General Virtual Sketching Framework for Vector Line Art

HAORAN MO, Sun Yat-sen University, China
EDGAR SIMO-SERRA, Waseda University, Japan
CHENGYING GAO*, Sun Yat-sen University, China
CHANGQING ZOU, Huawei Technologies Canada, Canada
RUOMEI WANG, Sun Yat-sen University, China

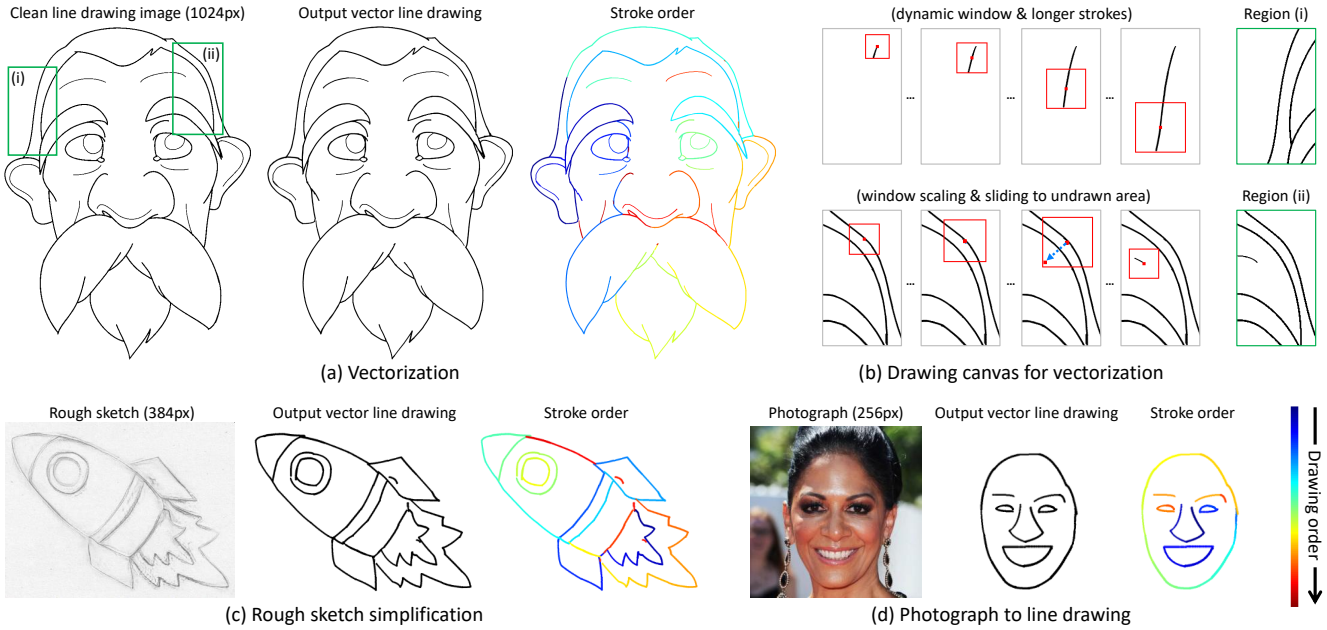

Fig. 1. Given clean line drawings, rough sketches or photographs of arbitrary resolution as input, our framework generates the corresponding vector line drawings directly. As shown in (b), the framework models a virtual pen surrounded by a dynamic window (red boxes), which moves while drawing the strokes. It learns to move around by scaling the window and sliding to an undrawn area for restarting the drawing (bottom example; sliding trajectory in blue arrow). With our proposed stroke regularization mechanism, the framework is able to enlarge the window and draw long strokes for simplicity (top example).

Vector line art plays an important role in graphic design, however, it is tedious to manually create. We introduce a general framework to produce line drawings from a wide variety of images, by learning a mapping from raster image space to vector image space. Our approach is based on a recurrent neural network that draws the lines one by one. A differentiable rasterization module allows for training with only supervised raster data. We use a dynamic window around a virtual pen while drawing lines, implemented with a proposed aligned cropping and differentiable pasting modules. Furthermore, we develop a stroke regularization loss that encourages the model to use fewer and longer strokes to simplify the resulting vector image. Ablation studies and comparisons with existing methods corroborate the efficiency of our approach which is able to generate visually better results in less computation time, while generalizing better to a diversity of images and applications.

CCS Concepts: • **Computing methodologies** → **Parametric curve and surface models**; *Neural networks*.

Additional Key Words and Phrases: vector line art generation, virtual sketching, dynamic window mechanism, stroke regularization

ACM Reference Format:
Haoran Mo, Edgar Simo-Serra, Chengying Gao, Changqing Zou, and Ruomei Wang. 2021. General Virtual Sketching Framework for Vector Line Art. *ACM Trans. Graph.* 40, 4, Article 51 (August 2021), 15 pages. https://doi.org/10.1145/3450626.3459833

*Corresponding author.

Authors' addresses: Haoran Mo, Sun Yat-sen University, Guangzhou, China, mohaor@mail2.sysu.edu.cn; Edgar Simo-Serra, Waseda University, Tokyo, Japan, ess@waseda.jp; Chengying Gao, Sun Yat-sen University, Guangzhou, China, mcsgcy@mail.sysu.edu.cn; Changqing Zou, Huawei Technologies Canada, Markham, Canada, aaronzou1125@gmail.com; Ruomei Wang, Sun Yat-sen University, Guangzhou, China, isswrm@mail.sysu.edu.cn.

1 INTRODUCTION

Vector images play a fundamental role in graphic design given that they can be rendered at arbitrary resolutions without loss of information, and are widely used in engineering design [Egiazarian

et al. 2020], 2D animation [Su et al. 2018], and 3D printing [Liu et al. 2017]. Instead of modifying the pixels directly, vector graphics are built using basic shapes that are described with few parameters, *e.g.*, control points of a bézier curve or vertices of a polygon. This allows much more natural and flexible editing in comparison with raster images, which are described by pixel values.

Line art is often represented using vector images, and in particular parametrized curves, which can be directly generated with illustration software. However, many line drawings are generated as raster images, such as scanned paper drawings, and it is necessary to convert them to vector drawings. For clean line drawings, vectorization approaches [Bessmeltsev and Solomon 2019; Favreau et al. 2016; Noris et al. 2013; Stanko et al. 2020] can directly produce vector line drawings. In most cases, it is not clean line drawings, but rough sketches that need to be cleaned up and vectorized. Image-to-image translation algorithms [Isola et al. 2017; Li et al. 2019; Simo-Serra et al. 2018a] are able to generate clean line raster drawings, although they require post-processing to convert the resulting line drawing to a vector image. We propose a framework to directly convert arbitrary input images to clean line drawings, which is applicable to a diversity of image types as shown in Fig. 1.

Our approach is based on a recurrent neural network that learns a raster image to vector image mapping directly, while not requiring vector images for training. We achieve this with a differentiable rendering module, which is able to render line drawing raster images from a vector parametrization, and is amenable to integration in an end-to-end training framework. To overcome the issue of processing images of arbitrary resolution, which is a challenge for most learning-based vector graphics generation algorithms [Huang et al. 2019; Kim et al. 2018; Zheng et al. 2019], we propose modeling a virtual pen using a dynamic window that updates its position and size at every frame as shown in Fig. 1-(b). The model learns to enlarge the window when moving the pen around and drawing long strokes, while it shrinks the window when drawing fine details without any direct supervision. Naive implementations of cropping and pasting operations necessary for the window suffer either from non-differentiability or quantization artifacts. We overcome both issues by using an *aligned cropping* and *differentiable pasting module* that both avoid discretization and allow for gradient propagation.

Another important aspect of vector images is that while many different vector images can be rendered to the same raster image, in general, we are interested in the simplest vector representation, that is, the one with the fewest parameters. For line drawings, this consists of representing a long stroke with a single parametrized curve instead of multiple shorter curves. To integrate this concept into our model, we propose a *stroke regularization mechanism* which encourages the model to use the minimum vector parameters necessary to represent a line drawing.

We evaluate for our approach through comprehensive ablation studies, and comparisons with the existing methods on line drawing vectorization, rough sketch simplification and photograph to line drawing. These experiments demonstrate that our approach is able to generate visually pleasing results while taking less computation time, and generalizes better to different types of images [1] .

[1]The source code can be found at https://github.com/MarkMoHR/virtual_sketching.

The main contributions of this work are summarized as follows:

(1) A general framework for vector line drawing generation that works with a wide variety of images, dependent exclusively on raster training data.
(2) A dynamic window mechanism that allows processing images of arbitrary resolution and high complexity.
(3) Stroke regularization mechanism that controls the simplicity of the output vector images.
(4) In depth comparison with existing approaches in a diversity of tasks.

2 RELATED WORK

2.1 Vector Graphics Generation

There are two main lines of work on learning-based vector graphics generation: data-driven or independent on vector training data. A number of works on learning with vector training data have been proposed in recent years, *e.g.*, sketch reconstruction [Das et al. 2020; Graves 2013; Ha and Eck 2018], and image-based drawing generation [Egiazarian et al. 2020; Song et al. 2018]. While, in general, it is more straightforward and easier to learn with direct vector supervision, it is not always feasible to collect vector training data. To avoid this issue, another line aims to get around the vector data. They firstly transform the predicted vector parameters into raster drawing images, and then optimize the model at raster level. The transformation is achieved by using an external black-box rendering simulator [Ganin et al. 2018; Mellor et al. 2019], or a differentiable rendering module [Huang et al. 2019; Li et al. 2020; Nakano 2019; Zheng et al. 2019].

Among works that do not require training vector data, Learning-To-Paint [Huang et al. 2019] is the approach closest to our framework, albeit with some noticeable differences. First, we adopt a continuous stroke representation instead of a discrete one for more natural stroke continuousness, and we use a virtual pen which can avoid the redundant drawing problem in Learning-To-Paint. Second, Learning-To-Paint is limited to a small fixed image size, while ours handles images of arbitrary resolution.

2.2 Vectorization

Vectorization approaches can be divided into two lines: optimization-based and learning-based approaches. Optimization-based algorithms have been widely studied and are still under active development [Bessmeltsev and Solomon 2019; Favreau et al. 2016; Noris et al. 2013; Stanko et al. 2020]. Due to the high computational complexity of the optimization process, these approaches take a long time to generate the vector images. In contrast, our approach is faster. Recently learning-based approaches have also been proposed as an alternative. VectorNet [Kim et al. 2018] combines neural networks and optimization algorithms to segment the raster image into a set of paths, and then uses the existing vectorization techniques, *e.g.*, Potrace, to vectorize each path. Guo *et al.* [2019] use neural networks to subdivide the lines and reconstruct the topology for each junction. Strokes are then traced by curve least-square fitting method. The learning of both these works lies in the pixel level rather than the vectorization stage. Thus, they require third-party vectorization techniques in contrast to our work. Furthermore, both

lines of works are designed for clean line drawings and have difficulties in sketches with rough textures. On the other hand, our method can generalize to a wide variety of images.

2.3 Line Generation from Other Domains

Line drawings can also be generated from images in other domains, such as rough sketches and photographs. For rough sketches (*a.k.a.* sketch simplification or sketch clean-up), ClosureAware [Liu et al. 2015] and StrokeAggregator [Liu et al. 2018b] use vector images as input and output the clean vector sketches. Models in [Simo-Serra et al. 2018a,b, 2016; Xu et al. 2019] work with raster rough sketches and output the clean ones in raster format. Line extraction from photographs includes widely-known edge detection techniques [Xie and Tu 2015], and GAN-based image translation methods such as Photo-Sketching [Li et al. 2019]. All these approaches consist of either vector to vector or pixel to pixel mappings. In contrast, our approach is able to learn a pixel to vector mapping directly.

2.4 Image and Feature Sampling

In object detection and instance segmentation [Girshick 2015; He et al. 2017], image or feature sampling based on the Region-of-Interests (RoIs) with floating point number position and size is fundamental. *RoIPool* proposed in [Girshick 2015] performs quantization for the RoIs twice to extract fixed-size feature maps. However, this introduces misalignment and breaks the gradients. *RoIWarp* proposed in [Dai et al. 2016] and *RoIAlign* in Mask R-CNN [He et al. 2017] try to resolve these problems with *RoIAlign* performing better in practice. *RoIAlign* subdivides each RoI into spatial bins and performs bilinear feature interpolation within each bin to compute the feature maps. This operation avoids quantization, and thus permits gradient propagation. We employ this approach in our aligned cropping operation that requires accurate alignment and our differentiable pasting module that needs to preserve the gradients for the position and window size.

3 LINE DRAWING GENERATION FRAMEWORK

3.1 Overview

Our approach is a general framework for vector line drawing generation given an arbitrary image as input. As illustrated in Fig. 2, the framework is based on a recurrent neural network-based model which predicts the drawing strokes step by step based from the input image. It is designed with a dynamic window of a square that moves around while drawing the lines. This window, with size W a cursor Q (*i.e.*, the position), is able to move and scale after each time step. There are two advantages of using a dynamic window. First, it allows us to model images directly at full resolution, *i.e.*, the resolution of the input image, while avoiding the sharp increase in training difficulty. Second, it enables our model to scale up to an arbitrary resolution even though the model is trained on low-resolution images.

As shown in Fig. 2-(a), the recurrent model consists of four main phases at each time step t:

(1) *Aligned Cropping:* given an image I with size W_I and a same-size canvas C_{t-1} as inputs, this module crops the patches according to the current window (Q_{t-1}, W_{t-1}), and resamples them to images of fixed size W_r.

(2) *Stroke Generation:* taking the cropped patches as input, the stroke generator (Fig. 2-(b)) predicts the parameters a_t of the next stroke. The generator consists of a Convolutional Neural Network (CNN) encoder which models the image-level information, and a Recurrent Neural Network (RNN) decoder which takes in the image features and outputs the stroke parameters. The RNN decoder also receives a latent vector from previous time step and passes along a new latent vector to the next time step.

(3) *Differentiable Rendering:* a neural renderer is then employed to approximate the stroke image S_t based on the Bézier curve stroke parameters q_t derived from the predicted parameters a_t while being fully differentiable. This enables raster-level supervision during an end-to-end training without the requirement of paired vector images that are not trivial to collect.

(4) *Differentiable Pasting:* the rendered stroke image in a fixed rendering size W_r is then pasted to the full-resolution canvas based on cursor Q_{t-1} and window size W_{t-1}. This pasting process is done in a differentiable manner, which enables the gradients to be propagated to the Q_{t-1} and W_{t-1}.

Afterwards, a predicted indicator p_t is used to decide whether a stroke is drawn or not. Each pasted stroke image to be drawn is inserted into the canvas to form a full-resolution line drawing image for the comparison with the target image.

3.2 Stroke Generation

3.2.1 Stroke Representation. In our framework, strokes are represented in a relative manner, *i.e.*, continuous strokes, which is similar to the stroke-3 format $(\Delta x, \Delta y, p)$ in Sketch-RNN [Ha and Eck 2018], where $(\Delta x, \Delta y) \in [-1, +1]^2$ is the offset of the next position and $p \in [0, 1]$ is the pen state to control whether to lift the pen. Given that such representation is limited to straight lines and fixed line thickness, we augment it with an intermediate control point $(x_c, y_c) \in [-1, +1]^2$ to form a quadratic Bézier curve, and a width factor $w \in [0, 1]$ to provide varying thickness. We also add a scaling factor $\Delta s \in [0, k]$ $(k > 1)$ into the stroke representation which controls the window size at each time step, allowing the model to learn the best window size for different situations. In summary, the stroke a_t at time step t within a coordinate system $[-1, +1]$ is formulated as follows:

$$a_t = (x_c, y_c, \Delta x, \Delta y, w, \Delta s, p)_t, \quad t = 1, 2, ..., T. \tag{1}$$

A quadratic Bézier curve specified by three control points $P_0 = (x_0, y_0)$, $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ is formulated as:

$$B(\tau) = (1-\tau)^2 P_0 + 2(1-\tau)\tau P_1 + \tau^2 P_2, \quad \tau \in [0, 1]. \tag{2}$$

Following the parameter design in [Huang et al. 2019], we define the stroke parameters $q_t = (x_0, y_0, x_1, y_1, x_2, y_2, r_0, r_2)_t$ for a quadratic Bézier curve based on a_t:

$$(x_0, y_0)_t = (0, 0),\ (x_1, y_1)_t = (x_c, y_c)_t,\ (x_2, y_2)_t = (\Delta x, \Delta y)_t,$$
$$(r_0)_t = w_{t-1} \text{ and } (r_2)_t = w_t. \tag{3}$$

Here, the starting control point (x_0, y_0) should always be exactly at the middle of the dynamic window, (x_1, y_1) and (x_2, y_2) are derived from the stroke a_t, and r_0 and r_2 denote the widths of P_0 and P_2.

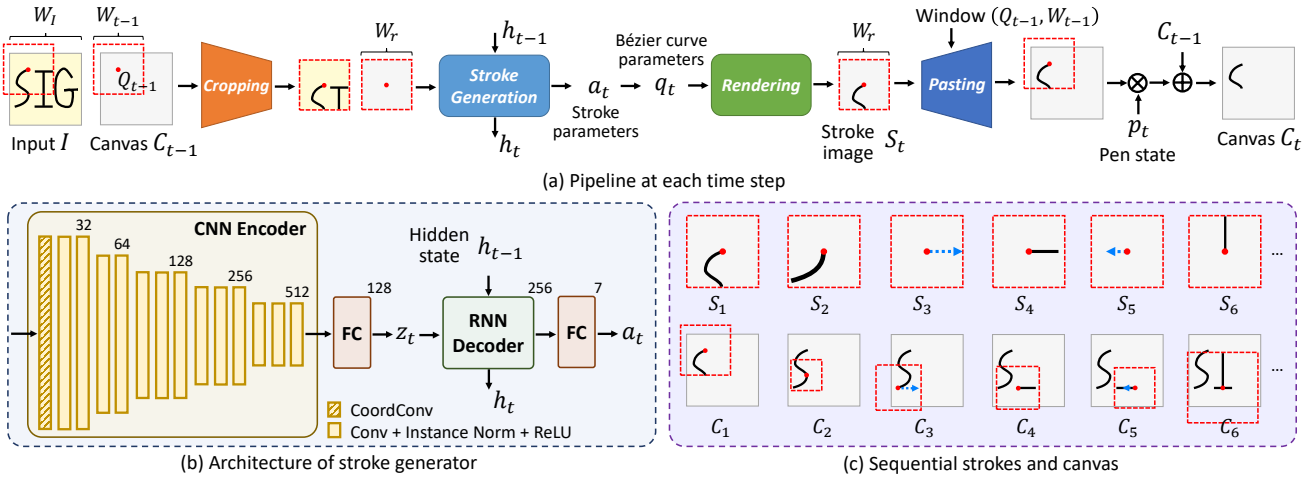

Fig. 2. Our framework generates the parametrized strokes step by step in a recurrent manner. It uses a dynamic window (dashed red boxes) around a virtual pen to draw the strokes, and can both move and change the size of the window. (a) Four main modules at each time step: aligned cropping, stroke generation, differentiable rendering and differentiable pasting. (b) Architecture of the stroke generation module. (c) Structural strokes predicted at each step; movement only is illustrated by blue arrows during which no stroke is drawn on the canvas.

After pasting, the pen state p is used to decide whether this stroke is drawn or not.

Relative Moving and Scaling. At each time step, the virtual pen along with the dynamic window moves to the ending position of the predicted stroke. If it is not necessary for a stroke to be done, *e.g.*, sliding to a different part of the image, the model can choose not to draw the stroke by setting the pen state to $p_t = 0$. We call this "movement only". Furthermore, the model can choose to change the size of the dynamic window at each time step by setting Δs_t, which allows enlarging or shrinking the window size. Given the cursor movement $\Delta Q_t = (\Delta x, \Delta y)_t \in [-1, +1]^2$ and scaling factor Δs_t, we can define the dynamic window update rule as:

$$\begin{aligned} \widehat{Q_t} &= \Delta Q_t \times W_{t-1}/2 + Q_{t-1}, \quad & Q_t &= \max(0, \min(W_I, \widehat{Q_t})), \\ \widehat{W_t} &= \Delta s_t \times W_{t-1}, & W_t &= \max(W_{min}, \min(W_I, \widehat{W_t})), \end{aligned} \tag{4}$$

where W_I is the size of the input image and W_{min} is the pre-defined minimum value for the dynamic window size. Value clipping is used to avoid potential out-of-bounds issues. In the experiments, we set $W_{min} = 32 \times 32$, the initial values $W_0 = 128 \times 128$, Q_0 a random position and $k = 2$ for Δs. In fact, an endpoint is the junction of two adjacent strokes belonging to two windows of different sizes, so its width factor value used in the previous window should also update to adapt to the next window at every time step. Please refer to the supplemental materials for in-depth details.

Differentiable Pen State Binarizing. The pen state $p \in [0, 1]$ is a continuous value during training, but is expected to be a discrete binary value with 0 corresponding to movement only and 1 corresponding to drawing a stroke. Direct discretization with a non-differentiable operation like *argmax* does not allow gradient propagation. To address this problem, we adopt *softargmax* [Luvizon et al. 2018] to approximate the argmax function while preserving the gradients. This differentiable operation can be formulated as:

$$\text{softargmax}(x) = \sum_i \frac{e^{\beta x_i}}{\sum_j e^{\beta x_j}} i. \tag{5}$$

Essentially, the *softargmax* operation outputs a continuous value close to the discrete index value produced by *argmax*. In our case, *softargmax* pushes pen state $p \in [0, 1]$ closer to 0 or 1 when applied to a vector $[1 - p, p]$ that has index values 0 and 1 only. We set $\beta = 10$ in our experiments.

3.2.2 Stroke Generator. Figure 2-(b) shows the architecture of the stroke generator, which consists of a CNN encoder and a RNN decoder built with LSTM cells [Hochreiter and Schmidhuber 1997]. The CNN encoder takes the patches cropped from the input image and canvas as input. Due to the dynamic window-based design, the model tends to get into a situation where the cropped patch has been fully drawn, as shown in the case in Fig. 7. To facilitate the learning of moving to an undrawn region outside the window, we resize the entire image and canvas to $W_r = 128 \times 128$, *i.e.*, the same size as the cropped patches, and feed them to the generator as additional global guidance that tells the model where undrawn strokes may be.

The CNN encoder employs CoordConv [Liu et al. 2018a] at the first layer, followed by convolution layers with instance normalization [Ulyanov et al. 2016] and ReLU activation function. A fully connected (FC) layer is then employed to project the image features to image embedding z_t. The RNN decoder takes z_t and the previous hidden state h_{t-1} as inputs and predicts the stroke parameters a_t, and a new hidden state h_t.

3.3 Aligned Cropping and Differentiable Pasting

The cropping and pasting modules play important roles in the dynamic window-based framework. As illustrated in Fig. 2-(a), at time step t, the cropping and pasting are done based on cursor Q_{t-1} and

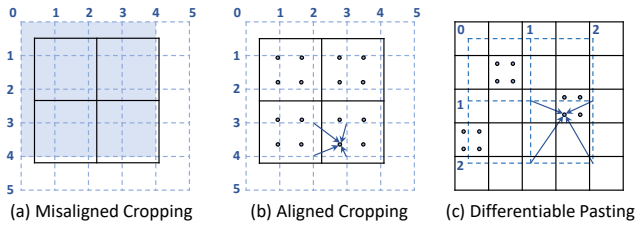

Fig. 3. Different types of cropping and pasting operations. (a) The image represented in a dashed grid is cropped by a window in black solid lines with a cursor and window size that are not aligned with pixels. Naive cropping operation with a quantized discrete window produces a misaligned patch in the light blue area. In pasting phase, this area is padded with pixels in white to form the pasted frame. (b) The aligned cropping works on a window that is not necessarily aligned with pixels. It relies on RoIAlign operation [He et al. 2017], which resamples the cropped patch into a fixed size W_r (2×2 in this example). (c) The differentiable pasting essentially has a similar working mechanism to the aligned cropping operation except for requiring a coordinate system change. Here the black solid lines indicate the cropping window, and the dashed grid indicates the rendered patch to be cropped. The pasted frame within the black solid lines is amenable to differentiation. Dots in the bins indicate the sampling points for RoIAlign. We omit dots in the other bins for brevity.

window size W_{t-1} both of which are floating values. Quantizing the floating numbers (Fig. 3-(a)) is a straightforward way to perform spatial cropping and pasting, but this could lead to two problems: the window and the cropped or pasted patches may not be aligned, and the gradients from the pasted canvas cannot be propagated to Q_{t-1} and W_{t-1}.

To address the misalignment problem in the cropping phase, we introduce an aligned cropping operator, which is able to cope with the cursor and window size that are not necessarily aligned with pixels. This operation is based on the RoIAlign module proposed in [He et al. 2017]. As shown in Fig. 3-(b), this operation first subdivides the window into spatial bins based on the window size W_{t-1} and resampling size $W_r = 128 \times 128$. Then, inside each bin, several sampling points are set and their values are computed by bilinear image interpolation. The average of these values is set to the final value of each bin finally. Afterwards, cropped patches in a fixed size W_r are obtained without the need for quantization, and they are spatially aligned with the window.

In the pasting phase, to guarantee alignment and enable gradients propagation to the cursor and window size, we introduce a differentiable pasting module. As shown in Fig. 3, like the aligned cropping, the differentiable pasting operation is based on bilinear image interpolation. The main different between both modules is that the 2D interpolations are under different coordinate systems, which requires a coordinate system change. After performing the coordinate change, the bilinear image interpolation operation under continuous value space enables differentiation. We introduce details of the coordinate system change in the supplemental materials.

3.4 Differentiable Rendering

The differentiable renderer is to render the 2D stroke raster image from the 1D stroke vector parameters, which facilitates an end-to-end training with a raster-level loss and avoids the requirement of vector training images. To enable gradients to be propagated from the rendered output to the stroke parameters, we follow a similar approach to Learning-To-Paint [Huang et al. 2019] and use a neural network to approximate the stroke image S_t given the quadratic Bézier curve parameters q_t mentioned in §3.2.1. The neural renderer has a similar architecture as the one in Learning-To-Paint. In our experiments, the rendering window size is set to 128×128, which is equal to the fixed image size W_r in the stroke generator. Please refer to the supplemental materials for more details about the neural renderer.

4 TRAINING

4.1 Overall Loss Function

Our training loss function is made up of three components: (1) a raster loss $\mathcal{L}_{ras}$ for visual supervision, (2) an out-of-bounds penalty loss $\mathcal{L}_{out}$ to avoid out-of-bounds issues of the stroke parameters with relative moving and scaling, and (3) a stroke regularization loss $\mathcal{L}_{reg}$ that can encourage the model to simplify the resulting stroke vector images. The total loss is formulated as below:

$$\mathcal{L}_{total} = \mathcal{L}_{ras} + \lambda_{out}\mathcal{L}_{out} + \lambda_{reg}\mathcal{L}_{reg}, \tag{6}$$

where λ_{out} and λ_{reg} are scalars.

4.2 Raster-level Supervision

The differentiable rendering module allows us to adopt a raster-level loss for end-to-end training without the need for vector images. It is straightforward to use an L_1 or L_2 loss to calculate the pixel-wise difference between the target line drawing image and the rendered output. However, our experiments in §5.5 show that they are not suitable for our task because they focus largely on the local details and this results in poor global completeness.

To this end, we seek for a raster loss which is able to guarantee both details and completeness of line drawings. Inspired by the study of perceptual similarity [Zhang et al. 2018], we employ the perceptual difference [Johnson et al. 2016], a kind of structural loss, as a raster-level loss. In particular, we use VGG-16 [Simonyan and Zisserman 2015] as the perceptual model, and we fine-tune it on a sketch dataset QuickDraw [Ha and Eck 2018] to make it more sensitive to line drawings.

Given a rendered line drawing image $\hat{y}$ (the last canvas), a target line drawing image y, and a perceptual network ϕ, we define $\phi_j(\cdot)$ as the activation map $\in \mathbb{R}^{D_j \times H_j \times W_j}$ of layer j. The perceptual loss of layer j can be defined as:

$$\mathcal{L}^j_{perc} = \frac{1}{D_j \times H_j \times W_j} \left\| \phi_j(\hat{y}) - \phi_j(y) \right\|_1. \tag{7}$$

Loss Value Normalization. As a raster loss, we use a combination of perceptual losses from a set of layers J. However, as loss values from different layers can have different orders of magnitude, directly summing the original loss values as in [Johnson et al. 2016] might lead to an undesired imbalance of layers. Furthermore, a simple

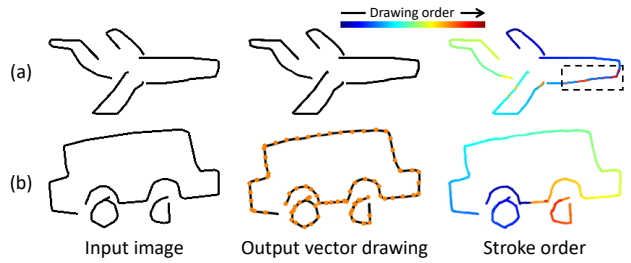

Fig. 4. Example of redundancy and incompactness. (a) The redundant strokes (shown in the black box) are overlapped with the drawn ones. (b) Orange dots indicate the endpoints of each stroke. The result is that a straight line is divided into many short segments.

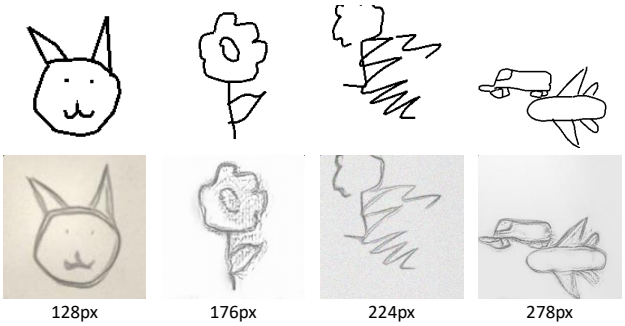

Fig. 5. Training examples for vectorization and rough sketch simplification.

weighted summation is not practical because it is hard to decide the best weights. To address this problem, we normalize the loss value for each layer. Practically, we divide the loss value of layer j by the mean loss calculated from all the previous training iterations, and then obtain the normalized perceptual loss $\mathcal{L}^j_{perc-norm}$. Finally, our raster loss is computed as:

$$\mathcal{L}_{ras} = \sum_{j \in J} \mathcal{L}^j_{perc-norm}. \tag{8}$$

4.3 Out-of-Bounds Penalty

The stroke offset $(\Delta x, \Delta y)$ and scaling factor Δs can be theoretically learnt from only the raster loss. However, we notice the out-of-bounds issue caused by the relative moving and scaling, as mentioned in §3.2.1 and Eq. (4), should also be penalized in order to better teach the model to predict the relative values inside the boundary.

Given the original values of cursor $\widehat{Q_t}$ and window size $\widehat{W_t}$ after relative moving and scaling, and the clipping values Q_t and W_t in Eq. (4), we then define the out-of-bounds penalty loss for moving factors by directly penalizing the out-of-bounds distance as:

$$\mathcal{L}^{moving}_{out} = \frac{1}{T} \sum_{t=1}^{T} \left| Q_t - \widehat{Q_t} \right|. \tag{9}$$

The penalty for the scaling factor is then formulated as the normalized outer distance to the upper (W_I) and bottom (W_{min}) bounds:

$$\begin{aligned} \mathcal{L}^{up} &= \max(\widehat{W_t} - W_I, 0)/W_I, \\ \mathcal{L}^{bottom} &= \max(W_{min} - \widehat{W_t}, 0)/W_{min}, \\ \mathcal{L}^{scaling}_{out} &= \frac{1}{T} \sum_{t=1}^{T} (\mathcal{L}^{up} + \mathcal{L}^{bottom}). \end{aligned} \tag{10}$$

The total out-of-bounds penalty $\mathcal{L}_{out}$ is the combination of losses for moving and scaling factors. That is, $\mathcal{L}_{out} = \mathcal{L}^{moving}_{out} + \mathcal{L}^{scaling}_{out}$. We show the effectiveness of this loss in the supplemental materials.

4.4 Stroke Regularization Mechanism

Any raster image can be rendered by different, yet visually equivalent, vector images. While they may be visually equivalent, the vector images can have varying degrees of complexity. Given that our approach is trained with a raster loss, we need some mechanism to encourage the vector representation to be as simple as possible, otherwise redundant strokes may appear as shown in Fig. 4-(a), or long strokes may be represented by many shorter strokes as in Fig. 4-(b). In graphic design, both redundancy and incompactness increase the difficulty of editing and we desire the simplest representation of the vector images.

The higher the simplicity, the fewer vector parameters are necessary to represent a line drawing. To this end, we introduce a *stroke regularization mechanism* by restricting the number of strokes that are related to the pen state $p_t \in [0, 1]$ (1 for drawing and 0 for lifting). The stroke regularization term is formulated as the proportion of drawn strokes:

$$\mathcal{L}_{reg} = \frac{1}{T} \sum_{t=1}^{T} p_t. \tag{11}$$

This term is differentiable and added to the total loss function $\mathcal{L}_{total}$ in Eq. (6) during end-to-end training. When minimizing the total loss, ideally the model will learn to use fewer strokes while producing a better line drawing simultaneously. As a result, the redundant and incompact strokes can be avoided. As with most regularization terms, weighting the stroke regularization term too strong can lead the model to use an insufficient number of strokes and thus worse results. We analyze the effect of this term in §5.6.

5 EXPERIMENTS

We evaluate our approach in a diversity of image-to-line drawing tasks such as line drawing vectorization, rough sketch simplification, and photograph to line drawing to show the generalness of our approach.

5.1 Dataset and Implementation Details

Dataset. Our model is able to process images of any resolution due to our dynamic window-based framework. For training efficiency, we train the model with low-resolution images, and then evaluate its performance on higher resolution images. For the evaluation of vectorization and rough sketch simplification, we use the QuickDraw [Ha and Eck 2018] dataset that contains vector stroke data, and render them as raster images of varying resolutions from 128 px to 278 px as shown in the top row of Fig. 5. In the case of rough sketch simplification, we utilize the pencil art generation and rough augmentation techniques from [Simo-Serra et al. 2018a,b] to synthesize

the corresponding rough sketches (bottom row in Fig. 5) from the clean line drawings. We additionally evaluate for photograph to line drawing task using the face image dataset CelebAMask-HQ [Lee et al. 2020], where we can generate the facial sketches from the annotated segmentation masks. The face images and sketches are rendered at a resolution of 256 px.

We also collect separate test sets for quantitative evaluation. In the case of vectorization and rough sketch simplification, we generate examples from the test set of QuickDraw. Raster images are rendered at four sizes: 128, 256, 384 and 512. Such test sets can be used to evaluate the generalization ability on higher resolutions. For photograph to line drawing task, we use the test split in CelebAMask-HQ.

Training Details. For vectorization task, we compute perceptual loss at layers `relu1_2`, `relu2_2`, `relu3_3` and `relu5_1` (short for $\cup(12, 22, 33, 51)$) of the VGG-16 model. For rough sketch simplification and photograph to line drawing, we adopt $\cup(22, 33, 51)$ and $\cup(22, 33, 42, 51)$, respectively. The loss weight λ_{out} in Eq. (6) is set to 10. For the stroke regularization mechanism, we adopt a linearly increasing loss weight λ_{reg} instead of a constant value. We resize the images using area interpolation when inputting the full image to the CNN to provide global guidance. We train for $75k/90k/90k$ iterations for the three tasks, respectively, with a batch size 20. Optimization is done with Adam [Kingma and Ba 2014] using an initial learning rate *1e-4*. The maximum number of time steps of the recurrent neural network during training is set to 48.

Testing Details. The testing is done step by step in a fully automatic manner, with the cropping phase, the stroke generation, the explicit rasterization and the pasting phase performed sequentially at each step as in training. When evaluating on images of high resolutions and complexity, we are able to theoretically use an infinite number of strokes because our model learns to lift the pen with the pen state. To avoid the prediction of unnecessary pen lifting after the model finishes drawing early, we design an early-stop strategy. Specifically, we set N_{round} rounds of drawing, each with a maximum number of strokes N_{stroke}. In each round, the moving is done automatically. When N_{break} continuous pen states corresponding to lifting the pen ($p = 0$) are predicted, we end the round. After each round, we randomly move the pen to slide the window to a distant undrawn area, which is especially useful with high-resolution images. We do this by randomly moving the cursor to a position far away from the drawn region. This strategy maximizes the completeness of the resulting vectorization, and manual intervention, *i.e.*, random movement, is performed only when the model cannot draw anymore inside a round.

We set $N_{break} = 12$ for the three tasks. The values of $N_{round} = 10/10/1$ and $N_{stroke} = 500/96/100$ are found to work well for vectorization, rough sketch simplification and photograph to line drawing, respectively. For the vectorization task, we also design a method to stop the drawing early, *i.e.*, before reaching the maximum round. Please refer to the supplemental materials for more details of this method, as well as the datasets and the implementation.

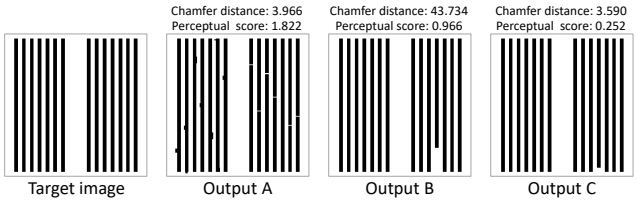

Fig. 6. Examples to show how the quantitative evaluation metrics work. Chamfer distances (↓) in 10^{-5} and perceptual scores (↓) in 10^{-2}. For both metrics, a lower value is better.

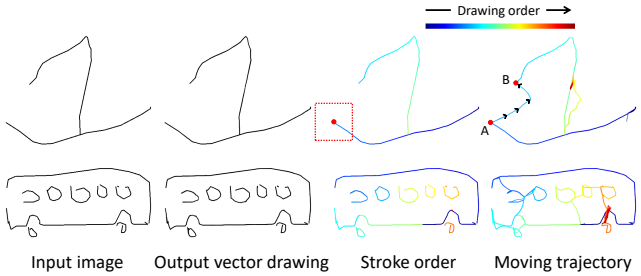

Fig. 7. Results of moving. The last column shows the drawing order of the strokes together with the trajectory of movement only that is not drawn in the final output. The dashed red box represents a window without undrawn pixel inside it. Black arrows indicate the moving direction.

5.2 Quantitative Evaluation Metrics

Our goal of quantitative evaluation is to measure the pixel-level similarity between target line drawing image and rendered output. This still remains an open problem as it is hard to define similarity metrics which are fully consistent with human perception. Chamfer distance [Fan et al. 2017] adopted in [Yan et al. 2020] is able to measure sketch-to-sketch similarity effectively, however, we notice some issues of this metrics. As highlighted in Fig. 6, output A has a lower chamfer distance value than B, but the details in A are visually worse.

From a rough inspection, we notice chamfer distance is more sensitive to the *completeness* of the drawing, but not to fine-grained details. However, fine-grained details play a fundamental role in line drawings. We notice the perceptual score derived from the perceptual loss in §4.2 is more sensitive to the *details*, and employ it as another metric. As shown in Fig. 6, output A has a lower chamfer distance corresponding to higher completeness, but a higher perceptual score corresponding to worse details compared with B. Output C is visually more complete than B, so both chamfer distance and perceptual score are lower.

Different from the perceptual loss during training (§4.2), there is no history value to perform loss normalization during testing. Therefore, we use only the middle layer `relu3_3` in the VGG-16 to calculate the perceptual score. For test sets with images of different resolutions, we average the values.

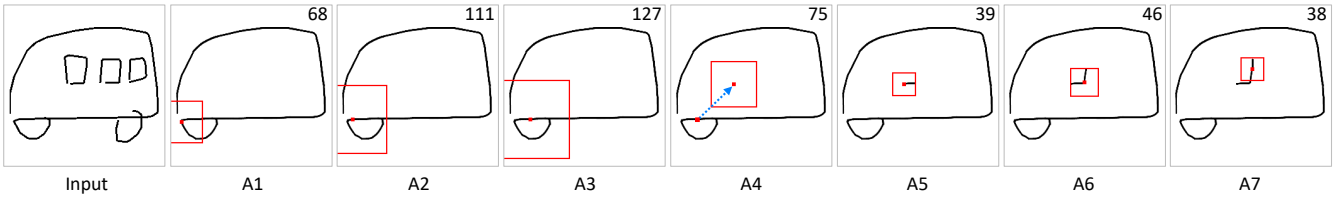

Fig. 8. Results of window scaling. Red boxes represent the windows. A1 to A7 are the sequential frames. Numbers on top right corner indicate the window size. We can see in A1 to A3 how the model enlarges the dynamic window in order to find an undrawn area to move to it. A blue arrow highlights in A4 shows a movement that does not draw a line.

5.3 Effectiveness of Moving and Scaling

Moving. Our model learns to move the virtual pen along with the dynamic window around by breaking the continuous strokes or sliding to an undrawn area with pen state $p = 0$. Figure 7 shows two representative examples. The top row demonstrates an easily encountered case: the window moves to where no undrawn pixel is inside. Next, it slides while not drawing from position A in the upper right direction to position B containing undrawn pixels and then the model restarts drawing. Bottom row shows a number of necessary breakings among the isolated strokes.

Scaling. Despite the lack of ground-truth scaling values for direct supervision, the model does learn some common and meaningful rules as shown in Fig. 8. Most of the time, the model uses small windows to draw fine details. However, when getting into the situation where the window does not contain undrawn pixels (A1), the model attempts to enlarge the window to find the undrawn pixels (A1-A3), and slides there with a long stride simultaneously (A3-A4). Afterwards, the model shrinks the window quickly to restart drawing (A5). These results imply that the model is capable of learning to choose a reasonable window size to adapt to different situations.

We also compare our model with one that uses a fixed-size window. For a fair comparison, we use the same neural render with rendering size set to 128 px, and set the fixed size to 128 px. Quantitative results in Table 1 show that such a model ("**Fixed**" in "Window" column) has worse performance for both metrics. We believe that this is probably because the 128 px window size is too large for this model to draw details well, after looking at the statistics of window size for our model. Most windows used for drawing strokes have sizes ranging from 30 px to 70 px at both low and high resolutions. It seems like smaller windows can draw shorter strokes and help to recover details. However, overusing small windows tends to cause worse completeness, given that we allow only a finite number of strokes during training. This also explains why our model learns to use a larger window for movement only. Visual results and distributions of window sizes are in the supplemental materials.

5.4 Effectiveness of Differentiable Pasting

As discussed in §3.3, misaligned cropping and non-differentiable pasting suffer from misalignment and non-differentiability. The latter issue in pasting phase results in the loss of gradient with respect to stroke position and scaling factor in the stroke parameters, and is detrimental to end-to-end training. We thus mainly study the effectiveness of differentiable pasting.

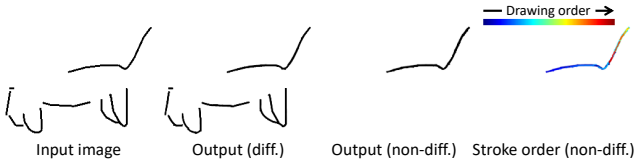

Fig. 9. Comparisons between differentiable ("diff.") and non-differentiable ("non-diff.") pasting.

We compute statistics for the distributions of scaling factor values, and find that for both low and high resolutions, the model with differentiable pasting predicts scaling factors concentrated in the interval $[0.8, 1.2]$. However, without differentiable pasting, there appears to be an abnormal distribution with peaks around 0.5 and 2.0. Visual results show the scaling values change alternately between ~ 0.5 and ~ 2.0, leading to severe jitter in the window size rather than meaningful changes as in the differentiable-pasting-based model. The abnormal scaling factors are due to the failure in its learning without normal gradient propagation. Please refer to the supplemental materials for the scaling value distributions and visual results.

Figure 9 shows visual comparisons between differentiable and non-differentiable pasting. We can see that the model with non-differentiable pasting suffers from bad completion of line drawings, and the stroke order shows the window fails to slide to undrawn pixels. Quantitative results in Table 1 are in line with the visual results, showing a rapid deterioration in both metrics ("**Non-diff.**" in "Pasting" column), especially in chamfer distance which is more sensitive to completeness. These results also indicate the misalignment, and in particular, the gradient blocking caused by non-differentiable pasting prevent the model from learning to update the stroke position well.

5.5 Ablation Study of Raster-level Loss

Other Raster Losses. We first compare the adopted perceptual loss with a pixel-wise difference loss, and in particular, the L_1 distance (the L_2 distance has similar performance). Figure 10 shows that model with L_1 loss draws only a small part of the sketches with dense short strokes. This is probably due to the limited ability the pixel-wise difference loss has when considering the overall structure of the drawings. Significantly worse chamfer distance in Table 1 also indicates the failure in completeness. We also study adversarial loss

Table 1. Ablation studies. For fair comparisons, we change only one factor (in bold type) while remaining others the same as those of our model at last row. Numbers in "Perceptual Layers" indicate the `relu` layers. Values of perceptual score are in e-2 and those of chamfer distance in e-3.

Raster Loss	Pasting	Window	Loss Norm.	Perceptual Layers	Perceptual score(↓)	Chamfer distance(↓)
$\boldsymbol{L_1}$	Diff.	Scalable	-	-	13.616	249.050
Perceptual	**Non-diff.**	Scalable	✓	∪(12, 22, 33, 51)	6.064	60.723
Perceptual	Diff.	**Fixed**	✓	∪(12, 22, 33, 51)	1.938	4.598
Perceptual	Diff.	Scalable	×	∪(12, 22, 33, 51)	1.789	2.597
Perceptual	Diff.	Scalable	✓	**∪(33, 51)**	1.513	2.995
Perceptual	Diff.	Scalable	✓	**∪(12, 22)**	1.399	3.242
Perceptual	Diff.	Scalable	✓	∪(12, 22, 33, 51)	**1.053**	**1.577**

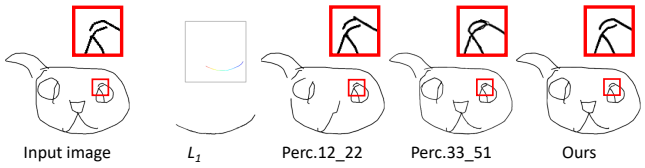

Fig. 10. Comparisons of different raster losses. The numbers after "Perc." indicate the `relu` layer combination.

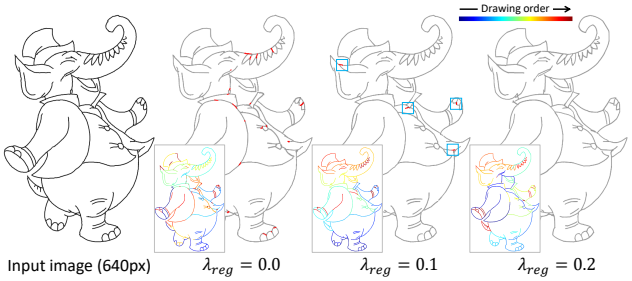

Fig. 11. Removal of redundant strokes by modifying the stroke regularization term weight λ_{reg}. Red strokes in the grayscale vector outputs represent the redundant or overlap strokes.

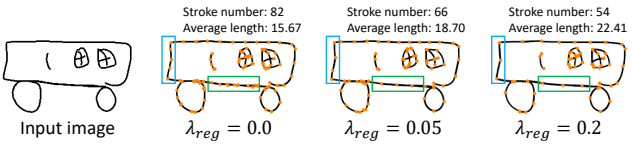

Fig. 12. Improving stroke compactness with different stroke regulation weights λ_{reg}. Orange dots represent the endpoints of each stroke.

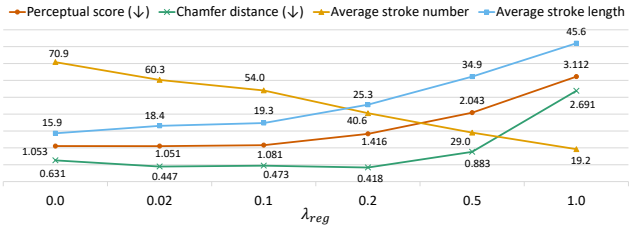

Fig. 13. Quantitative evaluation of the stroke regularization with different weights λ_{reg}. We rescale the values for better visualization of the trends.

which is broadly used in image generation tasks with GAN [Goodfellow et al. 2014]. However, we did not observe any significant improvement.

Different Perceptual Layer Combinations. We use both shallow (`relu1_2` and `relu2_2`) and deep layers (`relu3_3` and `relu5_1`) of the VGG-16 model for the perceptual loss and evaluate their respective performance. Figure 10 shows that models with shallow layers `Perc.12_22` suffer from worse completeness, although they are able to draw details well. Models based on deep layers `Perc.33_51` draw complete sketches but lack in fine details. These results reflect the different strengths of the different layers. In particular, shallow layers focus mainly on low-level information, and are thus able to recover the local details but have an issue of incomplete drawings. On the contrary, deep layers primarily store high-level information, so they benefit the global completeness but suffer from poor fine details. Quantitative comparisons further confirm these, showing a better perceptual score that is sensitive to details but a worse chamfer distance that is sensitive to completeness with **∪(12, 22)**, in comparison to **∪(33, 51)**. Our proposed approach is superior in both metrics. This demonstrates that combining both shallow and deep layers to form the perceptual loss can balance their performance and have the advantages of both.

Loss Value Normalization. We notice from a rough observation that loss values from deeper layers have higher orders of magnitude (*i.e.*, larger loss values). Hence, deeper layers play a more important role in the perceptual loss without value normalization. Quantitative evaluation in Table 1 shows that models without loss normalization ("×" in "Loss Norm." column) perform worse than ours on both metrics. These confirm that loss value normalization is necessary to balance the abilities of different layers. Please refer to the supplemental materials for more results of the ablation studies.

5.6 Stroke Regularization

Redundancy Removal. One function of the stroke regularization mechanism is to avoid the redundant strokes because it forces the model to use fewer strokes to draw while maintaining a similar appearance. Figure 11 shows how a model without regularization ($\lambda_{reg} = 0.0$) produces a number of redundant strokes superimposed onto the strokes already drawn. In contrast, redundant strokes can

be largely avoided in a model with regularization. For instance, only 4 overlapped strokes (blue boxes) are produced in the model with weight $\lambda_{reg} = 0.1$ and no overlap with a larger weight $\lambda_{reg} = 0.2$.

Compactness Improvement. Another function of the stroke regularization mechanism is to improve the compactness. Figure 12 shows as the stroke regularization weight λ_{reg} increases, the model draws the sketches with fewer but longer strokes while maintaining reasonably high fidelity. For instance, the leftmost part (blue boxes) of the bus consists of 4, 3 and 2 strokes for weights 0.0, 0.05 and 0.2, respectively, resulting in a more compact representation.

Statistics in Fig. 13 also show a downward tendency in average stroke number and an upward one in average stroke length as the weight λ_{reg} increases. When $\lambda_{reg} = 0.1$, the average stroke number is about 76% of that in a model without stroke regularization, while the former model still maintains a comparable performance in quantitative evaluation without significant degradation. This confirms that with stroke regularization mechanism, our model does learn to use fewer vector strokes necessary to represent a line drawing.

Parameter Sensibility. From Fig. 13, when adding stroke regularization with weight $\lambda_{reg} = 0.02$, both perceptual score and chamfer distance have a slight drop, and the model uses fewer (85%) strokes to draw. This indicates that a small amount of stroke regularization can improve both fidelity and simplicity. However, as described in §4.4, it reduces the overall performance when imposing an excessive constraint. Figure 13 shows that when $\lambda_{reg} > 0.2$, the model uses much fewer strokes to draw, but suffers a significant performance degradation in quantitative measurements simultaneously. Please refer to the supplemental materials for visual results.

5.7 Vectorization: Comparison with Existing Approaches

Evaluation Settings. We first compare with the learning-based method Learning-To-Paint [Huang et al. 2019], which is closest to our work. Since it works at a fixed resolution, we train different models for different image sizes. We use the same hyperparameters as the official implementation[2] except for the number of strokes in an action bundle, which is set 1 for stroke-by-stroke prediction as ours. Comparison is made only on QuickDraw sketches because the Learning-To-Paint training procedure does not seem to converge on images of higher resolutions.

We then compare with two representative vectorization methods, Fidelity-vs-Simplicity [Favreau et al. 2016] and PolyVectorization [Bessmeltsev and Solomon 2019], on real clean line drawings of different high resolutions. For Fidelity-vs-Simplicity, we try two sets of parameters [3] as in [Bessmeltsev and Solomon 2019] and select the visually best results. For PolyVectorization, we use the default parameters.

Qualitative Results. Comparison with Learning-To-Paint is shown in Fig. 14. At a low resolution (128 px), Learning-To-Paint is able to draw the sketch with a small stroke number (16). However, because

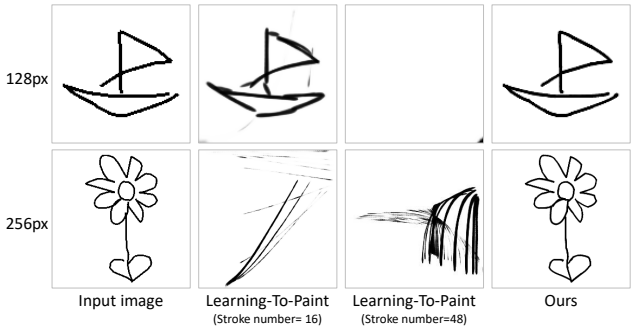

Fig. 14. Comparison with Learning-To-Paint [Huang et al. 2019].

of the discrete strokes it employs, the stroke continuousness is visually worse than ours with continuous strokes. Moreover, there appear redundant drawings due to the lack of a sign for lifting pen. When applied with a larger number of strokes (48) or a higher resolution (256 px), it fails to reconstruct the sketches. This reflects its limitation on higher resolutions and instability in modeling longer sequences, likely due to using reinforcement learning.

Figure 15 shows the comparisons on clean line drawings. Fidelity-vs-Simplicity easily produces connected strokes (green boxes), but fails to draw isolated strokes (*e.g.*, the eyebrows). The connected strokes are caused by its region segmentation-based skeleton extraction algorithm, which tends to add a connected stroke between close strokes for better subdividing of a large complex region. Furthermore, this algorithm is not sensitive to isolated strokes. PolyVectorization produces results with artifacts in regions with fine-grained details and complex junctions (red boxes). The reason is that this method mainly disambiguates X- and T-junctions by tracing the pixel orientations with frame field, but is not robust to more complex junctions with sharp turns or fine details. In contrast, our model works better on both completeness and details. With the proposed dynamic window, our model learns to find and slide to the undrawn area, and is thus able to draw the isolated strokes. Unlike the optimization-based methods above, which fail in situations where the pre-defined curve parameterization principles do not work, our model learns to recover the undrawn pixels as best as it can, so that the details can be guaranteed.

Computation Time. For all methods, we test the examples under the same environment on a Windows PC with an Intel i7-8700 @ 3.2GHz CPU, 48GB RAM, and an NVIDIA GeForce RTX 2070 GPU. For comparison, we show the running time of our model with and without GPU. Table 2 shows that under the same environment without GPU, our model is much faster than Fidelity-vs-Simplicity. Compared with PolyVectorization, our approach takes comparable computation time on low-resolution images while is faster on complicated high-resolution images. When running with a GPU, our method is even faster.

5.8 Other Applications

5.8.1 *Rough Sketch Simplification.*

[2] https://github.com/megvii-research/ICCV2019-LearningToPaint

[3] A default parameter set: maxNumOpenCurves=0, minLengthOpenCurves=30, minRegionSize=7. Another manually selected set: maxNumOpenCurves=30, minLengthOpenCurves=5, minRegionSize=3. Both use a 'fidelity-simplicity' weight of 0.5.

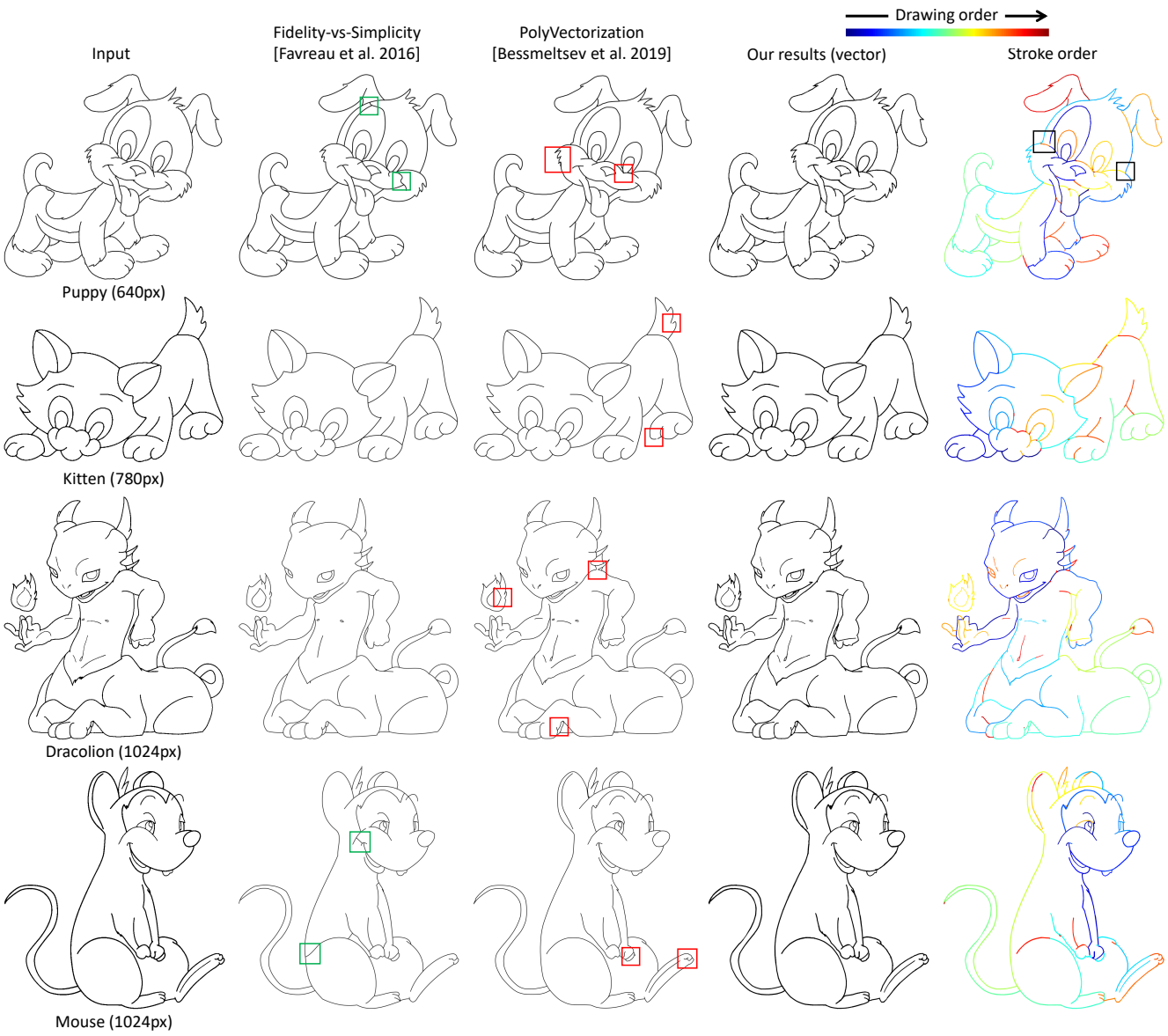

Fig. 15. Comparisons with existing vectorization approaches on real clean line drawings. Our results are from model with stroke regularization of weight $\lambda_{reg} = 0.02$. More examples can be seen in supplemental materials.

Evaluation Settings. We compare with Learning-To-Paint [Huang et al. 2019] on the fixed-resolution dataset QuickDraw with synthetic rough sketches. To the best of our knowledge, there is no existing work on rough sketch (with textured background) simplification which outputs vector lines directly. Therefore, we combine the pixel-level sketch simplification [Simo-Serra et al. 2018a] and a vectorization method as a baseline. A vectorization approach is also applied to rough sketches directly as another baseline. Here PolyVectorization [Bessmeltsev and Solomon 2019] is employed because of its better visual quality compared with Fidelity-vs-Simplicity [Favreau et al. 2016]. We evaluate their performance on complex rough sketches using both rough sketches synthesized from real clean line drawings through pencil art generation technique [Simo-Serra et al. 2018a], and rough sketches in the wild from a recent benchmark dataset [Yan et al. 2020].

For the quantitative evaluation on [Yan et al. 2020], we follow the proposed protocol and use the best chamfer distance score among all input variants compared to all ground truth cleanings for each rough sketch. In contrast with vector approaches such as Fidelity-vs-Simplicity and PolyVectorization that produce fixed stroke thickness, our approach is able to output lines of varying width. Following the evaluation protocol for the raster results from [Simo-Serra et al.

Table 2. Computation time for different vectorization methods on real line drawings. 'F-vs-S' denotes Fidelity-vs-Simplicity [Favreau et al. 2016] and 'PolyVec' PolyVectorization [Bessmeltsev and Solomon 2019].

	Res. (px)	F-vs-S	PolyVec	Ours (CPU)	Ours (GPU)
Elephant	640	55s	17s	20s	11s
Puppy	640	70s	16s	20s	10s
Hippo	700	55s	12s	17s	9s
Penguin	720	96s	8s	11s	6s
Kitten	780	83s	22s	19s	10s
Banana Tree	872	97s	22s	21s	12s
Muten	1024	89s	39s	33s	21s
Mouse	1024	89s	61s	37s	23s
Dracolion	1024	75s	69s	46s	29s
Sheriff	1024	69s	47s	47s	29s

2018a], we use a morphological dilation operation for the ground truth lines with a kernel size of 3 to match the thickness of our rendered outputs in order to not have the varying widths penalize the results.

Qualitative Results. On the QuickDraw dataset, Learning-To-Paint does not seem to perform well at any resolution nor number of strokes, which is similar to the results in the second row in Fig. 14 (see supplemental materials for the full results). This might be because Learning-To-Paint, which is originally designed for reconstruction tasks, is not suitable to domain translation.

On complex rough sketches, PolyVectorization is sensitive to the stroke intensity, as shown in Fig. 16. As strokes in synthetic rough sketches are lightly drawn, PolyVectorization tends to have low completion, and on sketch from the wild, it has an inclination to miss thin and light strokes. This is due to its built-in thresholding operation, which attempts to keep useful pixels but often discards important pixels incorrectly. The pixel-level sketch simplification method [Simo-Serra et al. 2018a] outputs simplified sketches with smooth lines, but there also appear some artifacts, *e.g.*, the redundant parallel strokes (red boxes) resulting from the noise in rough sketches, and some missing vital parts (eye of Bird, forehead of Penguin and sleeve of Hand). Subsequent vectorization is required to convert the pixel-wise images to vector images, but such artifacts are repeated inevitably in the vector outputs. In contrast, our model is able to produce comparable simplified results of vector format in a single step given rough sketch images as input. Furthermore, it gets rid of the noise to a large extent and distinguishes vital lines even though they are in light color.

Quantitative Results. We compute the best chamfer distance for each rough image and obtain an average value of 0.001696. Figure 17 shows the comparison with other approaches, which indicates that our method obtains comparable performance to other algorithms. Our approach is worse than MasteringSketching [Simo-Serra et al. 2018a], because MasteringSketching is trained on real rough sketches and thus tends to work better on most wild sketches in the benchmark. Our model, which is trained on synthetic rough sketches with textured background, works better than most vectorization approaches. We believe that training on a supervised dataset of real sketches could likely improve performance.

5.8.2 *Photograph to Line Drawing.*

Evaluation Settings. We use the test set of CelebAMask-HQ [Lee et al. 2020] dataset with images of 256 px resolution for evaluation. We use Learning-To-Paint [Huang et al. 2019] as one of the baselines. Similar to rough sketch simplification, we combine pixel-level image translation method Photo-Sketching [Li et al. 2019] and PolyVectorization [Bessmeltsev and Solomon 2019] as another baseline.

Qualitative Results. Figure 18 shows the comparisons between the baseline methods and our approach. Learning-To-Paint fails to draw plausible sketches as in rough sketch simplification, which further confirms that it has difficulty in the domain translation task. Photo-Sketching has a good performance in the facial sketch generation task, but it relies on additional vectorization techniques to obtain vector results. In contrast, our approach generates comparable facial sketches and more importantly, works with such domain translation and vectorization simultaneously.

6 LIMITATIONS AND DISCUSSION

Our dynamic window-based framework learns to slide to an undrawn area, and thus is applicable to images of an arbitrary resolution. However, it may still fail to reproduce all the lines in some highly complicated cases. Figure 19 shows some strokes on the left and bottom of the resulting vector line drawing are missing even though the maximum stroke number is set to a considerably large number of 5,000 (for reference, around 1,200 strokes are drawn for Dracolion in Fig. 15). This is because when the CNN encoder downscales the complicated image for a global guidance, it might result in the vanishing of thin lines. Athough we use the random movement method during testing as mentioned in §5.1, the missing strokes can be too small to be noticed. Therefore, resizing full-size images directly and random movement can be seen as feasible but not optimal solutions to detecting undrawn thin lines globally. To better capture global information and not necessitate random movements, training with higher-resolution images, alternative encoding methods (*e.g.*, pyramid views) with the global guidance, or a different global guidance could be considered to address this problem.

Another limitation is that it is still difficult for our framework to generalize well on complex rough sketches or photographs and it may produce artifacts in the results. As an example, in Fig. 16, the lines in the Penguin in our result are not as smooth as those from Sketch Simplification, and in Fig. 18, the noses in our results are connected with the eyebrows while they are clearly separated in those from Photo-Sketching. The non-smoothness problem could be addressed by using a curve refinement technique [Das et al. 2020] as post-processing. In addition, the performances on both tasks could be improved by combining the pixel-level sketch simplification or image-to-sketch model and our approach in a single end-to-end model, which may be a promising future direction.

Although the virtual pen-based approach is efficient when drawing short and long strokes, it can perform less than satisfactory in

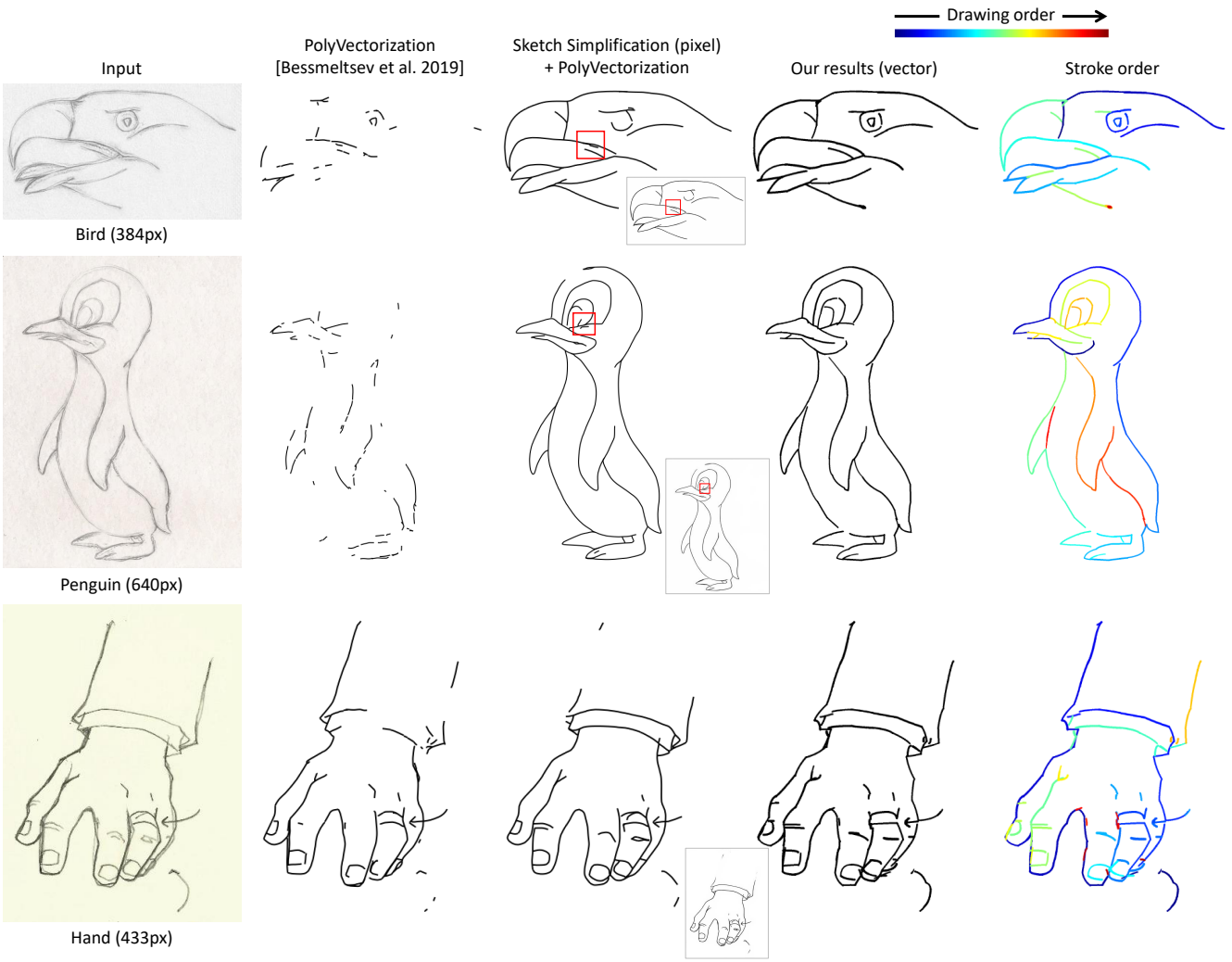

Fig. 16. Comparisons with existing approaches on rough sketch simplification. The Bird and Penguin are synthetic rough sketches, while the Hand is from a rough sketch benchmark dataset [Yan et al. 2020]. Small figures in the third column are the pixel-level outputs from the sketch simplification method [Simo-Serra et al. 2018a].

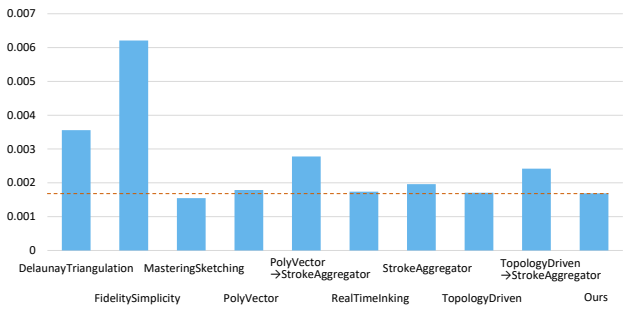

Fig. 17. Quantitative evaluation with chamfer distance (↓) on the rough sketch benchmark [Yan et al. 2020].

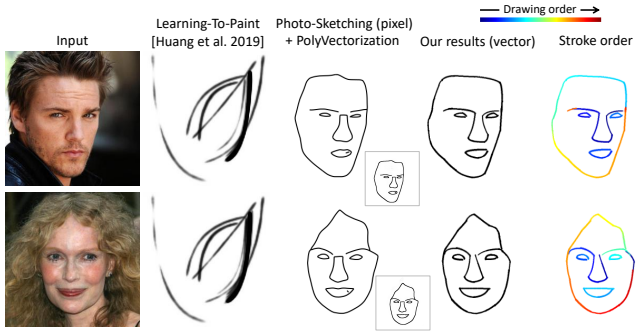

Fig. 18. Comparisons with existing approaches on photograph to line drawing. Small figures at third column are from Photo-Sketching [Li et al. 2019].

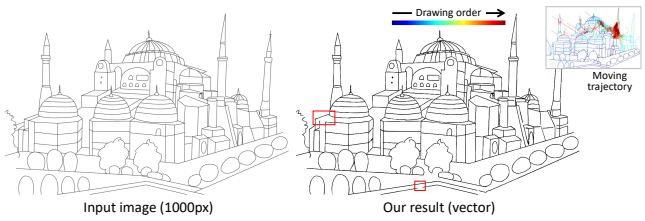

Fig. 19. Limitation of our approach on a highly complicated example. Some short strokes in the output are missing (red boxes) even when the maximum number of strokes is set to 5,000.

some types of junctions. For instance, looking at the stroke order for the Puppy in Fig. 15, there are some T-junctions (black boxes) in which horizontal curves are not drawn together (from left to right or the opposite). The reason is that our model is not intended for recovery of topology or meaningful drawing order while vectorizing line drawings. We think this could be an issue rather than a limitation of our approach, because there is no general consensus in drawing order (*e.g.*, how squares are drawn in east-Asia vs. the west). Furthermore, in reality different people have various ways of drawing. Some artists prefer to draw the silhouette first and then fill in the inner details with shorter segments, while others draw part by part. While correct topology and specific drawing orders might be beneficial for certain applications, post-processing, pre-defined principles as prior or constraint information can be incorporated to form a future extension of our approach.

ACKNOWLEDGMENTS

This work was supported by the Natural Science Foundation of Guangdong Province, China (Grant No. 2019A1515011075) and the National Key R&D Program of China (2018AAA0100300).

REFERENCES

Mikhail Bessmeltsev and Justin Solomon. 2019. Vectorization of line drawings via polyvector fields. *ACM Transactions on Graphics (TOG)* 38, 1 (2019), 1–12.

Jifeng Dai, Kaiming He, and Jian Sun. 2016. Instance-aware Semantic Segmentation via Multi-task Network Cascades. In *CVPR*.

Ayan Das, Yongxin Yang, Timothy Hospedales, Tao Xiang, and Yi-Zhe Song. 2020. BézierSketch: A generative model for scalable vector sketches. In *The European Conference on Computer Vision (ECCV)*.

Vage Egiazarian, Oleg Voynov, Alexey Artemov, Denis Volkhonskiy, Aleksandr Safin, Maria Taktasheva, Denis Zorin, and Evgeny Burnaev. 2020. Deep Vectorization of Technical Drawings. *arXiv preprint arXiv:2003.05471* (2020).

Haoqiang Fan, Hao Su, and Leonidas J Guibas. 2017. A point set generation network for 3d object reconstruction from a single image. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 605–613.

Jean-Dominique Favreau, Florent Lafarge, and Adrien Bousseau. 2016. Fidelity vs. Simplicity: a Global Approach to Line Drawing Vectorization. *ACM Transactions on Graphics (SIGGRAPH Conference Proceedings)* (2016).

Yaroslav Ganin, Tejas Kulkarni, Igor Babuschkin, SM Ali Eslami, and Oriol Vinyals. 2018. Synthesizing Programs for Images using Reinforced Adversarial Learning. In *ICML*.

Ross Girshick. 2015. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*. 1440–1448.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *Advances in neural information processing systems*. 2672–2680.

Alex Graves. 2013. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850* (2013).

Yi Guo, Zhuming Zhang, Chu Han, Wenbo Hu, Chengze Li, and Tien-Tsin Wong. 2019. Deep Line Drawing Vectorization via Line Subdivision and Topology Reconstruction. In *Computer Graphics Forum*, Vol. 38. Wiley Online Library, 81–90.

David Ha and Douglas Eck. 2018. A Neural Representation of Sketch Drawings. In *International Conference on Learning Representations*.

Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. 2017. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*. 2961–2969.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

Zhewei Huang, Wen Heng, and Shuchang Zhou. 2019. Learning to paint with model-based deep reinforcement learning. In *Proceedings of the IEEE International Conference on Computer Vision*. 8709–8718.

Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. 2017. Image-to-image translation with conditional adversarial networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1125–1134.

Justin Johnson, Alexandre Alahi, and Li Fei-Fei. 2016. Perceptual losses for real-time style transfer and super-resolution. In *European conference on computer vision*. Springer, 694–711.

Byungsoo Kim, Oliver Wang, A. Cengiz Öztireli, and Markus Gross. 2018. Semantic Segmentation for Line Drawing Vectorization Using Neural Networks. *Computer Graphics Forum (Proc. Eurographics)* 37, 2 (2018), 329–338.

Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

Cheng-Han Lee, Ziwei Liu, Lingyun Wu, and Ping Luo. 2020. MaskGAN: Towards Diverse and Interactive Facial Image Manipulation. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Mengtian Li, Zhe Lin, Radomir Mech, Ersin Yumer, and Deva Ramanan. 2019. Photo-sketching: Inferring contour drawings from images. In *2019 IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE, 1403–1412.

Tzu-Mao Li, Michal Lukáč, Michaël Gharbi, and Jonathan Ragan-Kelley. 2020. Differentiable vector graphics rasterization for editing and learning. *ACM Transactions on Graphics (TOG)* 39, 6 (2020), 1–15.

Chenxi Liu, Enrique Rosales, and Alla Sheffer. 2018b. Strokeaggregator: Consolidating raw sketches into artist-intended curve drawings. *ACM Transactions on Graphics (TOG)* 37, 4 (2018), 1–15.

Lingjie Liu, Duygu Ceylan, Cheng Lin, Wenping Wang, and Niloy J Mitra. 2017. Image-based reconstruction of wire art. *ACM Transactions on Graphics (TOG)* 36, 4 (2017), 1–11.

Rosanne Liu, Joel Lehman, Piero Molino, Felipe Petroski Such, Eric Frank, Alex Sergeev, and Jason Yosinski. 2018a. An Intriguing Failing of Convolutional Neural Networks and the CoordConv Solution. In *Advances in Neural Information Processing Systems*.

Xueting Liu, Tien-Tsin Wong, and Pheng-Ann Heng. 2015. Closure-aware sketch simplification. *ACM Transactions on Graphics (TOG)* 34, 6 (2015), 1–10.

Diogo C Luvizon, David Picard, and Hedi Tabia. 2018. 2D/3D Pose Estimation and Action Recognition Using Multitask Deep Learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 5137–5146.

John FJ Mellor, Eunbyung Park, Yaroslav Ganin, Igor Babuschkin, Tejas Kulkarni, Dan Rosenbaum, Andy Ballard, Theophane Weber, Oriol Vinyals, and SM Eslami. 2019. Unsupervised Doodling and Painting with Improved SPIRAL. *arXiv preprint arXiv:1910.01007* (2019).

Reiichiro Nakano. 2019. Neural painters: A learned differentiable constraint for generating brushstroke paintings. *arXiv preprint arXiv:1904.08410* (2019).

Gioacchino Noris, Alexander Hornung, Robert W Sumner, Maryann Simmons, and Markus Gross. 2013. Topology-driven vectorization of clean line drawings. *ACM Transactions on Graphics (TOG)* 32, 1 (2013), 1–11.

Edgar Simo-Serra, Satoshi Iizuka, and Hiroshi Ishikawa. 2018a. Mastering sketching: adversarial augmentation for structured prediction. *ACM Transactions on Graphics (TOG)* 37, 1 (2018), 1–13.

Edgar Simo-Serra, Satoshi Iizuka, and Hiroshi Ishikawa. 2018b. Real-time data-driven interactive rough sketch inking. *ACM Transactions on Graphics (TOG)* 37, 4 (2018), 1–14.

Edgar Simo-Serra, Satoshi Iizuka, Kazuma Sasaki, and Hiroshi Ishikawa. 2016. Learning to simplify: fully convolutional networks for rough sketch cleanup. *ACM Transactions on Graphics (TOG)* 35, 4 (2016), 1–11.

Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *International Conference on Learning Representations*.

Jifei Song, Kaiyue Pang, Yi-Zhe Song, Tao Xiang, and Timothy M Hospedales. 2018. Learning to sketch with shortcut cycle consistency. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 801–810.

Tibor Stanko, Mikhail Bessmeltsev, David Bommes, and Adrien Bousseau. 2020. Integer-Grid Sketch Simplification and Vectorization. In *Computer Graphics Forum*, Vol. 39. Wiley Online Library, 149–161.

Qingkun Su, Xue Bai, Hongbo Fu, Chiew-Lan Tai, and Jue Wang. 2018. Live sketch: Video-driven dynamic deformation of static drawings. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–12.

Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. 2016. Instance normalization: The missing ingredient for fast stylization. *arXiv preprint arXiv:1607.08022* (2016).

Saining Xie and Zhuowen Tu. 2015. Holistically-nested edge detection. In *Proceedings of the IEEE international conference on computer vision*. 1395–1403.

Xuemiao Xu, Minshan Xie, Peiqi Miao, Wei Qu, Wenpeng Xiao, Huaidong Zhang, Xueting Liu, and Tien-Tsin Wong. 2019. Perceptual-aware Sketch Simplification Based on Integrated VGG Layers. *IEEE Transactions on Visualization and Computer Graphics* (2019).

Chuan Yan, David Vanderhaeghe, and Yotam Gingold. 2020. A benchmark for rough sketch cleanup. *ACM Transactions on Graphics (TOG)* 39, 6 (2020), 1–14.

Richard Zhang, Phillip Isola, Alexei A Efros, Eli Shechtman, and Oliver Wang. 2018. The unreasonable effectiveness of deep features as a perceptual metric. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 586–595.

Ningyuan Zheng, Yifan Jiang, and Dingjiang Huang. 2019. StrokeNet: A Neural Painting Environment. In *International Conference on Learning Representations*.