ \text{s.t.} \quad & t > 0 \\ & \sigma^2 > 0 \, . \end{aligned} \tag{A.1}$$

The integral term can be expanded and evaluated as follows (assuming the constraints $t > 0$ and $\sigma^2 > 0$):

$$\begin{aligned} & \int_{-\infty}^{\infty} (e^{-x^2/\sigma^2} - \Pi(x/t))^2 \, dx \\ & = \int_{-\infty}^{\infty} e^{-2x^2/\sigma^2} - 2\Pi(x/t)e^{-x^2/\sigma^2} + \Pi(x/t)^2 \, dx \\ & = \sigma\sqrt{\frac{\pi}{2}} - 2\sigma\sqrt{\pi}\,\mathrm{erf}\left(\frac{t}{2\sigma}\right) + t \, . \end{aligned} \tag{A.2}$$

Now, differentiating and setting to zero gives

$$
\begin{aligned}
0 &= \frac{d\left(\sigma\sqrt{\frac{\pi}{2}} - 2\sigma\sqrt{\pi}\,\mathrm{erf}\left(\frac{t}{2\sigma}\right) + t\right)}{d\sigma} \\
&= \sqrt{\frac{\pi}{2}} - 2\sqrt{\pi}\frac{d\left(\sigma\,\mathrm{erf}\left(\frac{t}{2\sigma}\right)\right)}{d\sigma} \\
&= \sqrt{\frac{\pi}{2}} - 2\sqrt{\pi}\left(\mathrm{erf}\left(\frac{t}{2\sigma}\right) + \sigma\frac{d\left(\mathrm{erf}\left(\frac{t}{2\sigma}\right)\right)}{d\sigma}\right) \\
&= \sqrt{\frac{\pi}{2}} - 2\sqrt{\pi}\,\mathrm{erf}\left(\frac{t}{2\sigma}\right) - 2\sqrt{\pi}\sigma\left(-\frac{te^{-t^2/(4\sigma^2)}}{\sigma^2\sqrt{\pi}}\right) \\
&= \sqrt{\frac{\pi}{2}} - 2\sqrt{\pi}\,\mathrm{erf}\left(\frac{t}{2\sigma}\right) + \frac{2te^{-t^2/(4\sigma^2)}}{\sigma}\,. \qquad \text{(A.3)}
\end{aligned}
$$

Noting the common factors of t/σ in eq. (A.3) we can write the right hand side as an expression in terms of $c = t/\sigma$:

$$
\sqrt{\frac{\pi}{2}} - 2\sqrt{\pi}\,\mathrm{erf}\left(\frac{c}{2}\right) + 2ce^{-c^2/4}\,. \qquad \text{(A.4)}
$$

As shown in fig. IX this expression is monotonically decreasing and has a single root, which can be estimated numerically as $c \approx 1.820657$.

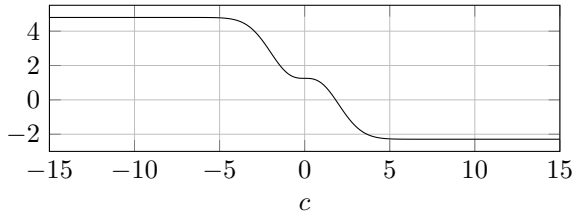


Figure IX: Plot of eq. (A.4).

This implies the relationship between t and σ is linear: $\sigma \approx 0.54925t\,\forall\, t > 0$. This can be easily verified by substituting $\sigma = 0.54925t$ in eq. (A.3).

B. Curve Parameterisations

Curves are often represented mathematically by parametric functions $C(t)$ that give the coordinates of the curve for values of t, commonly in the closed interval $[0, 1]$, with $t = 0$ representing the start point and $t = 1$ representing the end point of the curve. Curves are often parameterised as the coefficients of polynomial basis functions, either in Hermite (e.g. the curve is a linear combination of Hermite bases) or Bézier form (the curve is represented as a linear combination of Bernstein bases). The following parametric curve formulations are commonly used in computer graphics (and can all be used in our differentiable rasterisation approach):

Quadratic Bézier Curves are parameterised by three points: the start of the curve $\boldsymbol{P}_0$, the end of the curve $\boldsymbol{P}_2$ and the control point $\boldsymbol{P}_1$ which is the point the tangents to the curve at $\boldsymbol{P}_0$ and $\boldsymbol{P}_2$ intersect. The curve would not normally pass through $\boldsymbol{P}_1$. The curve can be thought of as leaving $\boldsymbol{P}_0$ in the direction of $\boldsymbol{P}_1$ and gradually bending to arrive at $\boldsymbol{P}_2$ from the direction of $\boldsymbol{P}_1$. The quadratic Bézier is defined as:

$$
\mathrm{C}_{\mathrm{bez}^2}(t, \boldsymbol{\theta}) = (1-t)^2\boldsymbol{P}_0 + 2(1-t)t\boldsymbol{P}_1 + t^2\boldsymbol{P}_2 \qquad \text{(B.1)}
$$

where

$$
\boldsymbol{\theta} = [\boldsymbol{P}_0|\boldsymbol{P}_1|\boldsymbol{P}_2]\,.
$$

Cubic Bézier Curves are defined by four points: $\boldsymbol{P}_0$ is the start of the curve; $\boldsymbol{P}_1$ is the first control point and indicates the direction the curve leaves $\boldsymbol{P}_0$ from; $\boldsymbol{P}_2$ is the second control point and indicates the direction that the curve arrives at the final end point $\boldsymbol{P}_3$ from. The curve would not normally pass through either control point. The cubic Bézier is defined as:

$$
\begin{aligned}
\mathrm{C}_{\mathrm{bez}^3}(t, \boldsymbol{\theta}) = &(1-t)^3\boldsymbol{P}_0 + 3(1-t)^2t\boldsymbol{P}_1 \\
&+ 3(1-t)t^2\boldsymbol{P}_2 + t^3\boldsymbol{P}_3 \qquad \text{(B.2)}
\end{aligned}
$$

where

$$
\boldsymbol{\theta} = [\boldsymbol{P}_0|\boldsymbol{P}_1|\boldsymbol{P}_2|\boldsymbol{P}_3]\,.
$$

Catmull-Rom Splines parameterise a curve by 4 points which the curve passes through smoothly. The curve is only drawn between the middle pair of points:

$$
\mathrm{C}_{\mathrm{crs}}(t, \boldsymbol{\theta}) = \frac{t_2 - t}{t_2 - t_1}\boldsymbol{B}_1 + \frac{t - t_1}{t_2 - t_1}\boldsymbol{B}_2 \qquad \text{(B.3)}
$$

where

$$
\begin{aligned}
\boldsymbol{B}_1 &= \frac{t_2 - t}{t_2 - t_0}\boldsymbol{A}_1 + \frac{t - t_0}{t_2 - t_0}\boldsymbol{A}_2 \\
\boldsymbol{B}_2 &= \frac{t_3 - t}{t_3 - t_1}\boldsymbol{A}_2 + \frac{t - t_1}{t_3 - t_1}\boldsymbol{A}_3 \\
\boldsymbol{A}_1 &= \frac{t_1 - t}{t_1 - t_0}\boldsymbol{P}_0 + \frac{t - t_0}{t_1 - t_0}\boldsymbol{P}_1 \\
\boldsymbol{A}_2 &= \frac{t_2 - t}{t_2 - t_1}\boldsymbol{P}_1 + \frac{t - t_1}{t_2 - t_1}\boldsymbol{P}_2 \\
\boldsymbol{A}_3 &= \frac{t_3 - t}{t_3 - t_2}\boldsymbol{P}_2 + \frac{t - t_2}{t_3 - t_2}\boldsymbol{P}_3 \\
t_0 &= 0 \\
t_{i+1} &= ||\boldsymbol{P}_{i+1} - \boldsymbol{P}_i||_2^{\alpha} + t_i
\end{aligned}
$$

and

$$
\boldsymbol{\theta} = [\boldsymbol{P}_0|\boldsymbol{P}_1|\boldsymbol{P}_2|\boldsymbol{P}_3]\,.
$$

The *centripetal* Catmull-Rom spline sets α to 0.5, which has the advantage that cusps or self-intersections cannot be formed in the curve.

Algorithm 1: Polyline approximation for the closest point on a curve. This approximation breaks the curve into *segments* uniform-Δt line segments, and might be sub-optimal in areas of high curvature (if such areas were to exist, then an adaptive variant of this algorithm could instead be used).

Function `MinDistanceToCurvePolyline` ($\boldsymbol{\theta}$, C, $\boldsymbol{n}$, *segments*)

Data:
$\boldsymbol{\theta}$: curve parameters.
C: function defining coordinates of curve at a distance $0 \leq t \leq 1$ along it.
$\boldsymbol{n}$: coordinate to compute distance from.
segments: number of line segments to use in the approximation.

Result: the square of the minimum distance between $\boldsymbol{n}$ and the curve.

mindist $\leftarrow \infty$
for ($i = 1;\ i \leq$ *segments* $;\ i = i + 1$) {
  $t_0 \leftarrow (i-1)$ / *segments*
  $t_1 \leftarrow (i)$ / *segments*
  dist $\leftarrow \mathrm{d}^2_{\mathrm{seg}}(\boldsymbol{n}, C(t_0, \boldsymbol{\theta}), C(t_1, \boldsymbol{\theta}))$ `// See eq. (4)`
  if *dist* $<$ *mindist* **then**
    mindist $\leftarrow$ *dist*
return *mindist*

C. Computing the Squared Euclidean Distance Transform for a curve

In general, it is not possible to write a closed form expression for the (squared) distance of an arbitrary point, $\boldsymbol{n}$ to the closest point on a curve, $C(t)$,

$$\begin{aligned} \mathrm{d}^2_{\mathrm{cur}}(\boldsymbol{n}) = \min_t \quad & ||C(t) - \boldsymbol{n}||_2^2 \\ \text{s.t.} \quad & 0 \leq t \leq 1 \,. \end{aligned} \tag{C.1}$$

Potential approaches to computing this would for example be through a polyline approximation (see algorithm 1), a recursive brute force search (see algorithm 2) or a method based on finding the roots of the polynomial given by the derivative

$$\frac{d}{dt}||C(t) - \boldsymbol{n}||_2^2 \,. \tag{C.2}$$

In the latter case, the root-finding itself could be achieved in several ways; for example, by computing the real eigenvalues of the companion matrix formed from eq. (C.2) that lie between 0 and 1 and selecting the one that gives minimum distance, or by locating two values of t that give opposing signs of eq. (C.2) and applying the bisection method. Another potential alternative is the method proposed by Li et al. [16] which uses bisection with the Newton-Raphson method, with the initial guess computed using isolator polynomials [26].

The challenge of all the latter approaches is efficient vectorised batch implementation, whereby computation of distance transforms (the computation of the minimum distance to a curve for all points in the image space) is performed for a *batch* of curves in parallel making efficient use of many-core hardware. An approach based on root finding using the real eigenvalues of the companion matrix, for example, should ultimately prove to be more accurate than a polyline approximation, and potentially better and faster than the brute-force search, however at the time of writing there are not any hardware optimised batch generalised Eigen-decomposition (GEVD) implementations available; a batch GEVD implementation (for small matrices) is necessary as the decomposition would have to be computed for every pixel $\boldsymbol{n} \in \mathcal{I}$.

Currently, we have proof-of-concept implementations using the former polyline approximation and brute force approaches, and these are both vectorised to run on many-core (particularly GPU) hardware. The polyline approximation is in general faster (obviously both the polyline and brute force approaches allow the degree of precision to be adjusted, and that changes the computational complexity), but it does have a potential disadvantage that the approximation can introduce degeneracies whereby a small change in a curve's parameters cause a topological change in the polyline approximation. In practice, however, we have not found this to be a problem in all our experiments with handwritten characters, which all use a 10-segment polyline approximation for each curve segment that is drawn.

D. Composition functions

The *soft-or* composition operator (eq. (10)) defined in section 3.3 provides a good model of drawing with an instrument like a black ink pen, where overlapping strokes are not visible. Such a function is invariant to the order of the strokes. We might however consider alternative drawing functions that enable different effects and models of draw-

Algorithm 2: Recursive brute-force search for the closest point on a curve. This is approximate in the sense that if *slices* is too small the wrong minima might be located, and that *iters* controls the precision of the solution that is found.

Function `MinDistanceToCurveBruteForce` ($\boldsymbol{\theta}$, C, $\boldsymbol{n}$, t_{min}, t_{max}, *iters*, *slices*, *mindist*=∞)

Data:
- $\boldsymbol{\theta}$: curve parameters.
- C: function defining coordinates of curve at a distance $0 \leq t \leq 1$ along it.
- $\boldsymbol{n}$: coordinate to compute distance from.
- t_{min}: starting value of t for the search.
- t_{max}: ending value of t for the search.
- *iters*: number of iterations to perform.
- *slices*: number of intervals between t_{min} and t_{max} to compute the distance at.
- *mindist*: current minimum distance estimate.

Result: the square of the minimum distance between $\boldsymbol{n}$ and the curve.

if *iters* ≤ 0 **then**
  return *mindist*

$\Delta_t \leftarrow (t_{max} - t_{min})$ / *slices*
$t \leftarrow t_{min}$
$t_{best} \leftarrow t_{min}$

repeat
  dist $\leftarrow ||C(t, \boldsymbol{\theta}) - \boldsymbol{n}||_2^2$
  if *dist* $<$ *mindist* **then**
    mindist $\leftarrow$ *dist*
    $t_{best} \leftarrow t$
  $t \leftarrow t + \Delta_t$
until $t \geq t_{max}$;

return `MinDistanceToCurveBruteForce` ($\boldsymbol{\theta}$, C, $\boldsymbol{n}$, $t_{best} - \Delta_t$, $t_{best} + \Delta_t$, *iters*-1, *slices*, *mindist*)

ing and blending, to be achieved. Here we discuss a few potential options, including the *over* operator used for our colour drawing examples. Note the focus here is on drawing opaque *colours*; compositions for colour with transparency are discussed in appendix E.3.

D.1. The *over* composition operator

The first potential alternative approach to the soft-or would be to define a composition that respects the ordering of the images and 'paints' each stroke *over* the top of the other (whilst not allowing background 0 pixels to cover already filled pixels) from the background to the foreground. Taking inspiration from Porter and Duff [22]'s methods for alpha composition of computer graphics we could define a composition of image $\boldsymbol{A}$ painted over image $\boldsymbol{B}$ as:

$$\mathrm{c}_{\mathrm{over}}(\boldsymbol{A}, \boldsymbol{B}) = \boldsymbol{A} + \boldsymbol{B}(\mathbf{1} - \boldsymbol{A}) \,. \tag{D.1}$$

This function could then be applied recursively over a sequence of depth-ordered rasterisations to compose in the desired way:

$$\mathrm{c}_{\mathrm{over}}(\boldsymbol{I}_4, \mathrm{c}_{\mathrm{over}}(\boldsymbol{I}_3, \mathrm{c}_{\mathrm{over}}(\boldsymbol{I}^{(2)}, \boldsymbol{I}^{(1)}))) \,. \tag{D.2}$$

This type of approach does however have a significant problem in terms of implementation: because it is recursive and sequential, it is not easily vectorised and introduces a significant processing bottleneck which makes it intractable to use with large numbers of images. This problem can be circumvented by rewriting[5] eq. (D.2) as follows,

$$\mathrm{c}_{\mathrm{over}}(\boldsymbol{I}^{(1)}, \ldots, \boldsymbol{I}^{(n)}) = \sum_{i=1}^{n} \boldsymbol{I}^{(i)} \odot \prod_{j=1}^{i-1} (\mathbf{1} - \boldsymbol{I}^{(j)}) \,. \tag{D.3}$$

In this form, we can see that in essence the computation required consists of the calculation of the cumulative product of a difference, a multiplication, and a summation; all of which can be efficiently vectorised. For numerical stability, the cumulative product can be computed as the exponentiated sum of the log differences,

$$\mathrm{c}_{\mathrm{over}}(\ldots) = \sum_{i=1}^{n} \boldsymbol{I}^{(i)} \odot \exp\left(\sum_{j=1}^{i-1} \log(\mathbf{1} - \boldsymbol{I}^{(j)}) \right) \,. \tag{D.4}$$

[5] This was first noted by Sintorn and Assarsson [27] for Porter and Duff's *over* operator with an alpha channel (see also appendix E.3).

The inner summation can easily be implemented using the `cumsum` operator built into most tensor processing libraries; note, however, that standard implementations will likely include cumulative sum up to and including the i-th image, so this must then be subtracted to give the required value. Additional care must also be taken to avoid taking the logarithm of zero; in practice adding a small epsilon value suffices.

D.2. The *max* composition operator

Another possible alternative composition would be to take the per-pixel maximum over the set of images:

$$\begin{aligned} &\mathrm{c}_{\mathrm{max}_{\mathrm{i,j}}}(\boldsymbol{I}^{(1)}, \boldsymbol{I}^{(2)}, \ldots, \boldsymbol{I}^{(n)}) \\ &\quad = \max(\boldsymbol{I}^{(1)}_{i,j}, \boldsymbol{I}^{(2)}_{i,j}, \ldots, \boldsymbol{I}^{(n)}_{i,j}) \,. \end{aligned} \tag{D.5}$$

Clearly this does not have usable gradients because of the $\max$, however, a suitable differentiable relaxation exists with the $\mathrm{smoothmax}$ function,

$$\mathrm{smoothmax}(\boldsymbol{x}) = \mathrm{softmax}(\boldsymbol{x}/\tau)^\top \boldsymbol{x} \,, \tag{D.6}$$

where $\boldsymbol{x}$ is a vector of values to find the maximum of, and τ is a temperature parameter. As $\tau \to 0$, $\mathrm{smoothmax}(\boldsymbol{x}) \to \max(\boldsymbol{x})$. Equation (D.6) can be applied pixel-wise over a vector formed from the stacking of $[\boldsymbol{I}^{(1)}_{i,j}, \boldsymbol{I}^{(2)}_{i,j}, \ldots, \boldsymbol{I}^{(n)}_{i,j}]$ to form a composition function:

$$\begin{aligned} &\mathrm{c}_{\mathrm{smoothmax}_{\mathrm{i,j}}}(\boldsymbol{I}^{(1)}, \boldsymbol{I}^{(2)}, \ldots, \boldsymbol{I}^{(n)}) \\ &\quad = \mathrm{smoothmax}([\boldsymbol{I}^{(1)}_{i,j}, \boldsymbol{I}^{(2)}_{i,j}, \ldots, \boldsymbol{I}^{(n)}_{i,j}]^\top) \,. \end{aligned} \tag{D.7}$$

E. Extended Drawing

The main body of this paper focused on the act of drawing strokes in a differentiable manner and did not explore extensions to the model that would allow for more nuanced drawing — for example in colour, and with different types of stroke. We demonstrate here how the framework we have already described can be extended to allow for more control over the drawings that are produced.

E.1. Stroke width

Appendix A demonstrates that there is a direct relationship between stroke thickness and the σ parameter used by the rasterisation function. In all the experimental results shown, we used the same fixed σ for all strokes, although it should be immediately evident that this isn't a requirement, and that different strokes could have different σ values, and thus different thicknesses.

Going further, the σ value doesn't have to be a hyperparameter of the model; without changing anything within the rasterisation approach it is evident that one can compute gradients with respect to σ for every stroke that is drawn. As such, it is entirely possible to learn the line thickness of each stroke (either independently or together) by appropriately parameterising the model.

Real drawings sometimes exhibit a variation in stroke width along the length of a stroke; often this is a result of variations in pressure on the drawing instrument. It is possible to incorporate such variation into our drawing model by noting that our functions for both line segments and curves have a parameter $0 \leq t \leq 1$ along their length that can be used as an input to a function that produces different values of σ along the length of the line (or equivalently we can modify the distance map). Such a function could be parameterised by *e.g.* a simple neural network, and thus learned during the training or optimisation of a model.

E.2. Colour

Different shades of grey for individual strokes can be achieved by scalar multiplication of each stroke's raster with a grey-value before composition (note that soft-or would no longer necessarily be appropriate, so a different composition would likely be used). The grey-value could be learned or be a hyperparameter.

To rasterise full-colour strokes, the simplest approach is to replicate the image for a rasterised stroke three times in the channel dimension, and then multiply by a tuple of values corresponding to the desired red, green, and blue values. Again, the parameters can be learned, as is illustrated in fig. VII, which uses the *over* composition operation (eq. (D.4)).

If we want to rasterise lines along which the colour changes, we can follow the same methodology for changing stroke width and learn functions that emit colour as a function of the relative position, t, along the stroke.

E.3. Incorporating Transparency

The differentiable rasterisation approach for colour described above can also be extended to deal with transparency. If we assume a pre-multiplied alpha colour model, where the the red, green and blue values of a pixel represent emission, and the alpha value represents occlusion, then we can directly use Porter and Duff [22]'s compositing arithmetic. For example, the over operator with alpha,

$$\begin{aligned} c_o &= c_a + c_b(1 - \alpha_a) \\ \alpha_o &= \alpha_a + \alpha_b(1 - \alpha_a) \,, \end{aligned} \tag{E.1}$$

allows for models that can learn appropriate colour and transparency for each stroke drawn.

F. Autotracing Model Architectures

This section details the model architectures used for the autotracing autoencoders described in section 5 of the main paper.

F.1. Encoders

A series of encoder networks were used for different datasets. In each encoder network architecture, one can control the dimensionality of the latent vector encoding (`latent_size`).

MLP Encoder The encoder network used for MNIST experiments is a simple multi-layer perceptron with default `hidden` $= 64$ and `latent_size` $= 64$:

```
Linear(28*28, hidden)
ReLU()
Linear(hidden, latent_size)
ReLU()
```

CNN Encoder For resized Omniglot experiments (28×28 pixel images), a convolutional encoder network with batch normalization was used:

```
Conv2d(in_channels=1, out_channels=64,
    kernel_size=3, padding=1, stride=1)
BatchNorm2d(num_features=64)
ReLU()
Conv2d(in_channels=64, out_channels=64,
    kernel_size=3, padding=1, stride=1)
BatchNorm2d(num_features=64)
ReLU()
Conv2d(in_channels=64, out_channels=64,
    kernel_size=3, padding=1, stride=1)
BatchNorm2d(num_features=64)
ReLU()
AdaptiveAvgPool2d(output_size=8)
Flatten()
Linear(4096, latent_size)
```

StrokeNet Agent Encoder Finally, when comparing against StrokeNet [38] on scaled MNIST (256×256), we present results replicating Zheng et al.'s AgentCNN encoder[6].

```
Conv2d(in_channels=1, out_channels=16,
    kernel_size=3, padding=1)
BatchNorm2d(num_features=16)
LeakyReLU(negative_slope=0.2)
Conv2d(in_channels=16, out_channels=16,
    kernel_size=3, padding=1)
BatchNorm2d(num_features=16)
LeakyReLU(negative_slope=0.2)
AvgPool2d(kernel_size=2)

Conv2d(in_channels=16, out_channels=32,
    kernel_size=3, padding=1)
BatchNorm2d(num_features=32)
LeakyReLU(negative_slope=0.2)
Conv2d(in_channels=32, out_channels=32,
    kernel_size=3, padding=1)
BatchNorm2d(num_features=32)
LeakyReLU(negative_slope=0.2)
AvgPool2d(kernel_size=2)

Conv2d(in_channels=32, out_channels=64,
    kernel_size=3, padding=1)
BatchNorm2d(num_features=64)
LeakyReLU(negative_slope=0.2)
Conv2d(in_channels=64, out_channels=64,
    kernel_size=3, padding=1)
BatchNorm2d(num_features=64)
LeakyReLU(negative_slope=0.2)
AvgPool2d(kernel_size=2)

Conv2d(in_channels=64, out_channels=128,
    kernel_size=3, padding=1)
BatchNorm2d(num_features=128)
LeakyReLU(negative_slope=0.2)
Conv2d(in_channels=128, out_channels=128,
    kernel_size=3, padding=1)
BatchNorm2d(num_features=128)
LeakyReLU(negative_slope=0.2)
AvgPool2d(kernel_size=2)

Conv2d(in_channels=128, out_channels=256,
    kernel_size=3, padding=1)
BatchNorm2d(num_features=256)
LeakyReLU(negative_slope=0.2)
Conv2d(in_channels=256, out_channels=256,
    kernel_size=3, padding=1)
BatchNorm2d(num_features=256)
LeakyReLU(negative_slope=0.2)
AvgPool2d(kernel_size=2)

Flatten()
```

F.2. Decoders

All our decoder networks, which provide different stroke parameterisations, have a common structure consisting of two linear layers followed by ReLU non-linear activation. For 28×28 pixel MNIST and Omniglot experiments, we used `hidden1` $= 64$ and `hidden2` $= 256$.

```
Linear(latent_size, hidden1)
ReLU()
```

[6]Code available at https://github.com/vexilligera/strokenet

```
Linear(hidden1, hidden2)
ReLU()
```

This common structure is followed by a sub-network to produce stroke parameters; this is usually a single linear layer followed by a tanh function. Specific details for the chosen type of primitive parameterisation is as follows:

Line. Line decoder outputs the start and end coordinates of `nlines` segments. The default number of lines used for MNIST is 5.

```
Linear(hidden2, nlines * 4)
Tanh()
```

PolyLine. This allows us to decode the stroke data to a sequence of consecutive points (each defined as $\boldsymbol{p} = [p_x, p_y]$). Default value of `npoints` is 16, but it can be varied.

```
Linear(hidden2, npoints * 2)
Tanh()
```

PolyConnect. Similar to *PolyLine*, but instead of decoding to a sequence of consecutive points, it outputs a set of points joined together by a learned connection matrix. The network computing `npoints` 2d coordinates is the same as in *PolyLine* and the sub-network computing the upper triangular part of the connection matrix is:

```
Linear(hidden2, nlines)
Sigmoid()
```

where `nlines` is computed as

```
int((npoints ** 2 + npoints) / 2)
```

if we allow single points to be drawn (*i.e.* compute the diagonal of the connection matrix), and as follows, otherwise:

```
int(npoints * (npoints - 1) / 2)
```

All possible combinations of lines formed between the set of `npoints` are rasterised and shaded by the appropriate connection weight before composition.

CRS. Decoder that parametrises stroke data as Catmull-Rom splines with default of $\texttt{nlines} = 1$ and $\texttt{npoints} = 16$ control points:

```
Linear(hidden2, nlines * npoints * 2),
Tanh()
```

Bézier. We can also choose to parametrise strokes as Bézier curved lines (default $\texttt{nlines} = 5$) and can specify the number of `segments` (default is 1).

```
Linear(hidden2, 2 * npoints * nlines)
Tanh()
```

where the number of control points is computed based on the number of segments:

```
npoints = (4 + (segments - 1) * 3)
```

Note that this allows for a connected path which isn't necessarily smooth as each segment has independent control points. It is possible to formulate a version which is smooth and has fewer $(4 + (\texttt{segments} - 1) \times 2)$ control points.

BézierConnect. Following the same pattern as *PolyConnect*, this decodes stroke data to a set of control points for Bézier curves. The connection matrix is learned using a network as described in the paragraph *PolyConnect* and each point corresponds to both a curve's end point and corresponding control point to allow for smooth curve segments.

F.3. Recurrent Decoders

We implemented Zheng et al. [38]'s recurrent model that at each time step uses two separate CNN networks, one to encode the target image and one for the previous frame. The vector encodings are concatenated, decoded to 'stroke data' and a new stroke is rendered. The new stroke is then overlaid on the previous frame. However, this approach proved is computationally expensive and difficult to train well.

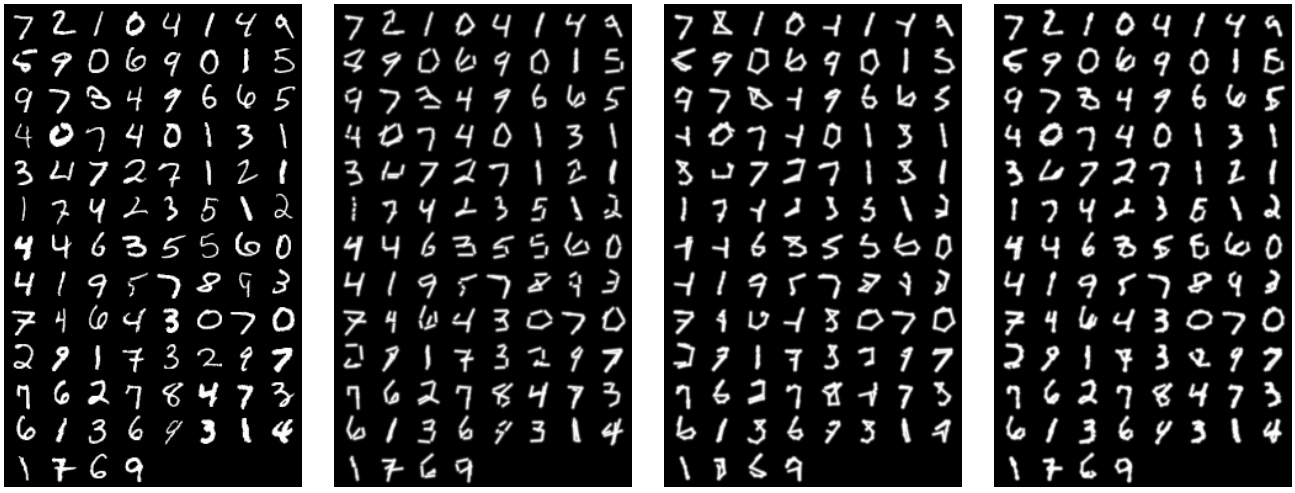
Instead, we used a GRU-based RNN which initially starts with a projection of a zeroed input and the target image's vector encoding as its initial hidden state. The RNN decoder architecture is as follows:

```
Linear(output_size, latent_size)
ReLU()
GRU(latent_size, latent_size)
Linear(latent_size, output_size)
```

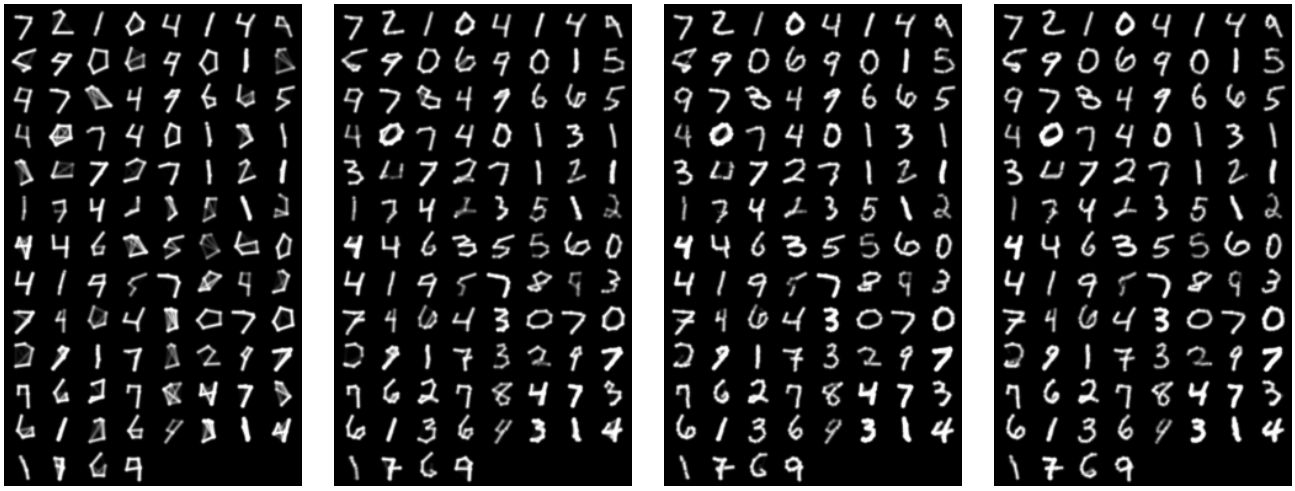
where `output_size` can be modified depending on the chosen type of primitive parametrisation (*e.g.* a Bézier curve has 4 control points, hence $\texttt{output\_size} = 8$). The GRU model is both trained and evaluated for a predefined number of time steps (3 in all our experiments), corresponding to the number of independent strokes produced.

G. MNIST Reconstructions

Figures X and XI illustrate the effect of different stroke parameterisations of the MNIST dataset. Varying the number of (L)ines, (S)egments, and (P)oints and introducing a learned connection matrix between them leads to distinct



(a) Test Samples (b) Lines(L=5) (c) PolyLine(P=8) (d) PolyLine(P=16)



(e) PolyConnect(P=5) (f) PolyConnect(P=8) (g) PolyConnect(P=16) (h) PolyConnect(P=32)

Figure X: MNIST test set samples and reconstructions using different parameterisations of 'stroke data': Lines, PolyLine (*i.e.* a series of consecutive (P)oints) and PolyConnect (a set of 2d (P)oints joined by a learned connection matrix).

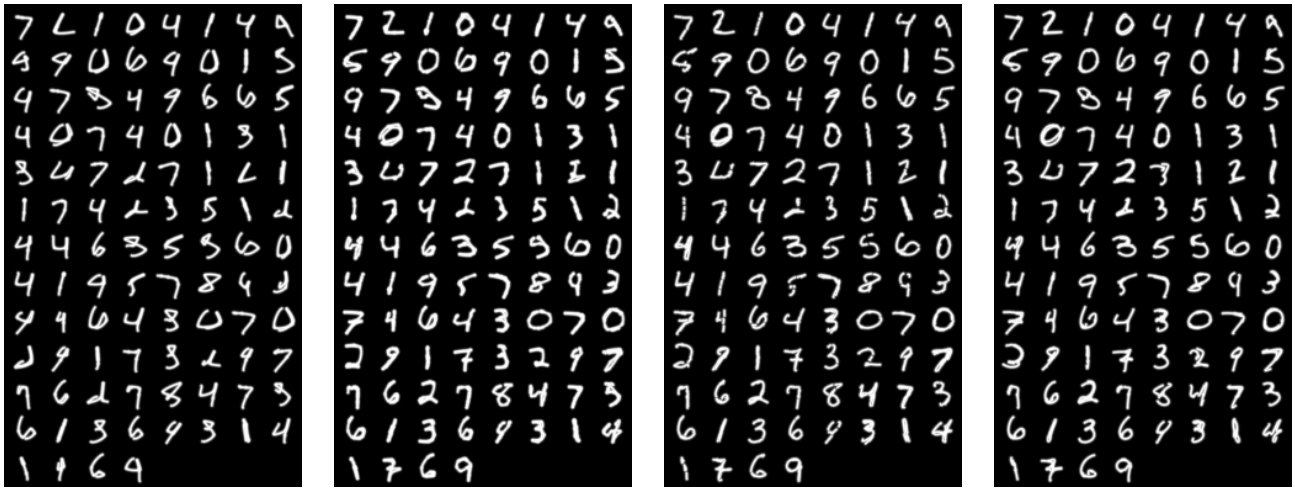
approaches to drawing. As depicted in figs. Xg and XIg, *Connect* models produce the closest reconstructions. Likewise, parameterisation using simple Bézier curves (fig. XIc) leads to convincing results.

H. Omniglot (28×28 pixels) comparison

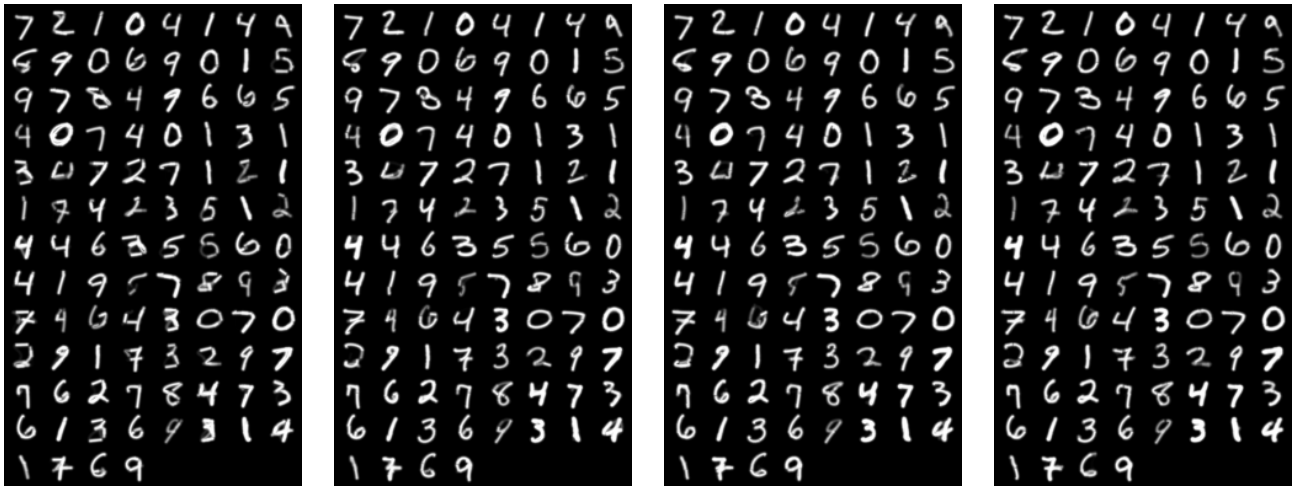
Table H.1 shows the effect of different parameterisations on Omniglot dataset. All the models demonstrate reasonable generalisation to the test dataset (as measured by MSE) even though the test alphabets are completely disjoint from the training/validation ones. Bézier curves work particularly well, although we note that they do appear to struggle with forming dots (for example in the Braille alphabet which can found in the training/validation sets).

Decoder	St	#P	#S	#L	Val	Test
Line	1	20	1	10	0.0189	0.0223
PolyConnect	1	16	-	-	0.0127	0.0151
Bézier	1	20	1	5	0.0158	0.0194
BézierConnect	1	16	-	-	0.0117	0.0144
RNNBézier	10	16	1	1	0.0152	0.0181
Bézier*	1	50	3	5	0.0091	0.0118

Table H.1: Omniglot validation and test MSE for models constructed with different parameterisations and architecture (*i.e.* recurrent vs single-(St)ep). Bézier* corresponds to the model whose reconstructions were shown in fig. 6 and has `hidden1` $= 512$ and `hidden2` $= 1024$.



(a) CRS(L=1, P=8) (b) CRS(L=1, P=16) (c) Bézier(L=5, S=1) (d) Bézier(L=2, S=2)



(e) BézierConnect(P=5) (f) BézierConnect(P=8) (g) BézierConnect(P=16) (h) BézierConnect(P=32)

Figure XI: MNIST test set reconstructions (of samples in fig. Xa) with curves parametrised as Catmull-Rom splines (CRS) and Bézier curves (Bézier and BézierConnect). In both CRS and Bézier Decoders, we can vary the number of (L)ines, (P)oints and, respectively, (S)egments. BézierConnect allows control over the (P)oints joined by the learned connection matrix.

I. StrokeNet ScaledMNIST comparison

The StrokeNet paper [38] describes an evaluation of the model on scaled-up MNIST characters by comparing performance against a CNN-based classifier trained on the scaled images, and then evaluated on the reconstructions. The paper implies that the MNIST characters were just resampled to 256x256, however from analysis of the source code it can be determined that the scaling procedure was to: resize the 28x28 characters to 120x120 using bilinear interpolation, pad the 120x120 images to 256x256, and change the contrast by multiplying pixels by 0.6. Although the original rationale for these choices is unclear, we follow exactly the same procedure for our experiments.

The structure of the classifier model in the paper is not described beyond it being convolutional with 5-layers, and no code for this aspect of the experiments was provided. We thus chose to implement our own classifier as follows:

```
Conv2d(in_channels=1, out_channels=30,
    kernel_size=5, padding=0, stride=1)
ReLU()
Conv2d(in_channels=30, out_channels=15,
    kernel_size=5, padding=0, stride=1)
ReLU()
Linear(6000, 128)
ReLU()
Linear(128, 50)
ReLU()
Linear(50, 10)
```

We did not use any form of regularisation or dropout during training. The network was trained for 10 epochs using the Adam optimiser with a learning rate of 0.001 and PyTorch's `CrossEntropyLoss` which incorporates the Softmax activation. This network performs considerably better than the results presented in the original paper on the raw scaled MNIST test dataset (originally reported accuracy is 90.82%, whereas the above network achieves 98.58%). To compute the performance of the StrokeNet paper with our classification network we take the pretrained model weights provided by the StrokeNet authors and use them to generate reconstructions of the scaled MNIST test set, which are then fed to the classifier network to make predictions from. Again we found considerably higher performance than was originally reported, as detailed in the main paper.