General Virtual Sketching Framework for Vector Line Art — Supplemental Material —

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

1 FRAMEWORK DETAILS

1.1 Stroke Representation

1.1.1 Width Factor Updating. The strokes are represented in a relative manner in our framework. That is, we need to use the ending point of a stroke from last time step as the starting position for the next stroke, so that a quadratic Bézier curve can be formed. Since the stroke representation a_t in Eq. (1) in main paper contains the width factor w_t, we need to use the width factor w_{t-1} from last time step as the width $(r_0)_t$ for starting position for the next Bézier curve q_t, as shown in Eq. (3) in main paper.

Because of the scalable window in our model, the size of two consecutive windows might be different. As a relative value based on the window size, when the width factor from last time step is used for the next window, it should be scaled to adapt to the new window. Practically, we re-define the $(r_0)_t$ in Eq. (3) in main paper as:

$$(r_0)_t = w'_{t-1}, \tag{1}$$

where w'_{t-1} is the scaled width from last time step. The scaling process is:

$$\widehat{w'_t} = w_t \times (W_{t-1}/W_t), \quad w'_t = \min(1.0, \widehat{w'_t}), \tag{2}$$

where W denotes the window size and $\widehat{\cdot}$ the original values after relative scaling. *min* operations performing value clipping is adopted to avoid the out-of-bounds issue. In the experiments, we set initial values $w'_0 = 0.01$.

1.1.2 Differentiable Pen State Binarizing. The differentiable *softargmax* operation introduced in Section 3.2.1 is formulated as:

$$\text{softargmax}(x) = \sum_i \frac{e^{\beta x_i}}{\sum_j e^{\beta x_j}} i, \tag{3}$$

where $\frac{e^{y_i}}{\sum_j e^{y_j}}$ is the standard *softmax* operation and $\sum_i z_i i$ is the expectation for the index of the maximum probability. The β is to raise the maximum value and lower the others for a more accurate index. We note that this operation is a trade-off between accuracy and the intensity of gradients, which means a bigger β leads to a smaller amount of gradients.

*Corresponding author.

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

1.2 Stroke Generator

1.2.1 CNN Encoder. Apart from the patches cropped from the input image $\in \mathbb{R}^{3\times128\times128}$ and canvas $\in \mathbb{R}^{1\times128\times128}$, as well as the resized entire image $\in \mathbb{R}^{3\times128\times128}$ and canvas $\in \mathbb{R}^{1\times128\times128}$, we use the cursor position $\in \mathbb{R}^2$ as additional input. The cursor position is tiled to spatial maps $\in \mathbb{R}^{2\times128\times128}$.

The architecture of CNN encoder is shown in Table 1. At the first layer of the encoder, we employ CoordConv [Liu et al. 2018] to boost the encoding of spatial relation between global images and local patches. The image features $\in \mathbb{R}^{512\times4\times4}$ are flattened to feature vector $\in \mathbb{R}^{8192}$. Then a fully connected layer converts the feature vector to image embedding $z_t \in \mathbb{R}^{128}$.

Table 1. Architecture of CNN encoder.

Layer Type	Kernel Size	Stride	Normalization Function	Activation Function	Output
Input	-	-	-	-	$10 \times 128 \times 128$
CoordConv	-	-	-	-	$12 \times 128 \times 128$
Convolutional	3×3	2	Instance Norm.	ReLU	$32 \times 64 \times 64$
Convolutional	3×3	1	Instance Norm.	ReLU	$32 \times 64 \times 64$
Convolutional	3×3	2	Instance Norm.	ReLU	$64 \times 32 \times 32$
Convolutional	3×3	1	Instance Norm.	ReLU	$64 \times 32 \times 32$
Convolutional	3×3	2	Instance Norm.	ReLU	$128 \times 16 \times 16$
Convolutional	3×3	1	Instance Norm.	ReLU	$128 \times 16 \times 16$
Convolutional	3×3	1	Instance Norm.	ReLU	$128 \times 16 \times 16$
Convolutional	3×3	2	Instance Norm.	ReLU	$256 \times 8 \times 8$
Convolutional	3×3	1	Instance Norm.	ReLU	$256 \times 8 \times 8$
Convolutional	3×3	1	Instance Norm.	ReLU	$256 \times 8 \times 8$
Convolutional	3×3	2	Instance Norm.	ReLU	$512 \times 4 \times 4$
Convolutional	3×3	1	Instance Norm.	ReLU	$512 \times 4 \times 4$
Convolutional	3×3	1	Instance Norm.	ReLU	$512 \times 4 \times 4$

1.2.2 RNN Decoder. At each time step t, the RNN decoder takes as input the image embedding $z_t \in \mathbb{R}^{128}$ and the previous hidden state $h_{t-1} \in \mathbb{R}^{256}$. In the implementation, besides the two factors above, we use the following additional three inputs: (1) previous width factor $w_{t-1} \in \mathbb{R}$ for the prediction of w_t because the their corresponding endpoints form a shared stroke; (2) previous cursor $Q_{t-1} \in \mathbb{R}^2$ to tell the model not to draw outside the full canvas; (3) $\Delta S_{t-1}^{up} = W_{t-1}/W_I \in \mathbb{R}$ and $\Delta S_{t-1}^{bottom} = W_{t-1}/W_{min} \in \mathbb{R}$ to suggest the model not to scale the window beyond the upper and bottom bound. The output $o_t \in \mathbb{R}^{256}$ of RNN is then converted to stroke parameters $a_t \in \mathbb{R}^7$ through a fully connected layer.

1.3 Coordinate System Change in Differentiable Pasting

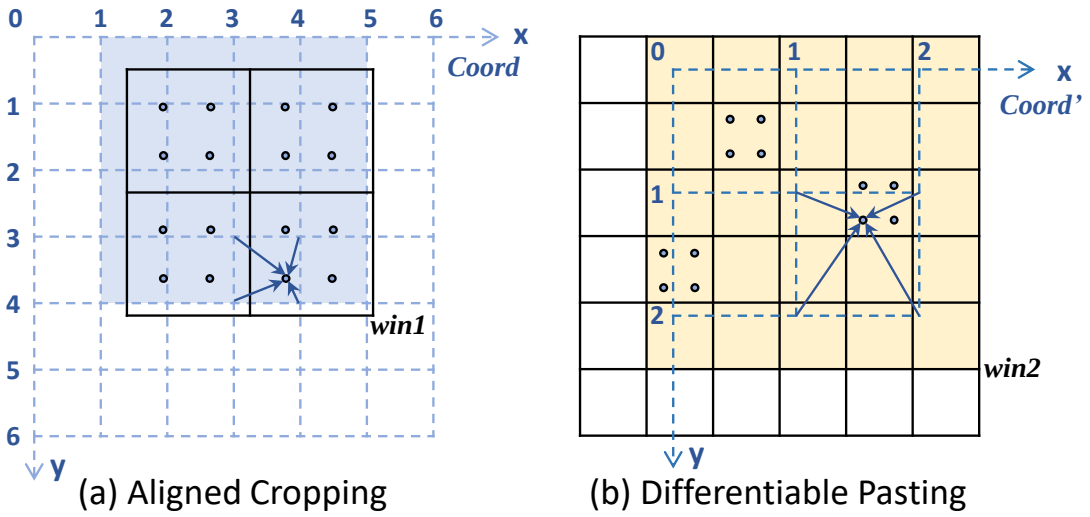

Fig. 1. Aligned cropping and differentiable pasting.

In Fig. 1, we define original image space as coordinate system *Coord* and the rendered (or cropped) patch space as coordinate system *Coord*′. Q/W and Q'/W' denote the cursor/window size under *Coord* and *Coord*′, respectively. $win1$ represents cropping window (black lines) in Fig. 1-(a), and $win2$ the one (black lines of yellow region) in Fig. 1-(b). For example, $W_{win1} = 3.7$ and $W'_{win1} = 2$. Note that $W'_{win1} = W_r$.

The aligned cropping operation depends on cursor and window size (Q_{win1}, W_{win}) from cropping window $win1$ under the base coordinate *Coord*. So during pasting (*i.e.*, another kind of aligned cropping), our goal is to compute (Q'_{win2}, W'_{win2}) for cropping window $win2$ under *Coord*′. This is done by first computing (Q_{win2}, W_{win2}) under *Coord* and then conversing them to *Coord*′. The computation steps are as follows:

(1) Position of $win1$ under *Coord*: the top-left position $P^{\nwarrow}_{win1}$ and bottom-right position $P^{\searrow}_{win1}$ of $win1$ are:

$$P^{\nwarrow}_{win1} = Q_{win1} - W_{win1}/2.0, \quad P^{\searrow}_{win1} = Q_{win1} + W_{win1}/2.0. \tag{4}$$

(2) Position of $win2$ under *Coord*: $win2$ is the bounding box of rendered patch (dashed grid in Fig. 1-(b)) within $win1$. Its top-left and bottom-right positions are:

$$P^{\nwarrow}_{win2} = \left\lfloor P^{\nwarrow}_{win1} \right\rfloor, \quad P^{\searrow}_{win2} = \left\lceil P^{\searrow}_{win1} \right\rceil. \tag{5}$$

(3) Cursor and window size of $win2$ under *Coord*: they are calculated according to the bounding box:

$$Q_{win2} = \left(P^{\nwarrow}_{win2} + P^{\searrow}_{win2}\right)/2.0, \quad W_{win2} = P^{\searrow}_{win2} - P^{\nwarrow}_{win2}. \tag{6}$$

(4) Changing from *Coord* to *Coord*′:

$$\begin{aligned} Q'_{win2} &= \left(W'_{win1}/W_{win1}\right) \times \left(Q_{win2} - P^{\nwarrow}_{win1}\right), \\ W'_{win2} &= \left(W'_{win1}/W_{win1}\right) \times W_{win2}. \end{aligned} \tag{7}$$

After the coordinate system change for the cursor and size of window *win*2 in Fig. 1-(b), differentiable pasting is performed as an aligned cropping process, and produces the patch in yellow region. By padding the left pixels with constant values, a pasted canvas in full resolution is obtained, which is able to propagate the gradients derived from the raster-level loss to the floating-number cursor and window size.

1.4 Neural Renderer

The neural renderer in our work is similar to that in Learning-To-Paint [Huang et al. 2019]. The main difference is that we discard the RGB and the transparency parameters, which are unnecessary in our task. The architecture is as follows:

Table 2. Architecture of neural renderer.

Layer Type	Kernel Size	Activation Function	Output
Input	-	-	10
Fully Connected	-	ReLU	512
Fully Connected	-	ReLU	1024
Fully Connected	-	ReLU	2048
Fully Connected	-	ReLU	4096
Reshape	-	-	$16 \times 16 \times 16$
Convolutional	3×3	ReLU	$32 \times 16 \times 16$
Convolutional	3×3	-	$32 \times 16 \times 16$
Pixel Shuffle	-	-	$8 \times 32 \times 32$
Convolutional	3×3	ReLU	$16 \times 32 \times 32$
Convolutional	3×3	-	$16 \times 32 \times 32$
Pixel Shuffle	-	-	$4 \times 64 \times 64$
Convolutional	3×3	ReLU	$8 \times 64 \times 64$
Convolutional	3×3	-	$4 \times 64 \times 64$
Pixel Shuffle	-	Sigmoid	$1 \times 128 \times 128$

2 DATASETS

2.1 Sketches for Vectorization and Rough Sketches Simplification

During training and evaluation, we use library gizeh [1] to render raster sketches with vector sequential points in QuickDraw [Ha and Eck 2018] dataset. In training, we render sketches in multiple resolutions ranging from 128px to 278px. All the raster images are rendered on the fly during training. For different resolutions, we use different numbers of sketch objects. For each object, we render it into different sizes with different line thickness, and then place them to different position. All the information are shown below:

Table 3. Information of raster sketch rendering during training.

Resolution	Object Number	Object Size	Position	Line Thickness
$[128, 172]$	1	*Original*	Center	3
$(172, 225]$	1	*Original*	Center	3 or 4
$(172, 225]$	2	$[128, Original \times 0.75]$	Random	3 or 4
$(225, 278]$	2	$[128, Original \times 0.75]$	Random	3 or 4

During evaluation, we render sketches at four resolutions: 128px, 256px, 384px and 512px. All have only one object in original size and line thickness 3. All the objects are placed in the center.

We use 10 categories for training: airplane, bus, car, sailboat, bird, cat, dog, tree, flower and zigzag. During evaluation, we augment three categories: rabbit, circle and line.

2.2 Rough Sketches for Simplification

In rough sketches simplification task, we use the clean sketches for vectorization as target images and use the pencil art generation technique in [Simo-Serra et al. 2018] to produce the rough sketches as input images. We use the released code and model [2]. There are two drawing styles from two artists, and we use both during training for better generalization.

2.3 Photograph to Line Drawing

There are face images and the corresponding segmentation masks in CelebAMask-HQ [Lee et al. 2020] dataset, which contains 19 categories for the facial details. We discard categories (l_ear, r_ear, hair, hat, neck, ear_r, neck_l and cloth) and adopt (skin, nose, eye_g, l_eye, r_eye, mouth, u_lip, l_lip, l_brow and r_brow). For mouth, u_lip and l_lip, we merge their masks to form an entire shape. For l_brow and r_brow, we use skeleton extraction algorithm to convert them into thin lines. The original size for face images is 1024px and annotated mask 512px. We resize them to both 256px for training and testing. The facial sketches are obtained by applying canny edge detection algorithm to the mask image. The canny edge is in 1-pixel width, and then morphological dilation method is employed to thicken the lines. Some training or evaluation examples are shown as follows:

[1] https://github.com/Zulko/gizeh
[2] https://github.com/bobbens/sketch_simplification

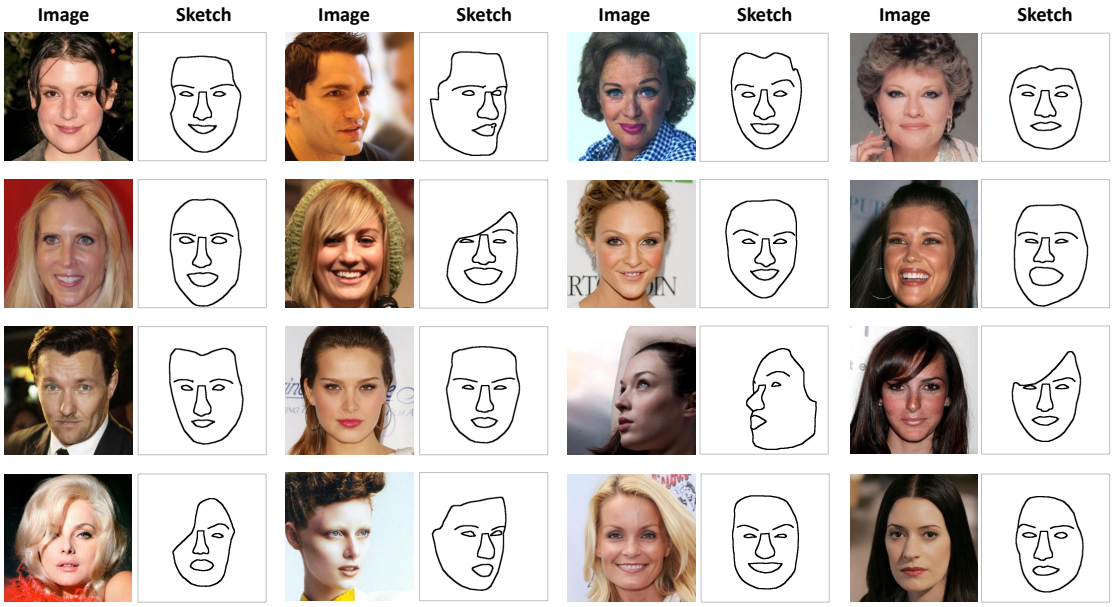

Fig. 2. Training or evaluation examples of photograph to line drawing.

3 IMPLEMENTATION DETAILS

3.1 Avoid Recursive Gradient Propagation

When training the sequential model, some outputs from last time step are used as inputs for the next step, like the canvas C, cursor Q, etc. To avoid recursive gradient back-propagation, we break the gradients of all the inputs of CNN encoder and the additional inputs of RNN decoder. Gradient breaking should also be done when updating the cursor Q_t and window size W_t in Eq. (4) in main paper, where gradients should not be propagated recursively from Q_t and W_t to Q_{t-1} and W_{t-1}.

3.2 More about Training

For each experiment of our method, the training is done on 2 GPUs, each with different resolutions (except for photograph to line drawing which is trained on images of single resolution).

3.3 Random Movement of Cursor during Testing

During testing or evaluation, in order to help the window slide to distant undrawn area more efficiently, the cursor is randomly moved to another position. We use different strategies of random cursor movement for different types of images:

Real Clean Sketches. Since the input clean sketches are also the target ones, we are able to detect the undrawn pixels by comparing the target and the canvas. After each round of drawing, we first divide the full-size sketch and canvas into several 128×128 grids, and calculate their *number of undrawn stroke pixels* and *stroke drawing accuracy*. Then, when the stroke drawing accuracy of all grids is higher than $k\%$, the drawing is stopped early without using up all the rounds for drawing. Otherwise, we select the grid with the most undrawn stroke pixels, and move the cursor to a random position inside that grid. $k = 95$ is used in the experiments.

Rough Sketches and Photographs. Different from clean sketches, rough sketches and photographs cannot be used as target during testing. So the cursor movement strategy is different from the one for clean sketches. We move the cursor to a random position without any restriction.

4 RESULTS

4.1 Effectiveness of Moving and Scaling

Figure 3 shows results of how to move, break the continuous strokes and slide to undrawn region. Figure 4 shows results of how to enlarge the window to search for undrawn region, and then slide there with long strides. Figure 5 shows the window size distributions of our dynamic window-based model on vectorization task. In contrast, fixed-size window-based model uses a window size of 128px. Visual comparisons between dynamic window-based model and fixed-size window-based one is shown in Figure 6.

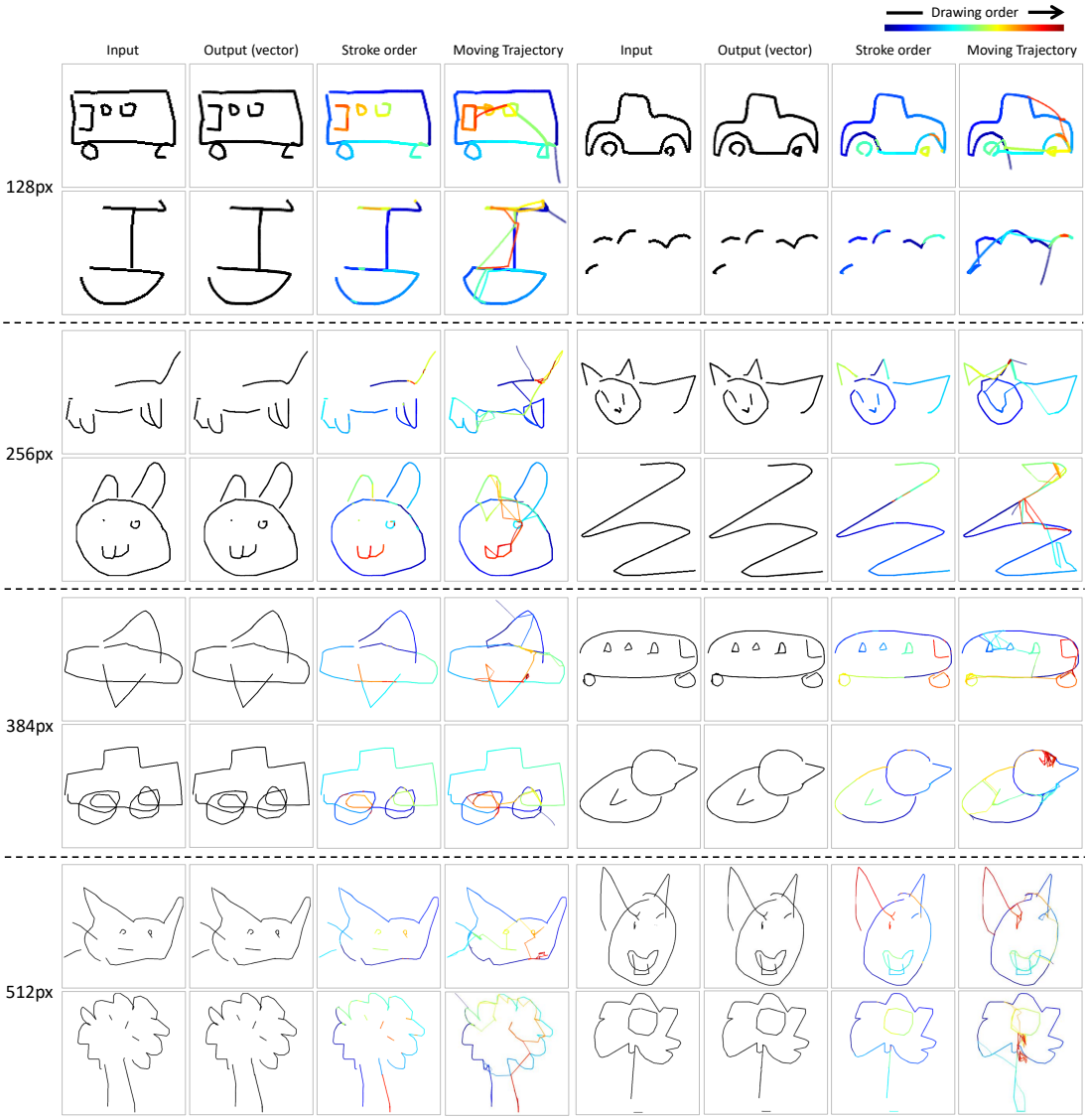

Fig. 3. Results of moving, sliding and breaking.

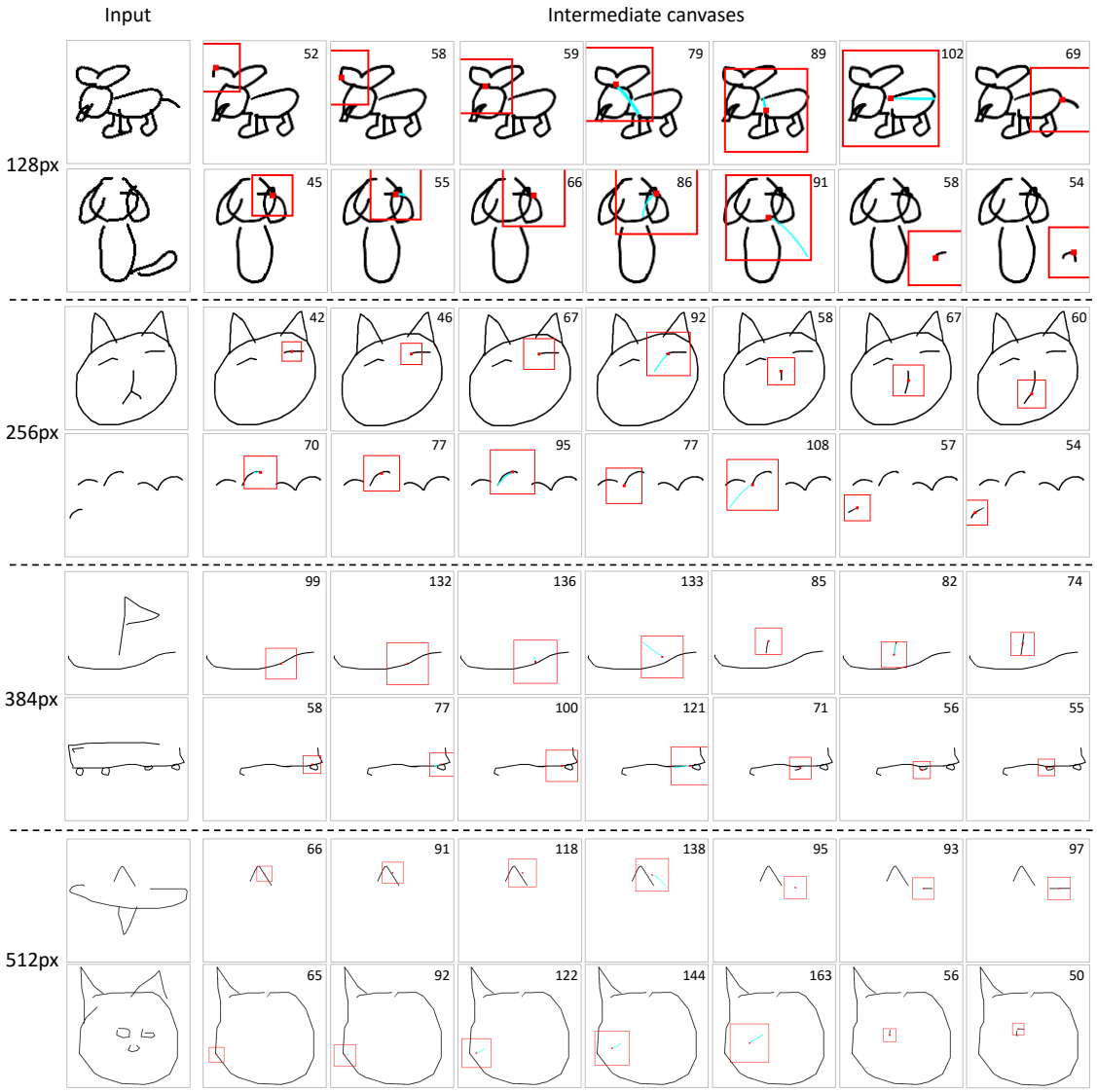

Fig. 4. Results of window scaling. Red boxes represent the windows. Numbers on top right corner indicate the window size. Blue lines are the slide trajectory which are not drawn finally.

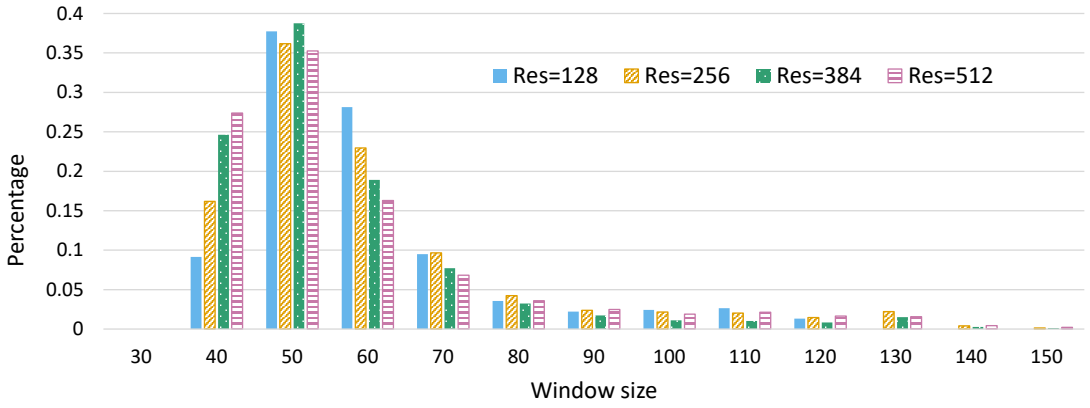

Fig. 5. Distributions of window sizes in our dynamic window-based model without stroke regularization. Here we only consider windows used to draw strokes instead of sliding or breaking.

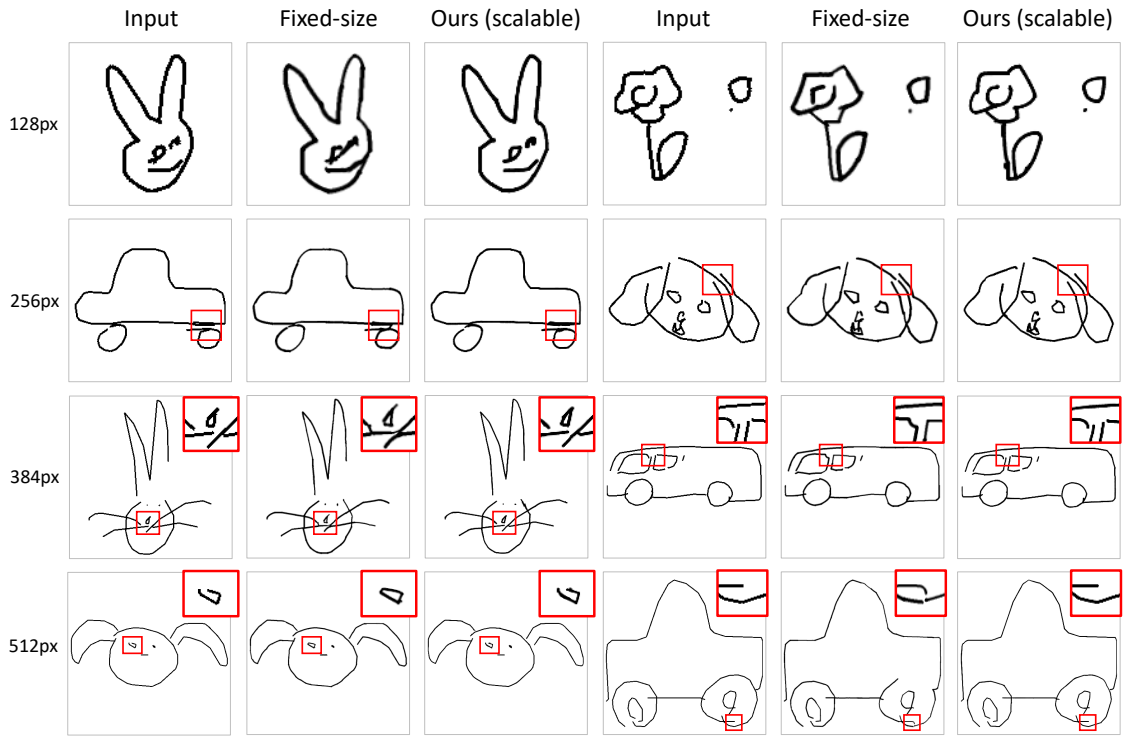

Fig. 6. Comparisons between scalable/dynamic window-based model (ours) and fixed-size window-based one.

4.2 Effectiveness of Differentiable Pasting

Figure 7 shows the distributions in model with or without differentiable pasting at a low and a high resolution on vectorization task. Figure 8 shows how the model with non-differentiable pasting scales the window. Figure 9 shows visual comparisons between differentiable and non-differentiable pasting.

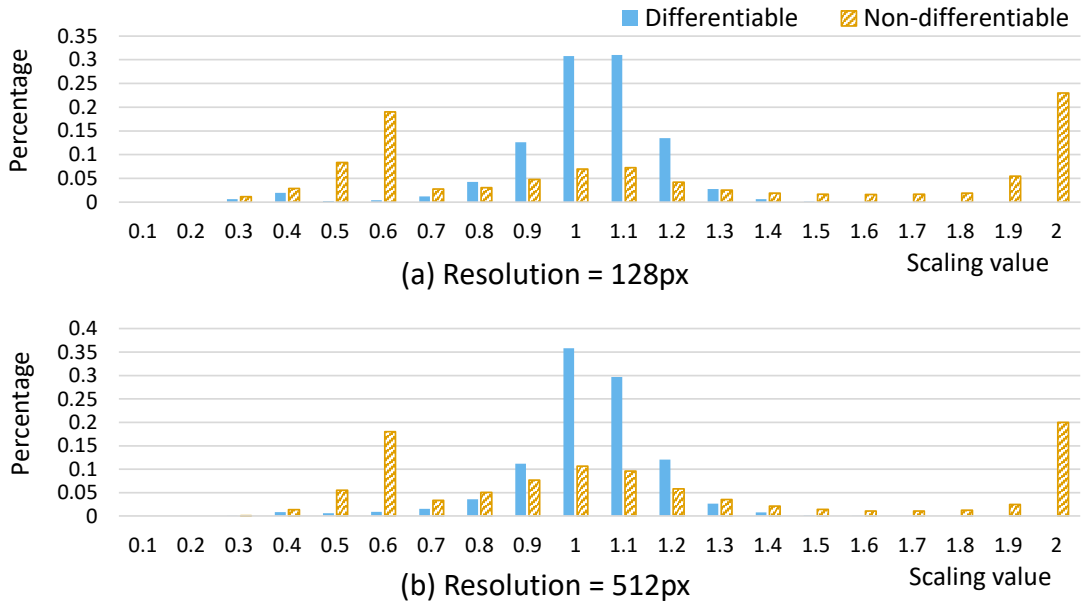

Fig. 7. Distributions of scaling factor values in model with differentiable or non-differentiable pasting. Here we only consider the scaling values of windows used to draw strokes instead of sliding or breaking.

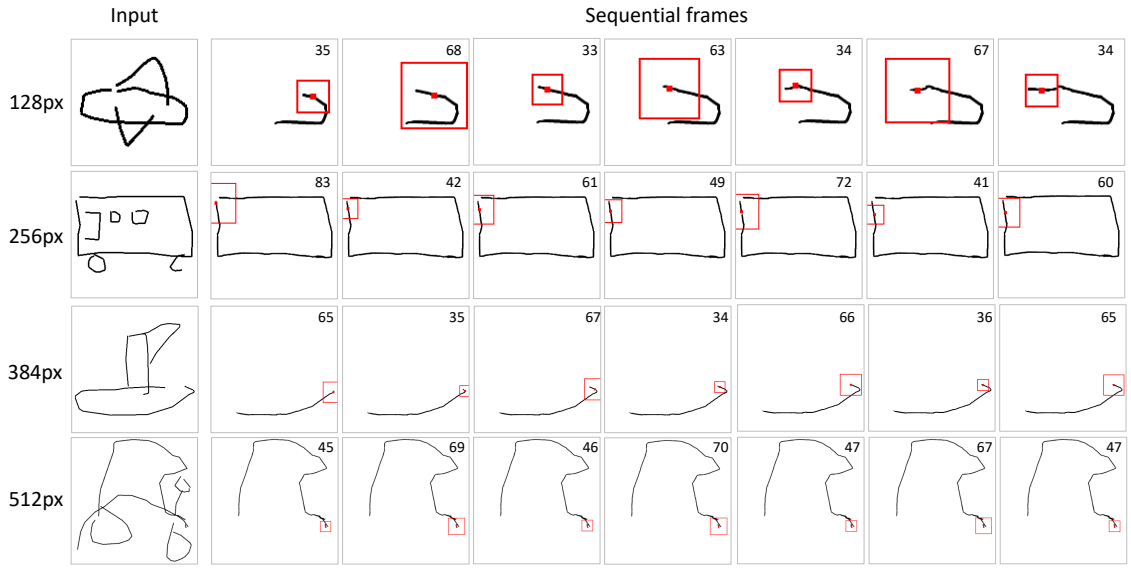

Fig. 8. Window scaling of the model with non-differentiable pasting.

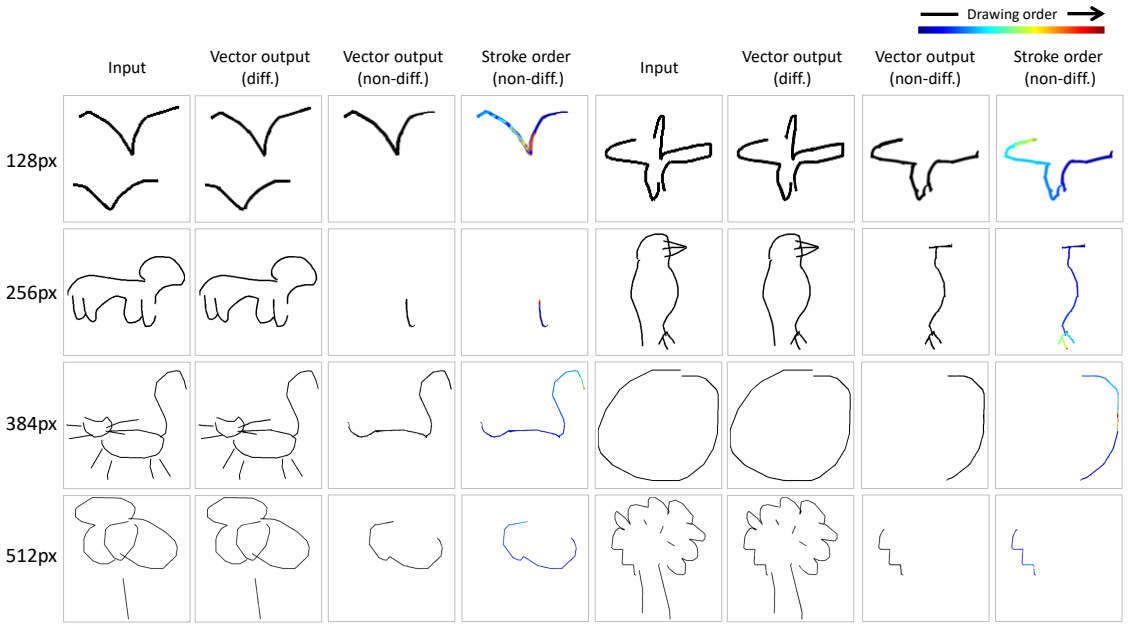

Fig. 9. Comparisons between differentiable ("diff.") and non-differentiable ("non-diff.") pasting.

4.3 Ablation Study of Raster-level Loss

4.3.1 Other Raster Losses. Besides pixel-wise difference loss (L_1 difference) and perceptual loss, we also study adversarial loss. For a fair comparison, we use perceptual loss as the base supervision loss and add it to the objective function of generator. Out-of-bounds penalty loss and stroke regularization loss are also added to the objective function of generator during training. Figure 12 shows visual comparisons among different raster losses. When adopting the adversarial loss, no significant difference is observed from the visual results of different resolutions. Though there is slight improvement in the quantitative results (Table 4), it is not significant enough. Given that an additional learnable discriminator leads to higher training complexity, we discard the adversarial loss and use perceptual loss only.

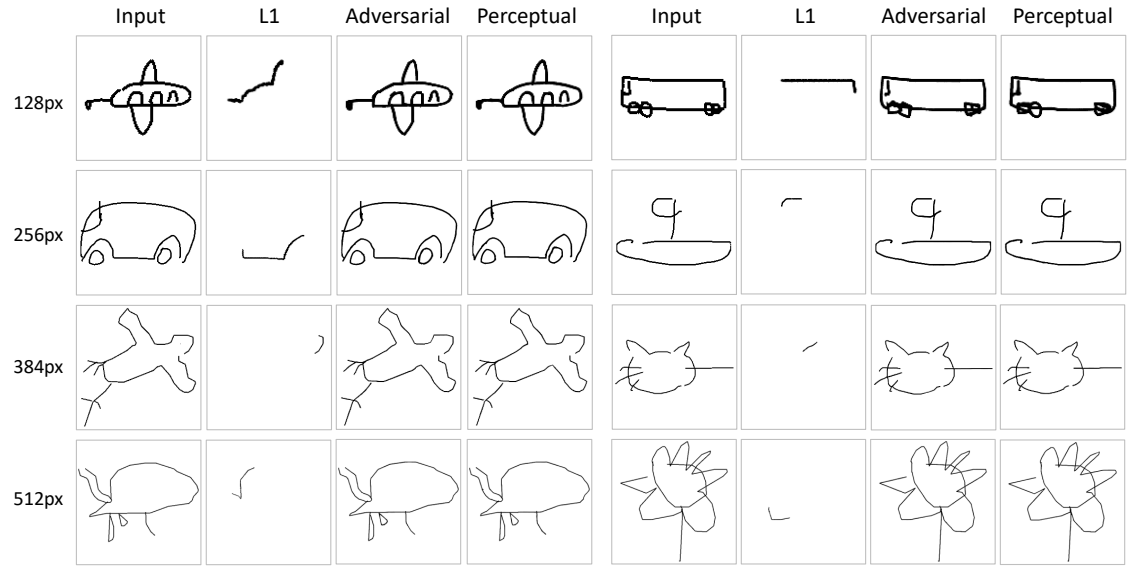

Fig. 10. Comparisons among different raster losses.

Table 4. Quantitative results of different raster losses.

Raster Loss	Perceptual score(↓)	Chamfer distance(↓)
Perceptual	1.053	1.577
Perceptual + Adversarial	1.041	1.343

4.3.2 Different Layer Combinations. Figure 11 shows visual results of model with different perceptual layer combinations from VGG-16 [Simonyan and Zisserman 2015]. Table 5 shows quantitative results.

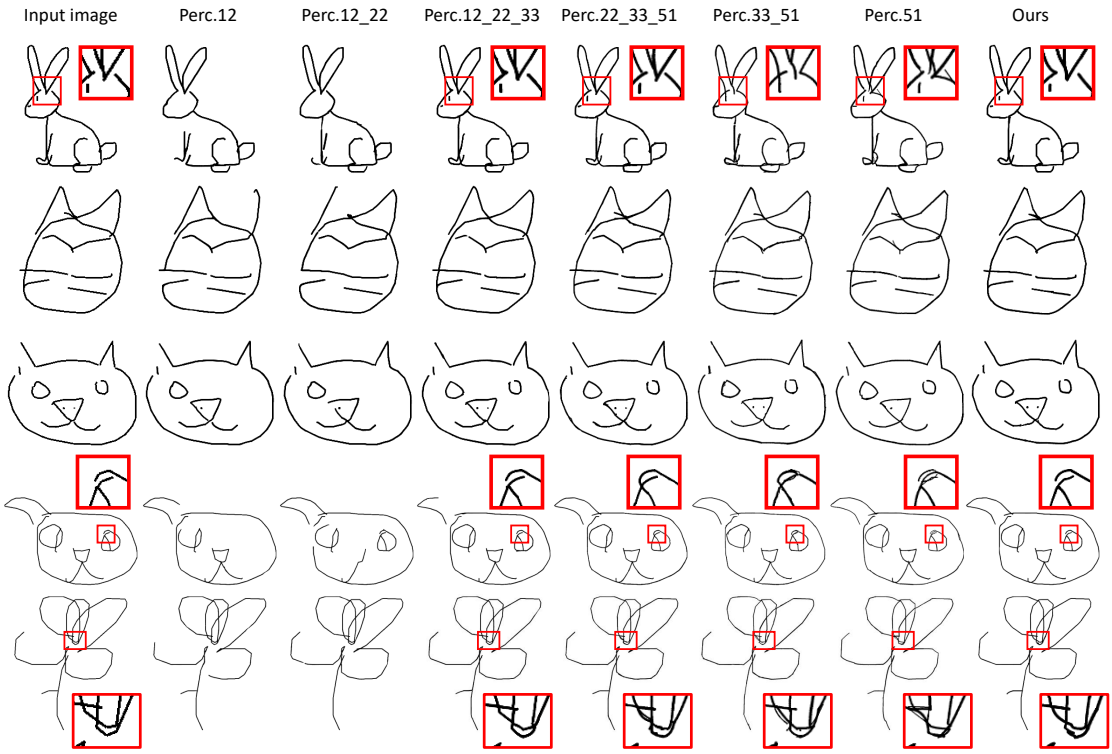

Fig. 11. Comparisons among different perceptual layer combinations. Our model uses "Perc.12_22_33_51".

Table 5. Quantitative results of different perceptual layer combinations. Our model uses $\cup(12, 22, 33, 51)$.

Perceptual Layers	Perceptual score(↓)	Chamfer distance(↓)
$\cup(12)$	1.481	3.556
$\cup(12, 22)$	1.399	3.242
$\cup(12, 22, 33)$	1.058	1.219
$\cup(12, 22, 33, 51)$	1.053	1.577
$\cup(22, 33, 51)$	1.076	1.150
$\cup(33, 51)$	1.513	2.995
$\cup(51)$	1.764	4.079

4.3.3 Loss Value Normalization. Figure 12 shows results from models with and without loss normalization. Model without loss normalization is similar to model dominated by deep perceptual layers, which produces results with good completeness but worse details.

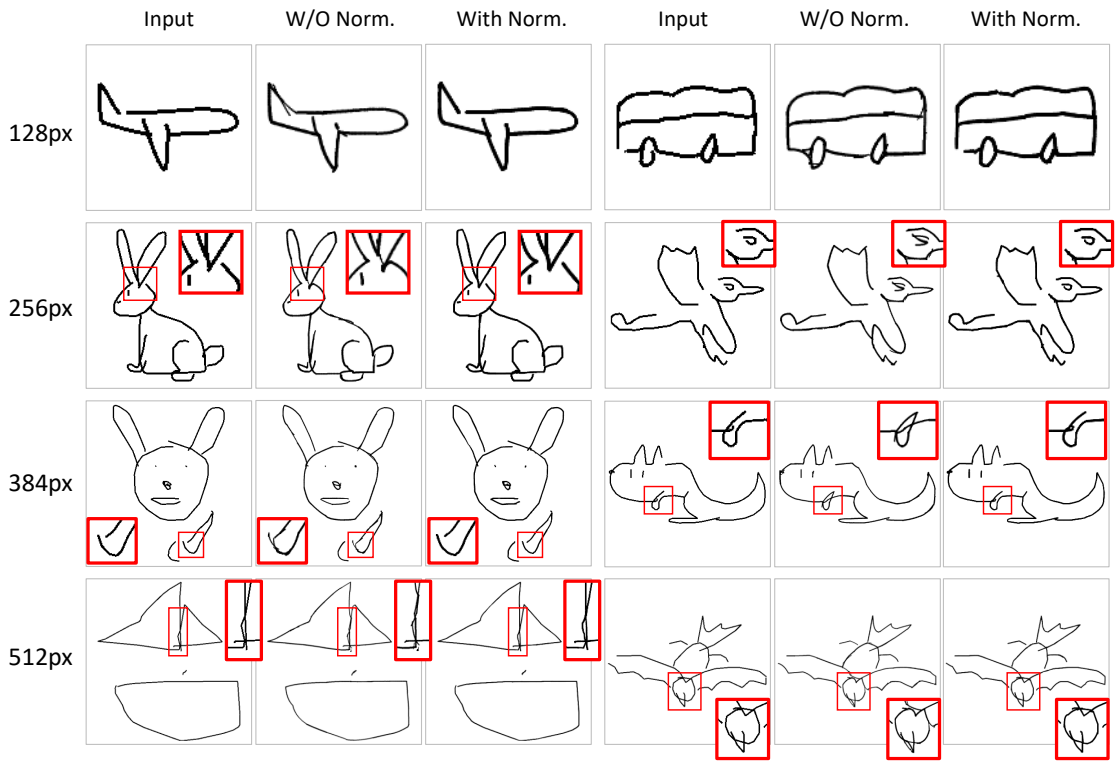

Fig. 12. Comparisons between models with and without loss normalization.

4.4 Effectiveness of Out-of-Bounds Penalty

Figure 13 shows results on vectorization from model without out-of-bounds penalty. Clearly, we can see empty canvases in the output results. The high out-of-bounds penalty loss (calculated but not applied to the total loss function) during training explains the reason. The window in our model moves in a relative manner, so it is possible that the window moves outside the valid region of input image. At the beginning of training, where the model hasn't learned how to draw well, it tends to move the window outside to produce an empty canvas so that a high raster loss can be avoided. To this end, out-of-bounds penalty is necessary to tell the model not to do like this and make it learn to draw in a correct direction.

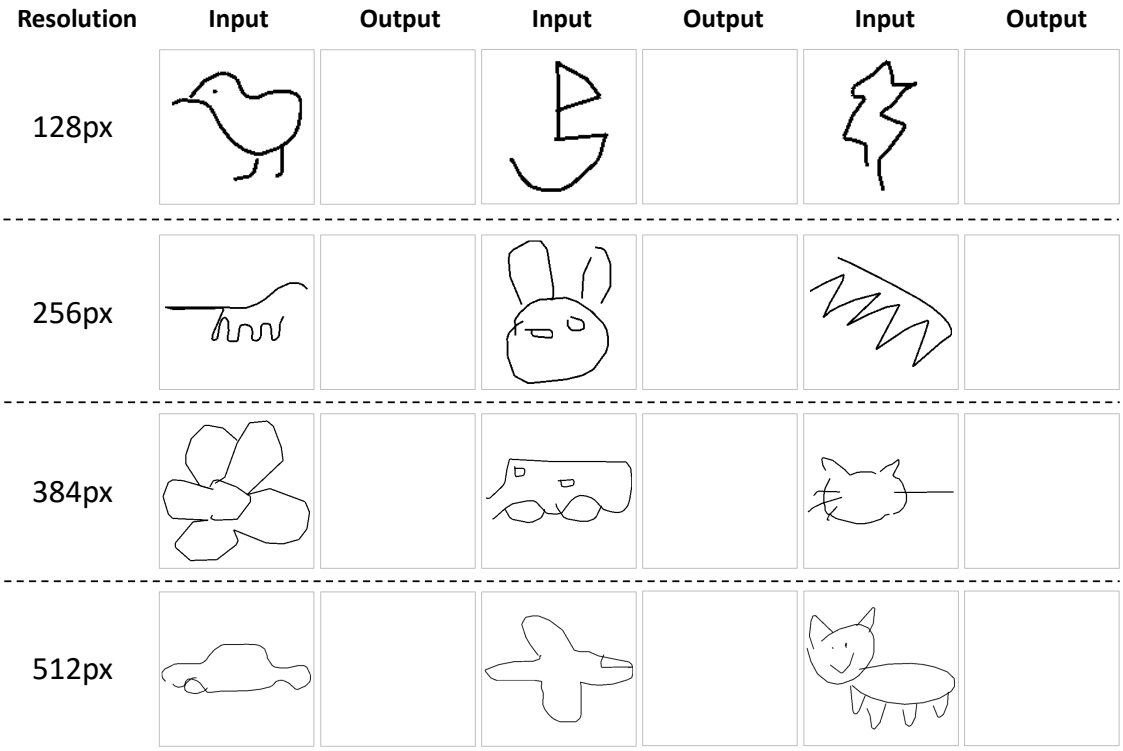

Fig. 13. Results of model without out-of-bounds penalty.

4.5 Stroke Regularization

4.5.1 Redundancy Removal. After obtaining the output stroke data in vector format, we are able to detect the redundant or overlapped strokes: we draw each stroke step by step onto the canvas. Then, for each new stroke, we calculate its intersection with the already drawn pixel region. Finally, if the intersection area is larger than half of the stroke area, we regard this stroke as an overlapped one. With this detecting method, we are able to remove the overlapped strokes as a post-processing operation. Figure 14 shows the performance of redundancy removal with different stroke regularization weights λ_{reg}.

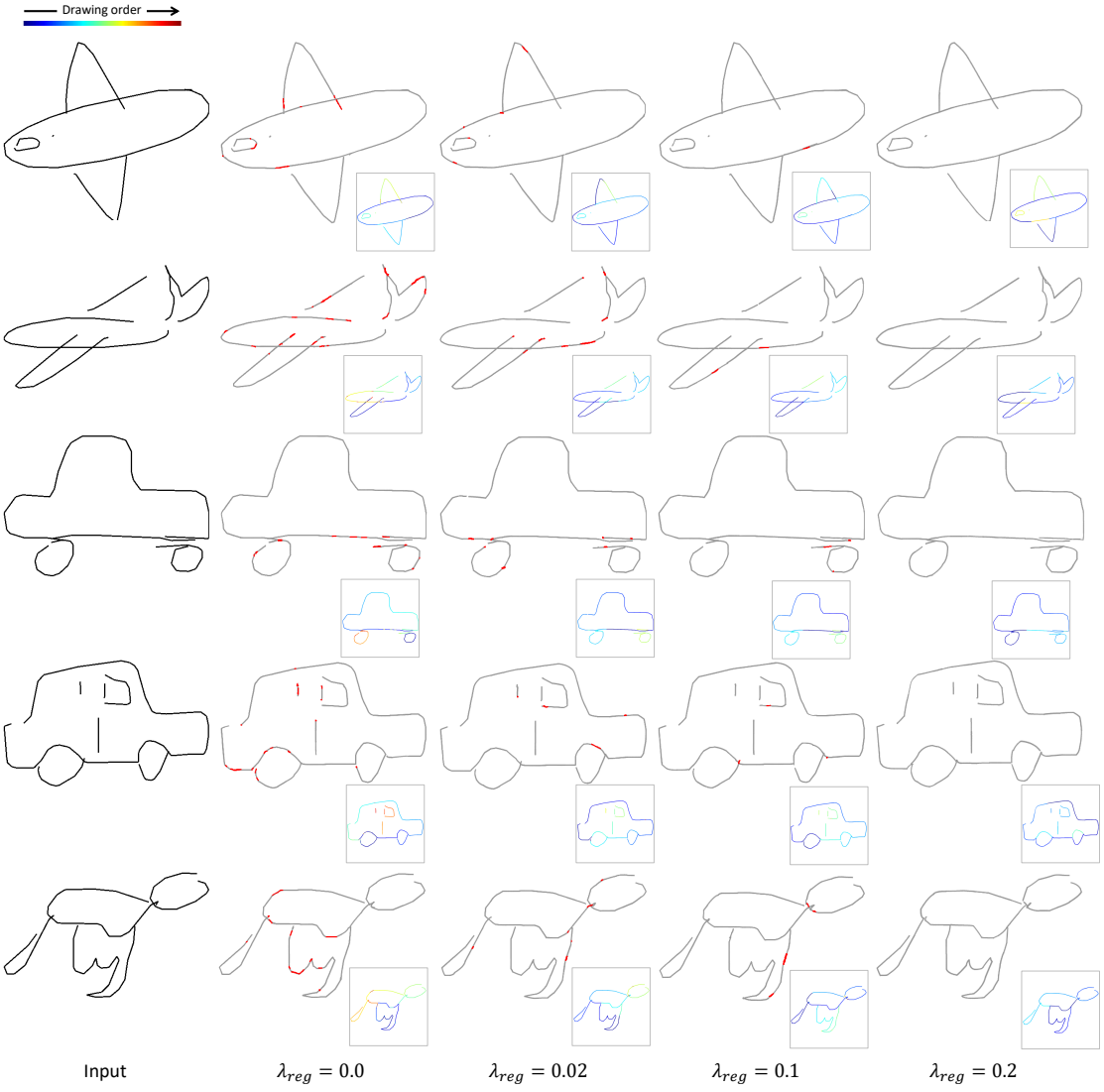

Fig. 14. Redundancy removal of stroke regularization with different weights λ_{reg}. Red strokes in the grayscale vector outputs represent the redundant or overlap strokes.

4.5.2 Compactness Improvement. Figure 15 shows the compactness improvement by stronger stroke regularization.

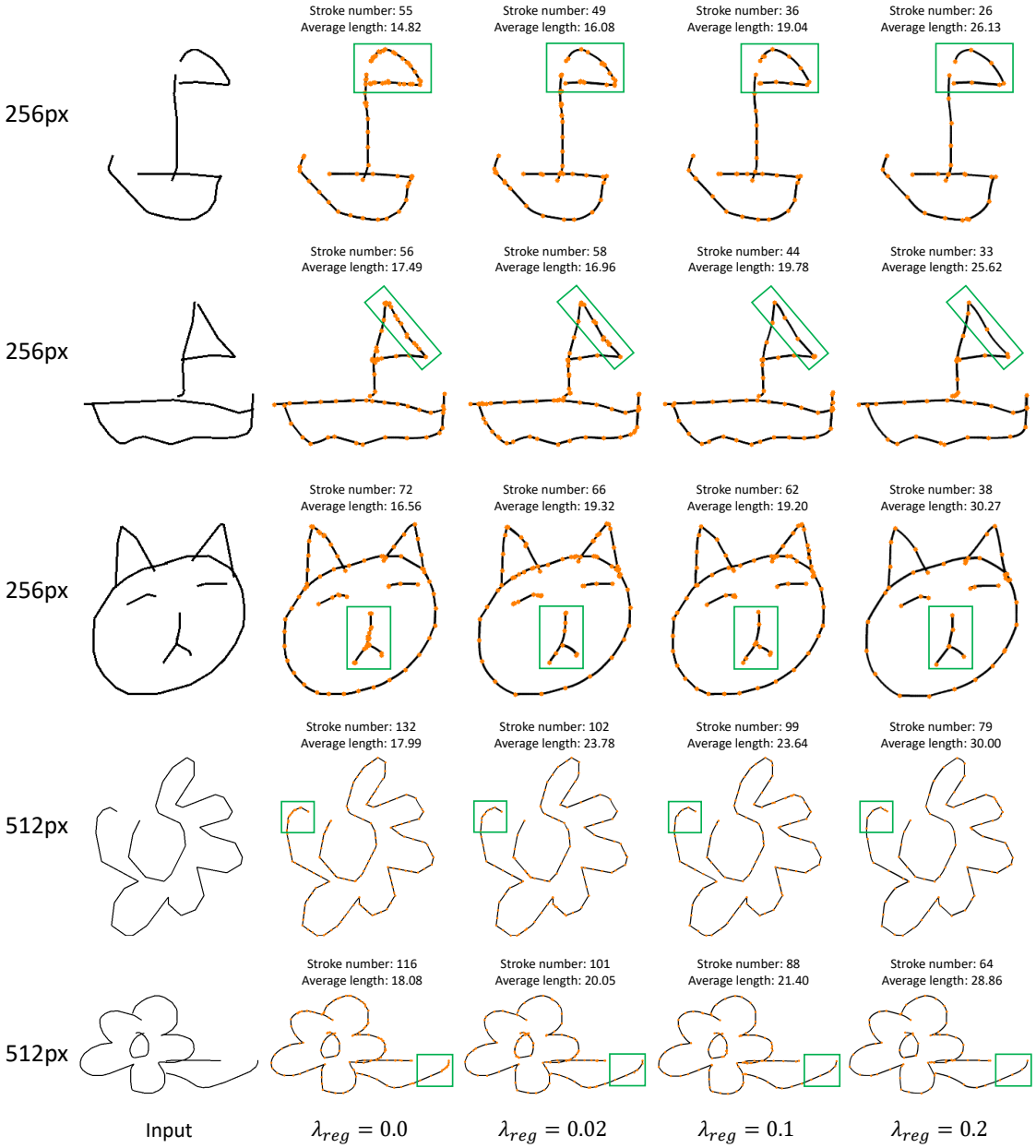

Fig. 15. Compactness improvement of stroke regularization with different weights λ_{reg}. Larger weights means stronger regularization ($\lambda_{reg} = 0.0$ indicates no regularization). Orange dots represent the endpoints of each drawn stroke.

4.5.3 Parameter Sensibility. Figure 16 and Figure 17 show visual comparisons among different stroke regularization weights λ_{reg}. Model with a large λ_{reg} suffers from bad completeness.

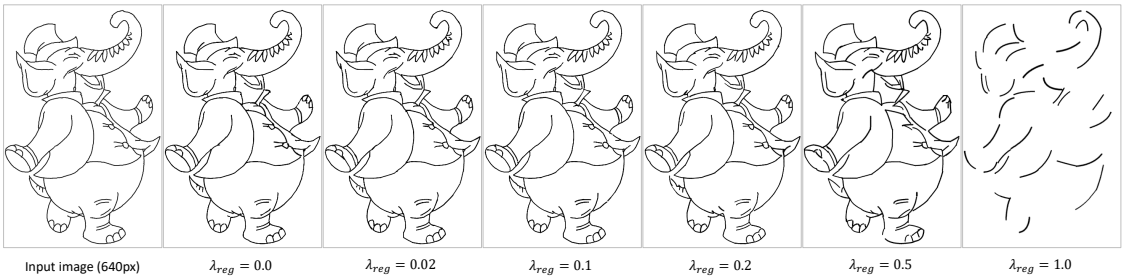

Fig. 16. Comparisons among different stroke regularization weights λ_{reg}.

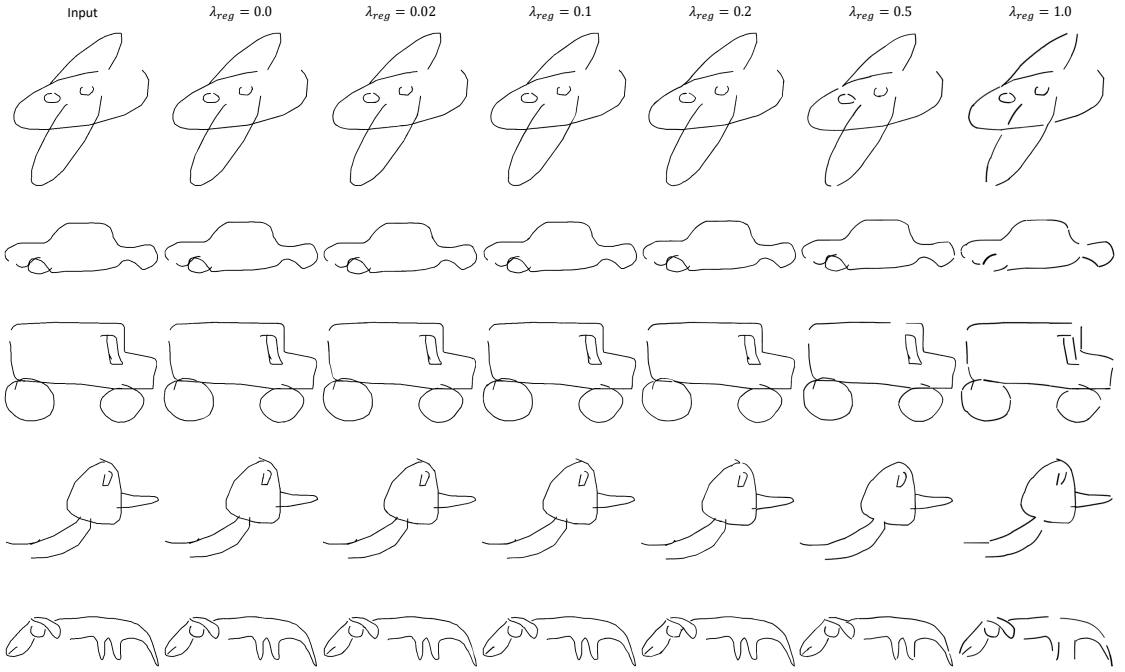

Fig. 17. Comparisons among different stroke regularization weights λ_{reg}.

4.6 Vectorization

4.6.1 Qualitative Results. We compare with Learning-To-Paint [Huang et al. 2019] on sketches from QuickDraw dataset of different resolutions and different stroke numbers. Results are shown in Fig. 18, Fig. 19 and Fig. 20. More results and comparisons with existing vectorization methods on real clean sketches are shown from Fig. 21 to Fig. 27.

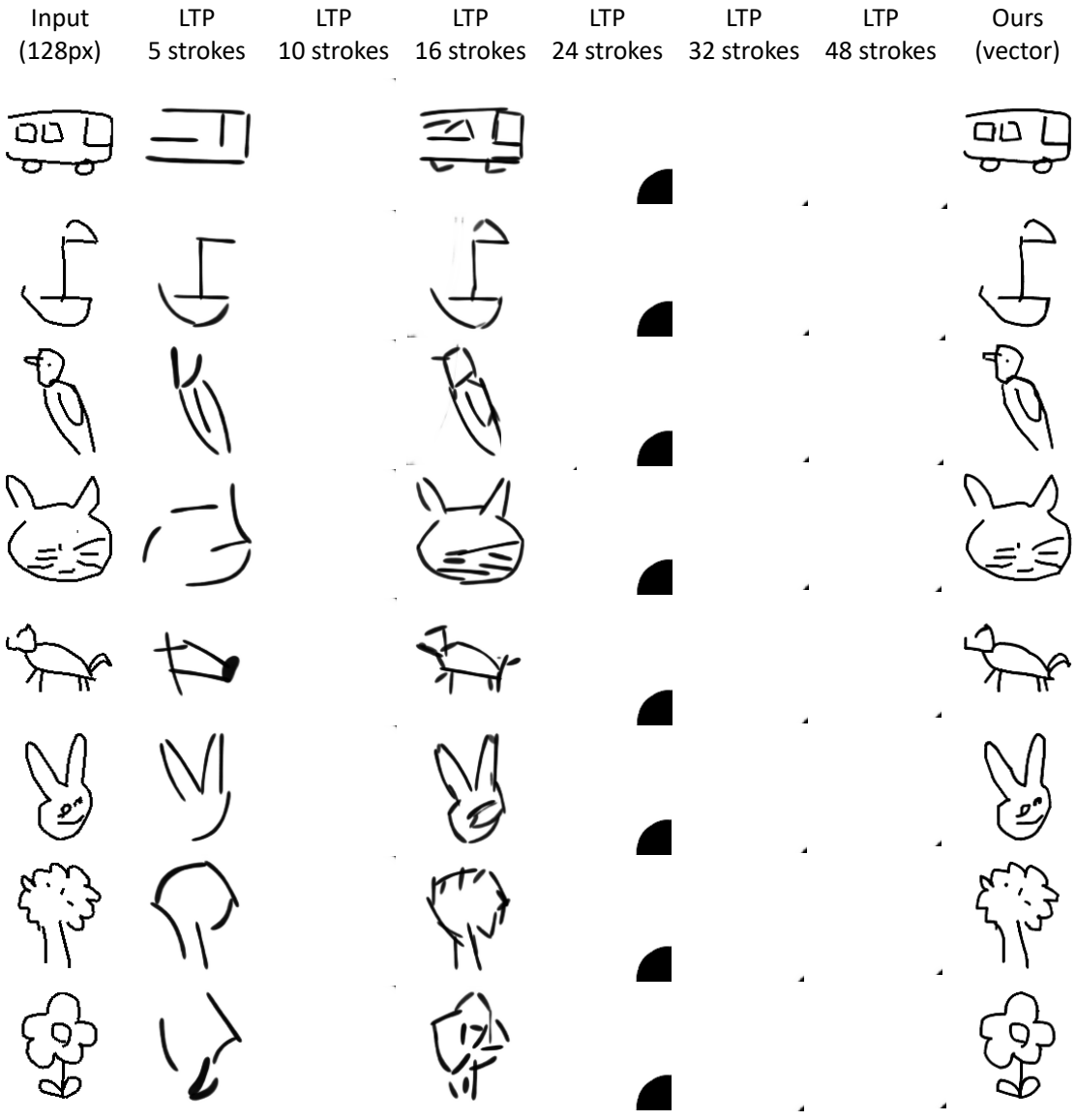

Fig. 18. Comparisons with Learning-To-Paint [Huang et al. 2019] (LTP) with different stroke numbers on clean sketch vectorization.

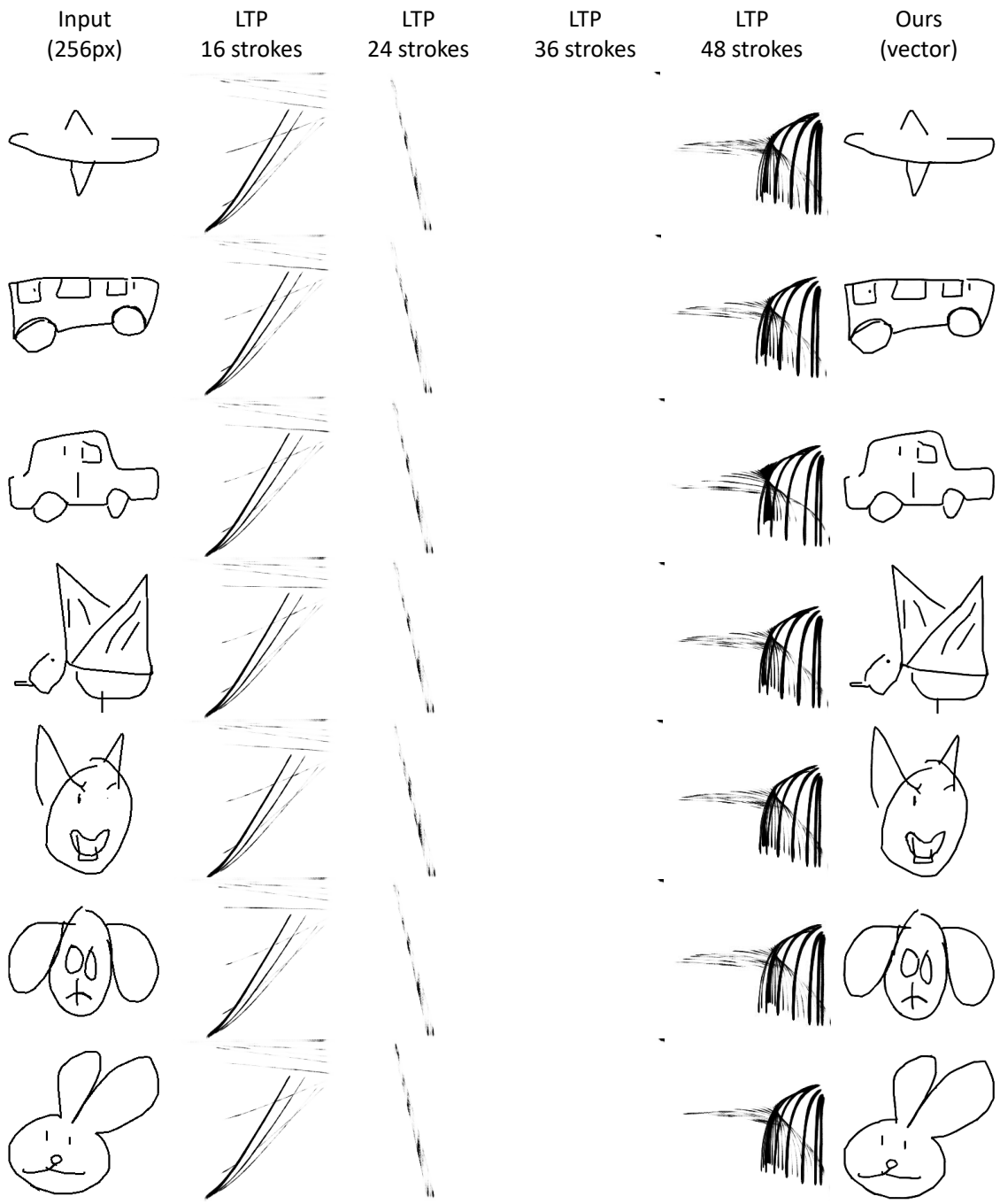

Fig. 19. Comparisons with Learning-To-Paint [Huang et al. 2019] (LTP) with different stroke numbers on clean sketch vectorization.

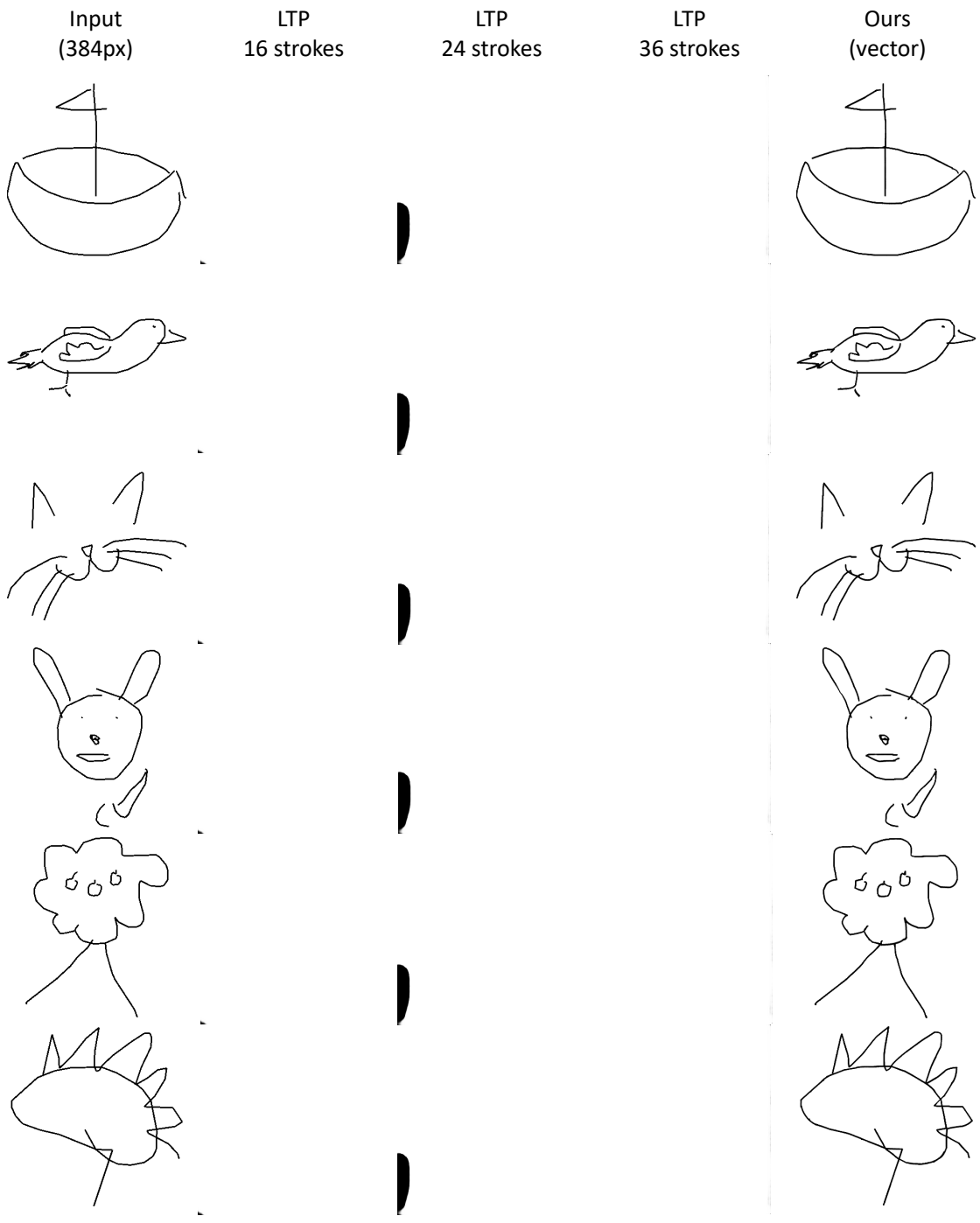

Fig. 20. Comparisons with Learning-To-Paint [Huang et al. 2019] (LTP) with different stroke numbers on clean sketch vectorization.

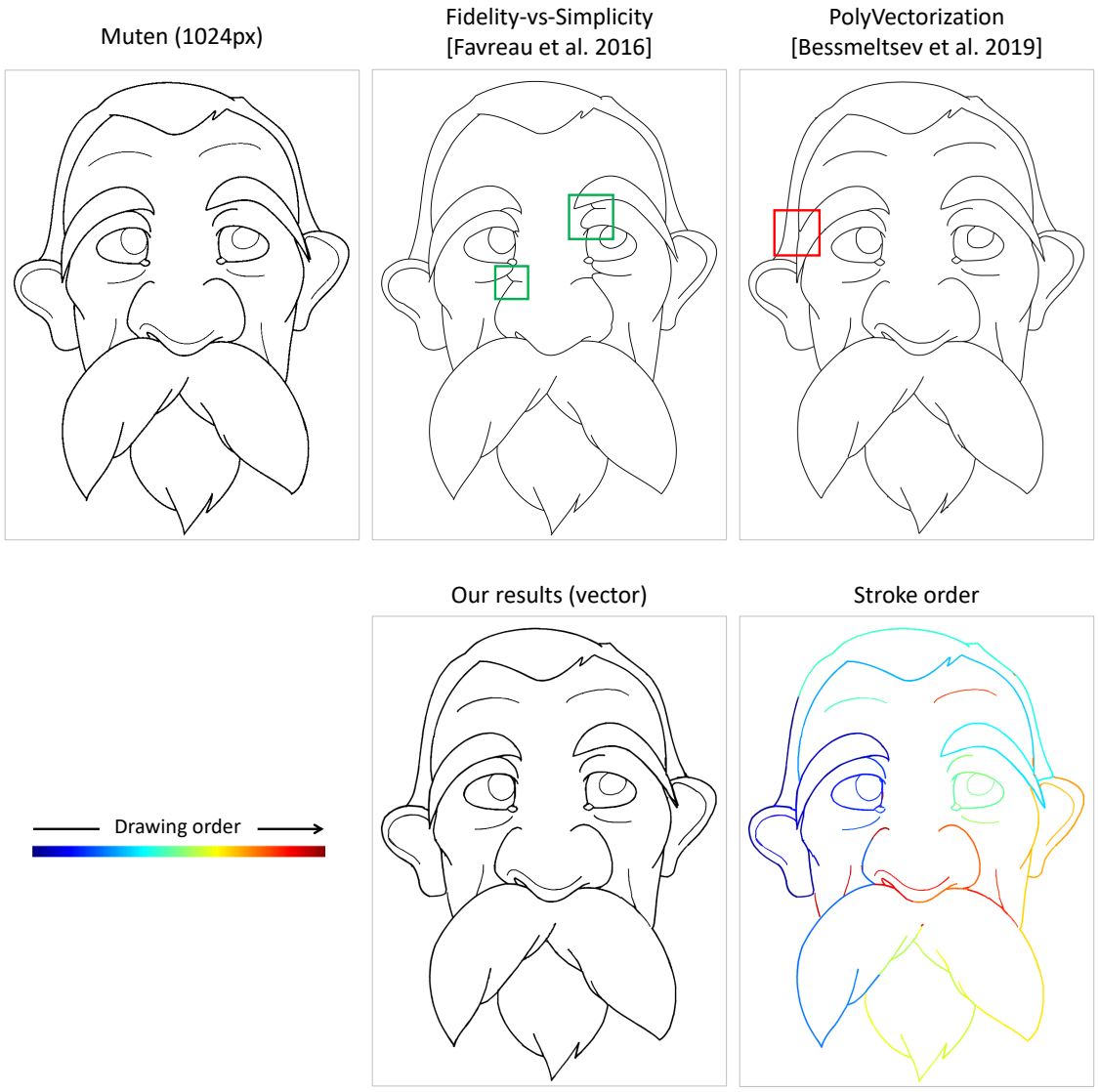

Fig. 21. Comparisons with existing approaches on real clean sketch vectorization.

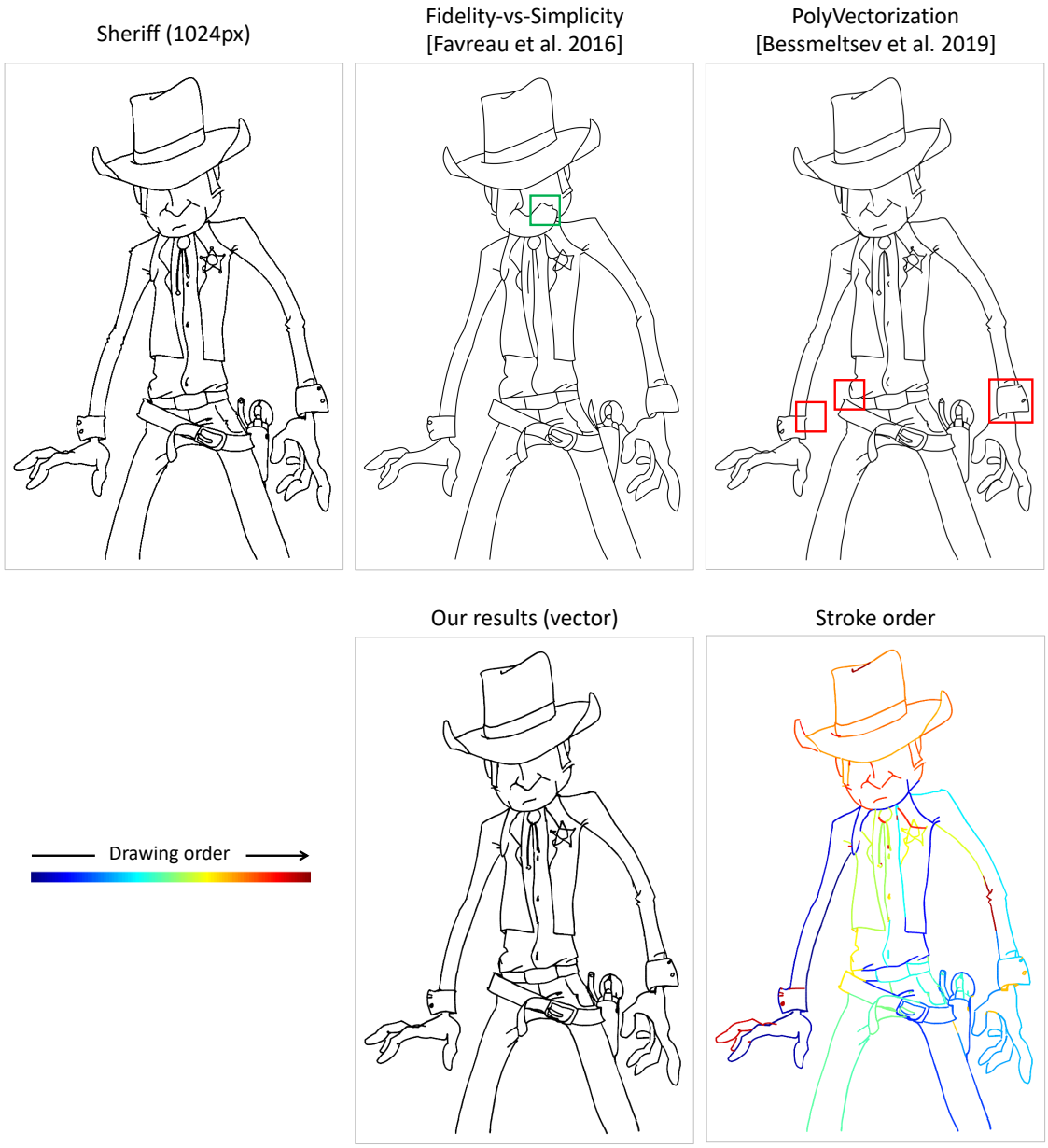

Fig. 22. Comparisons with existing approaches on real clean sketch vectorization.

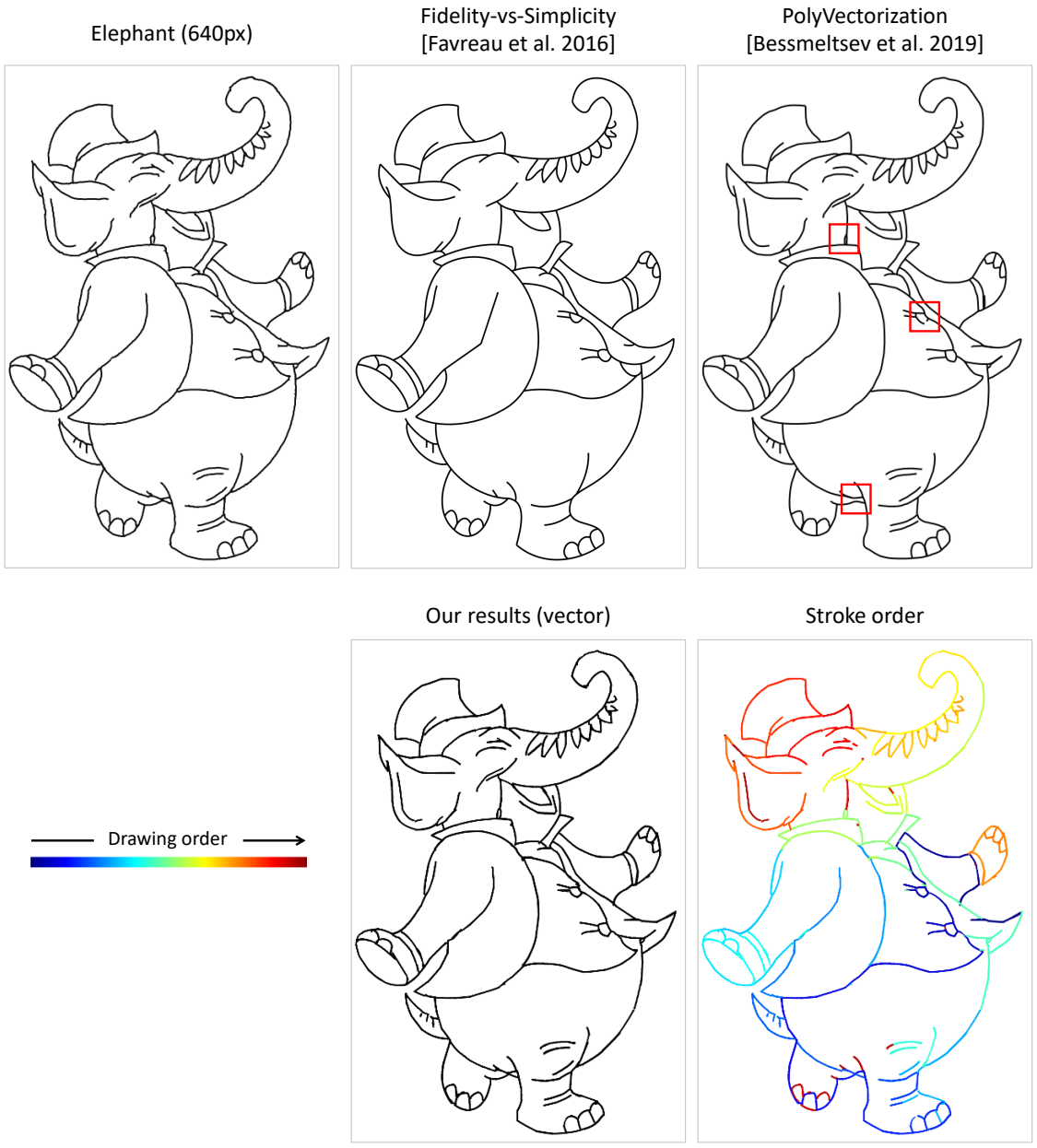

Fig. 23. Comparisons with existing approaches on real clean sketch vectorization.

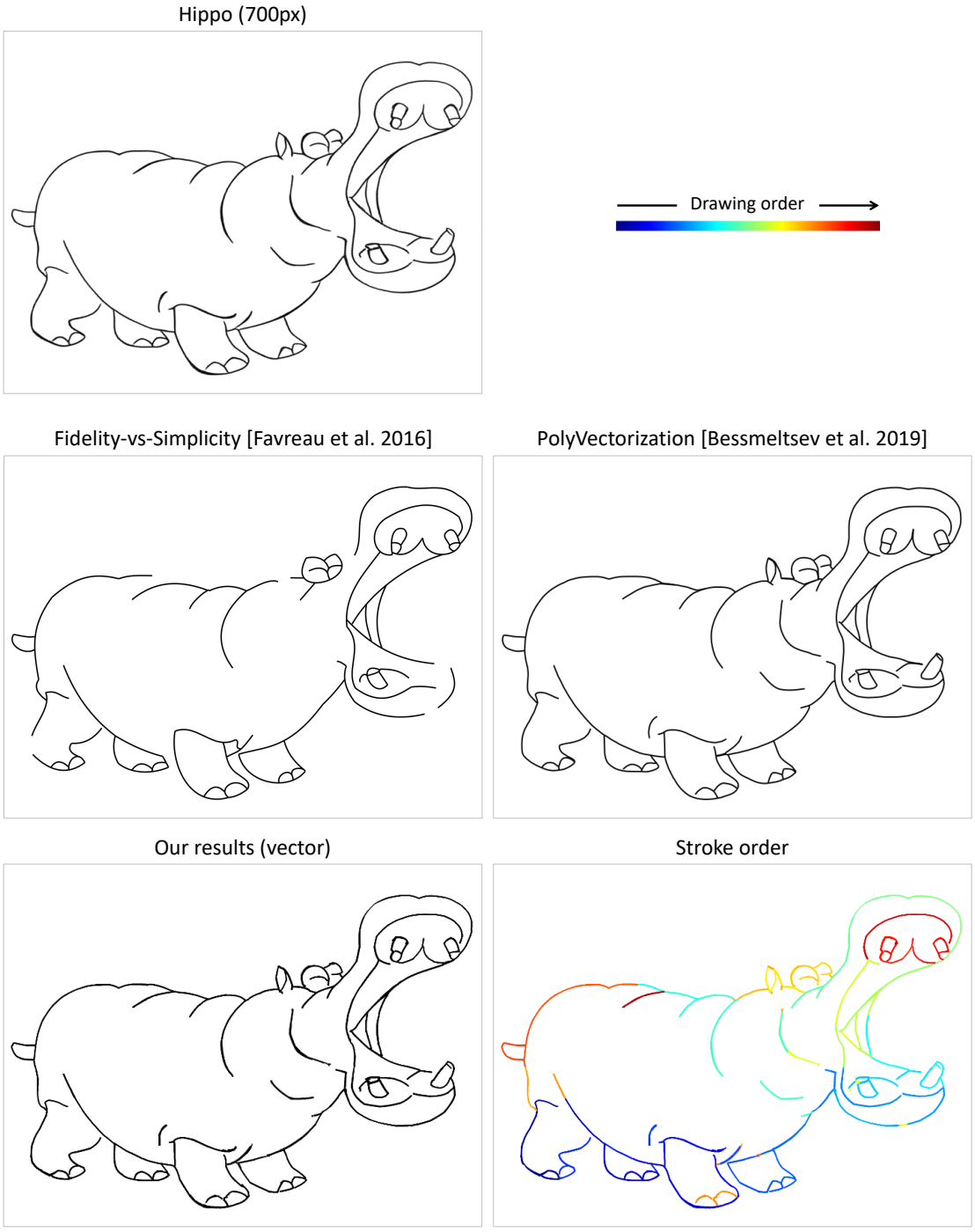

Fig. 24. Comparisons with existing approaches on real clean sketch vectorization.

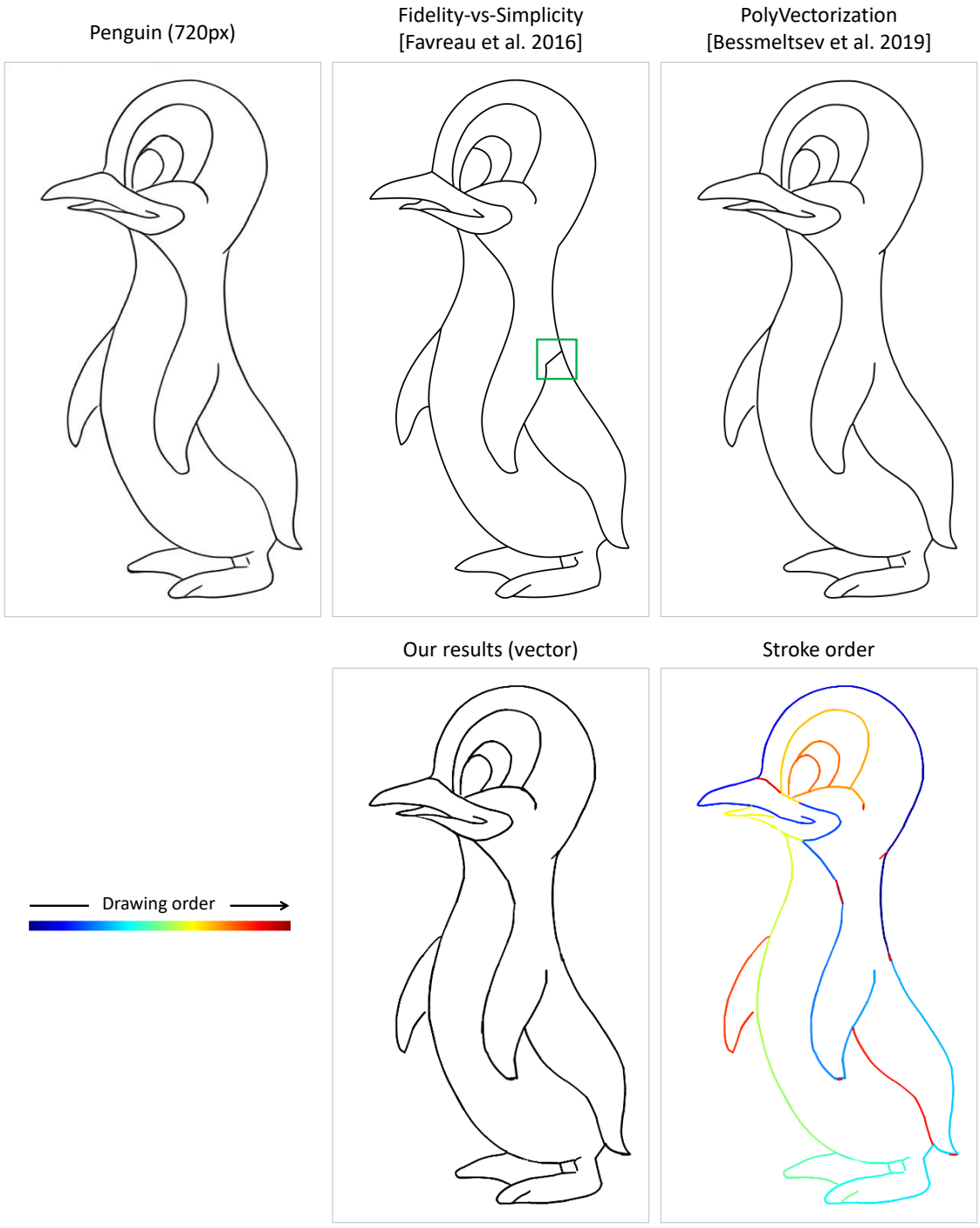

Fig. 25. Comparisons with existing approaches on real clean sketch vectorization.

Fig. 26. Comparisons with existing approaches on real clean sketch vectorization.

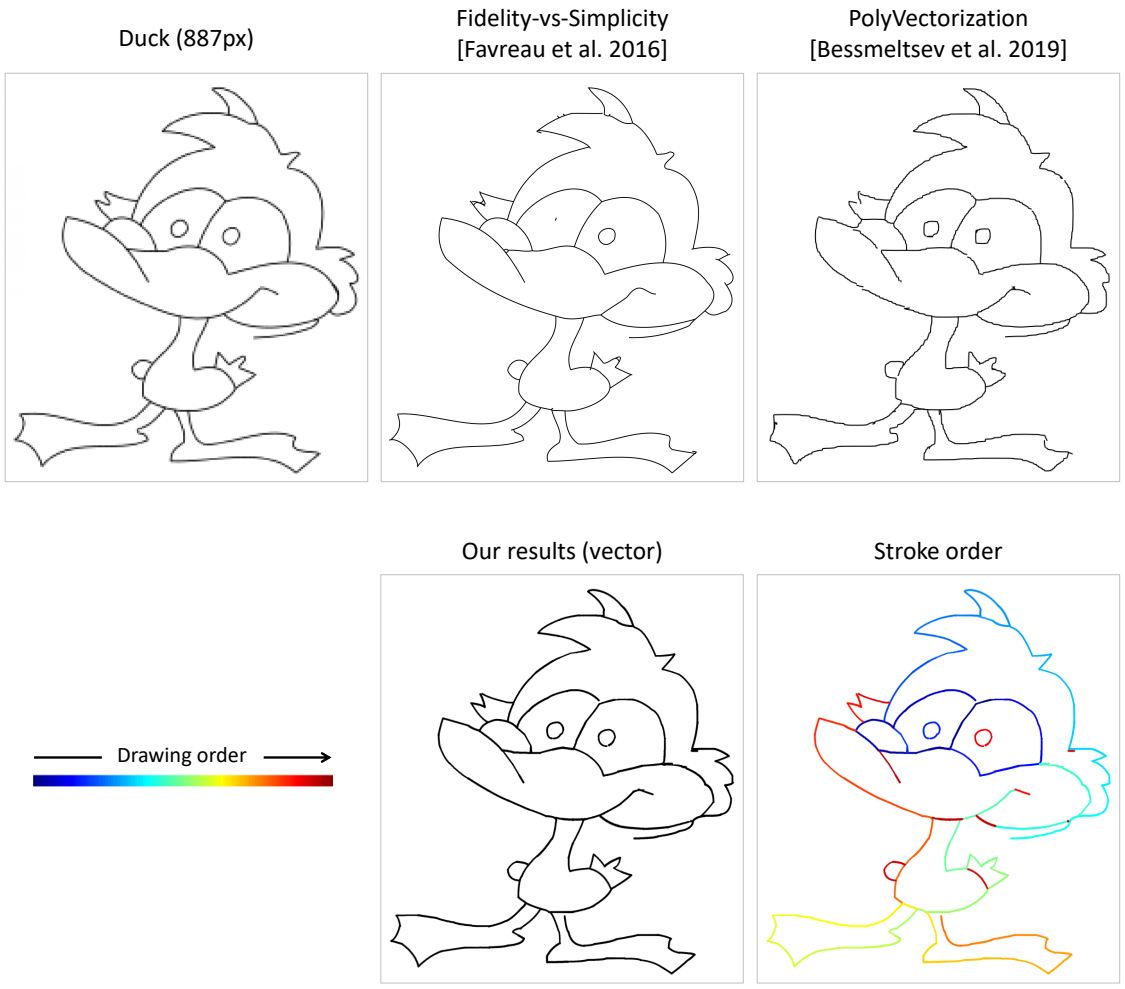

Fig. 27. Comparisons with existing approaches on real clean sketch vectorization.

4.7 Rough Sketch Simplification

We compare with Learning-To-Paint [Huang et al. 2019] on "Rough QuickDraw" dataset of different resolutions and different stroke numbers. Results are shown from Fig. 28 to Fig. 31. More results and comparisons with existing methods on complex rough sketches are shown in Fig. 32, Fig. 33 and Fig. 34.

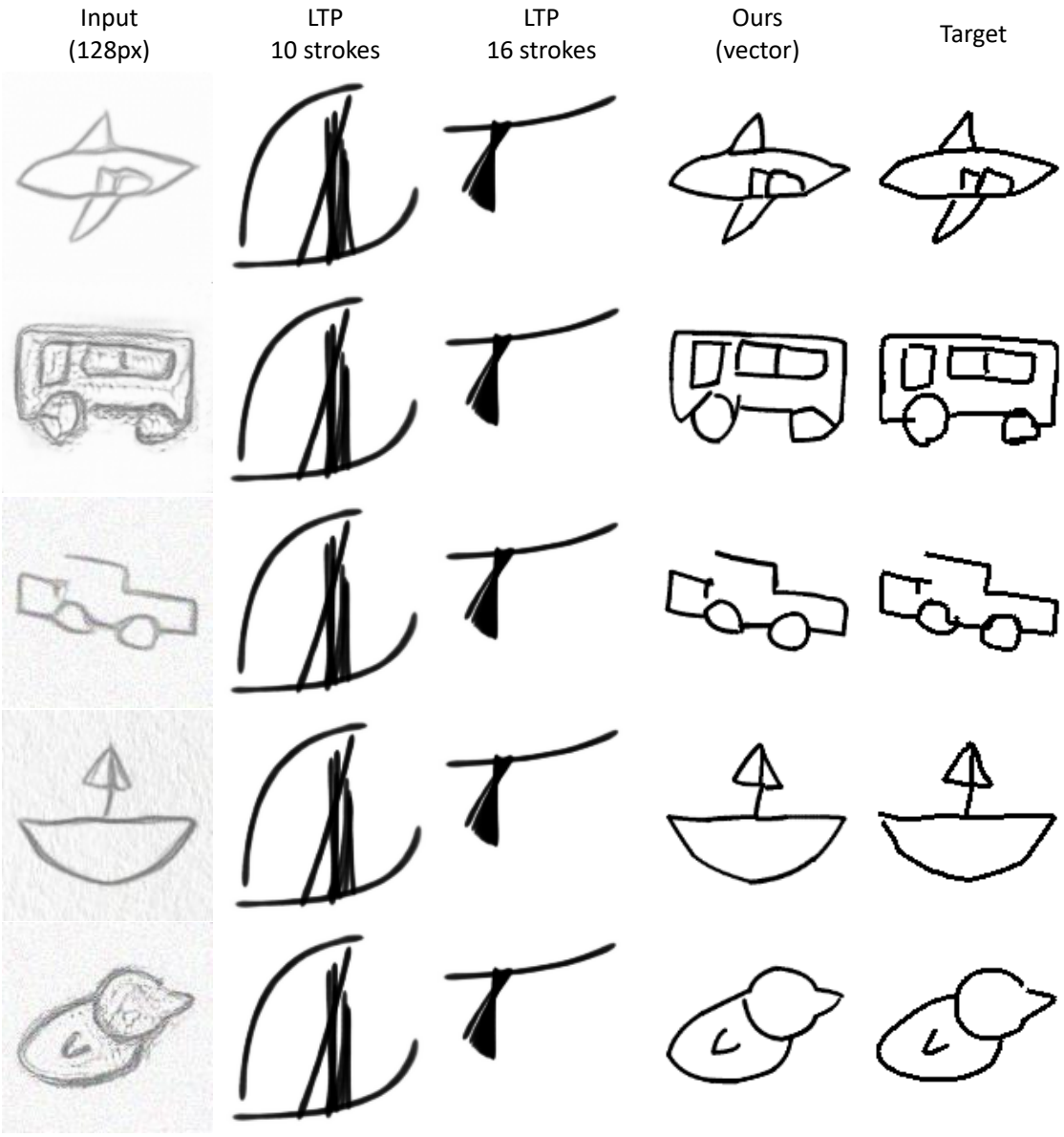

Fig. 28. Comparisons with Learning-To-Paint [Huang et al. 2019] (LTP) with different stroke numbers on rough sketch simplification.

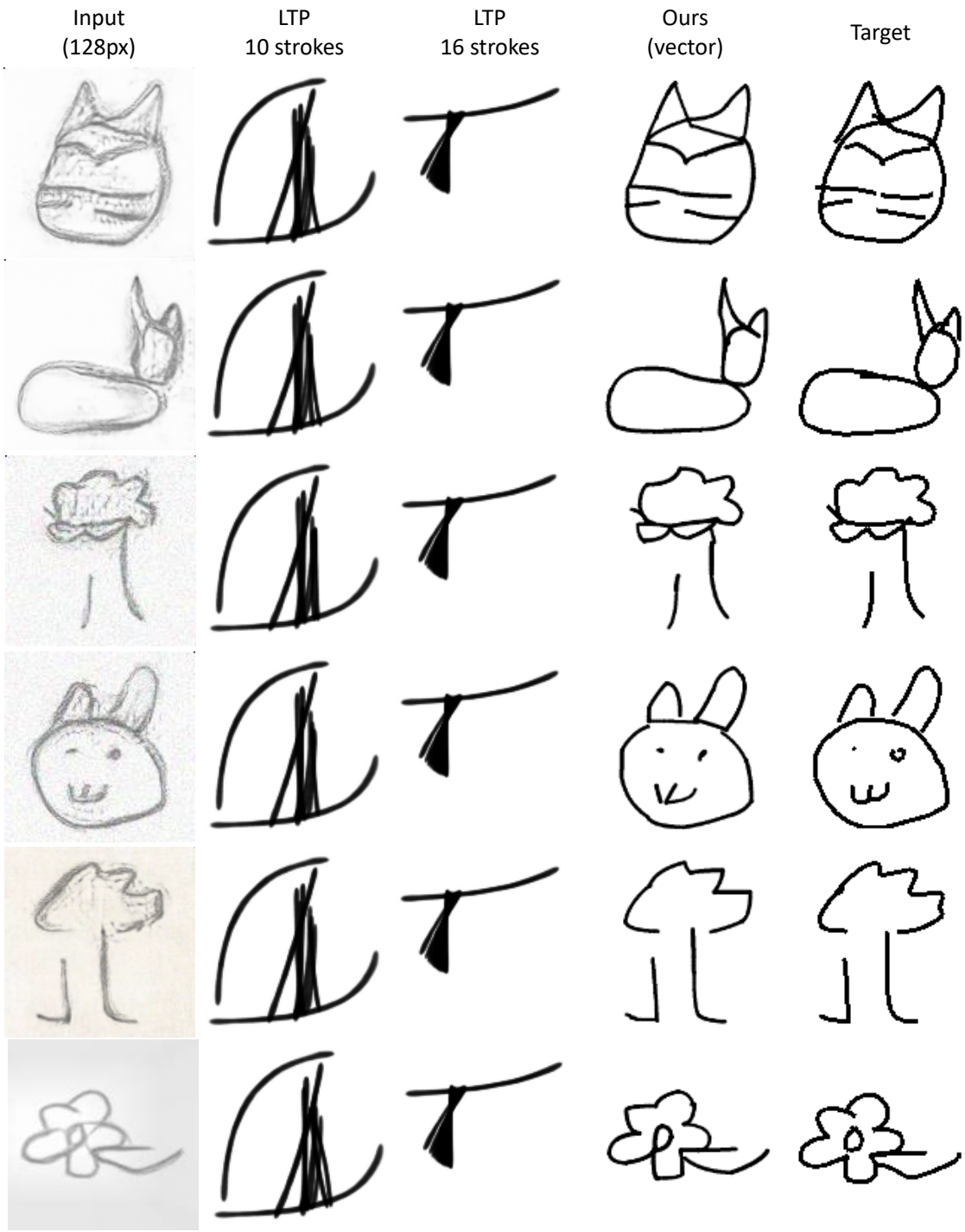

Fig. 29. Comparisons with Learning-To-Paint [Huang et al. 2019] (LTP) with different stroke numbers on rough sketch simplification.

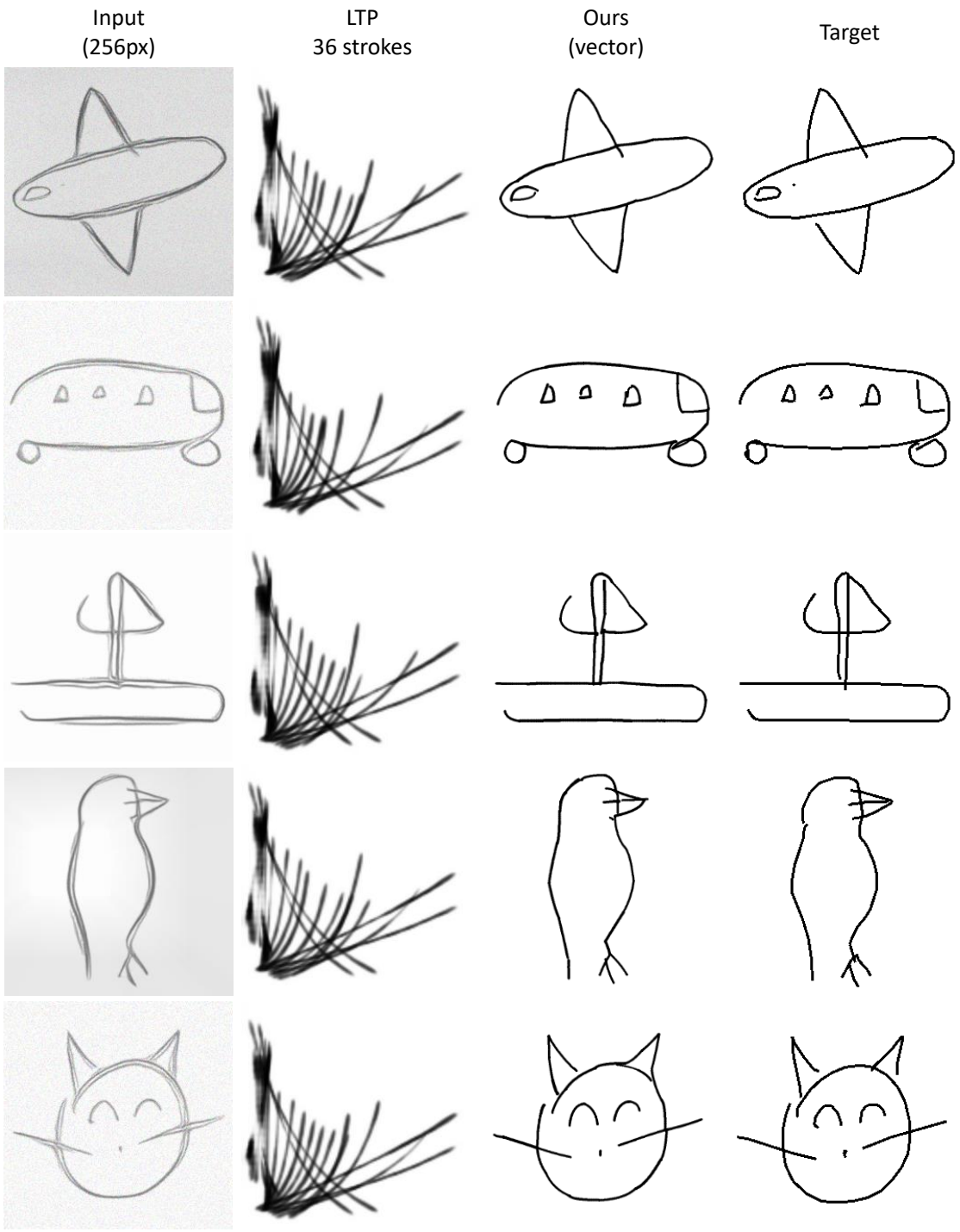

Fig. 30. Comparisons with Learning-To-Paint [Huang et al. 2019] (LTP) with different stroke numbers on rough sketch simplification.

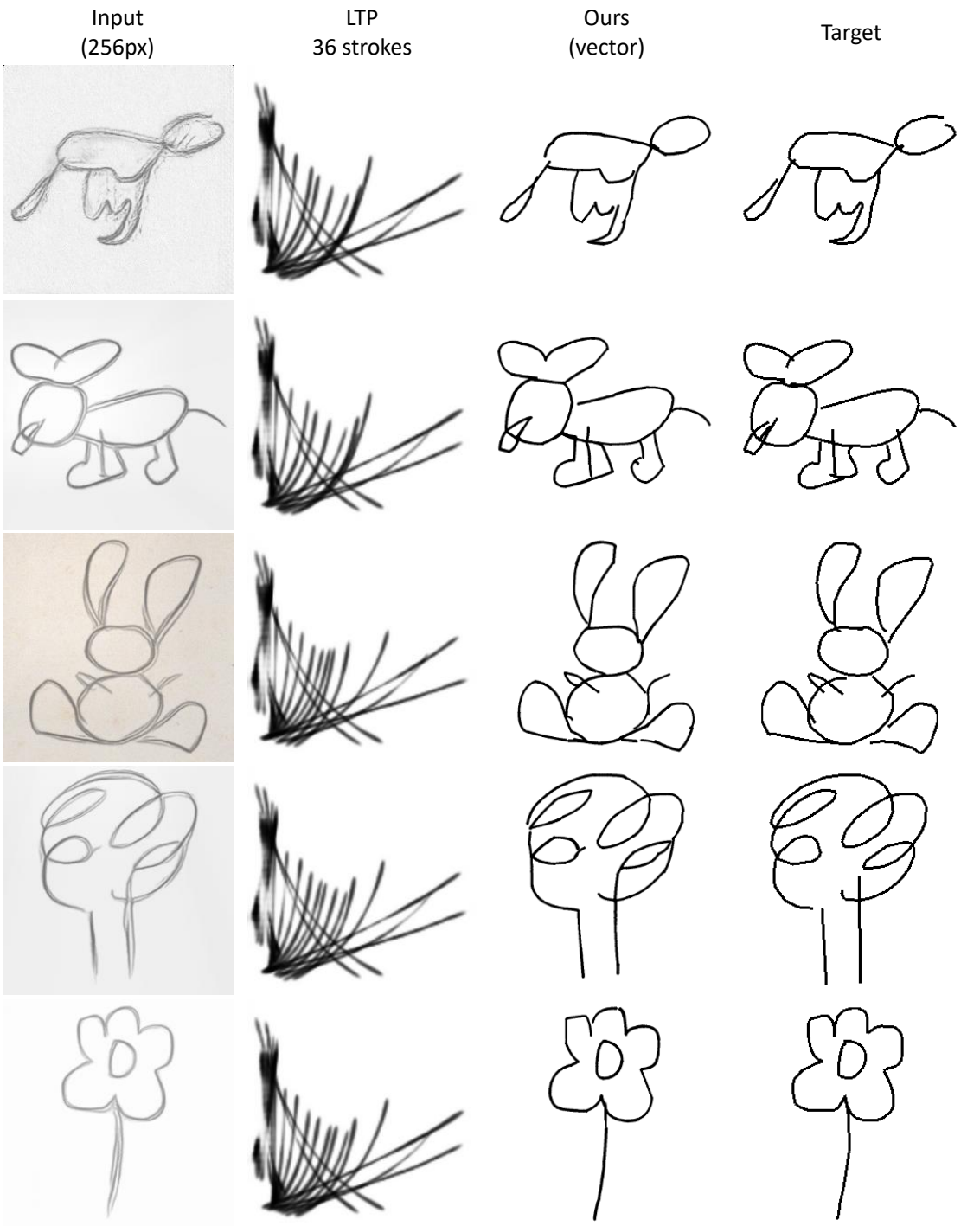

Fig. 31. Comparisons with Learning-To-Paint [Huang et al. 2019] (LTP) with different stroke numbers on rough sketch simplification.

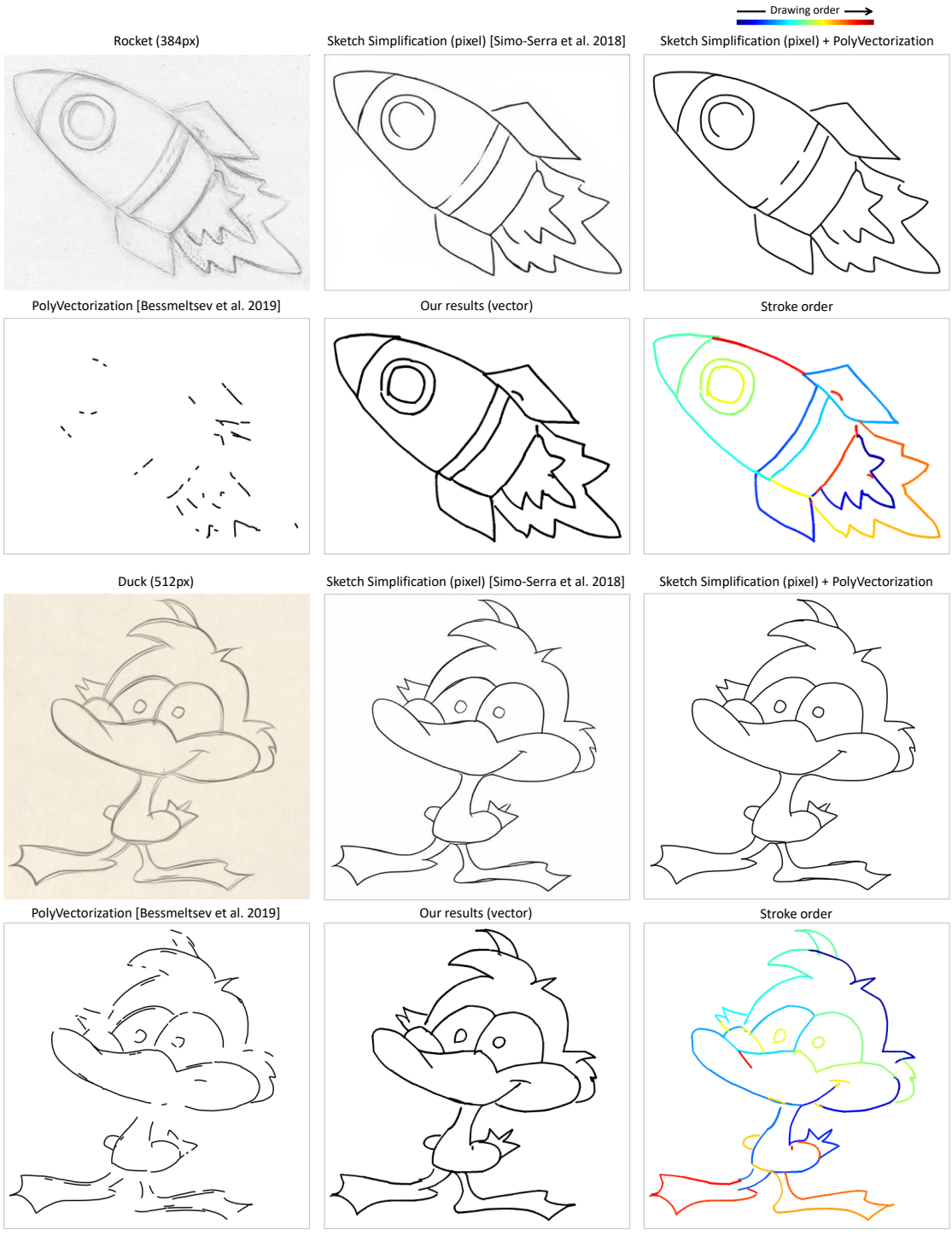

Fig. 32. Comparisons with existing approaches on rough sketch simplification.

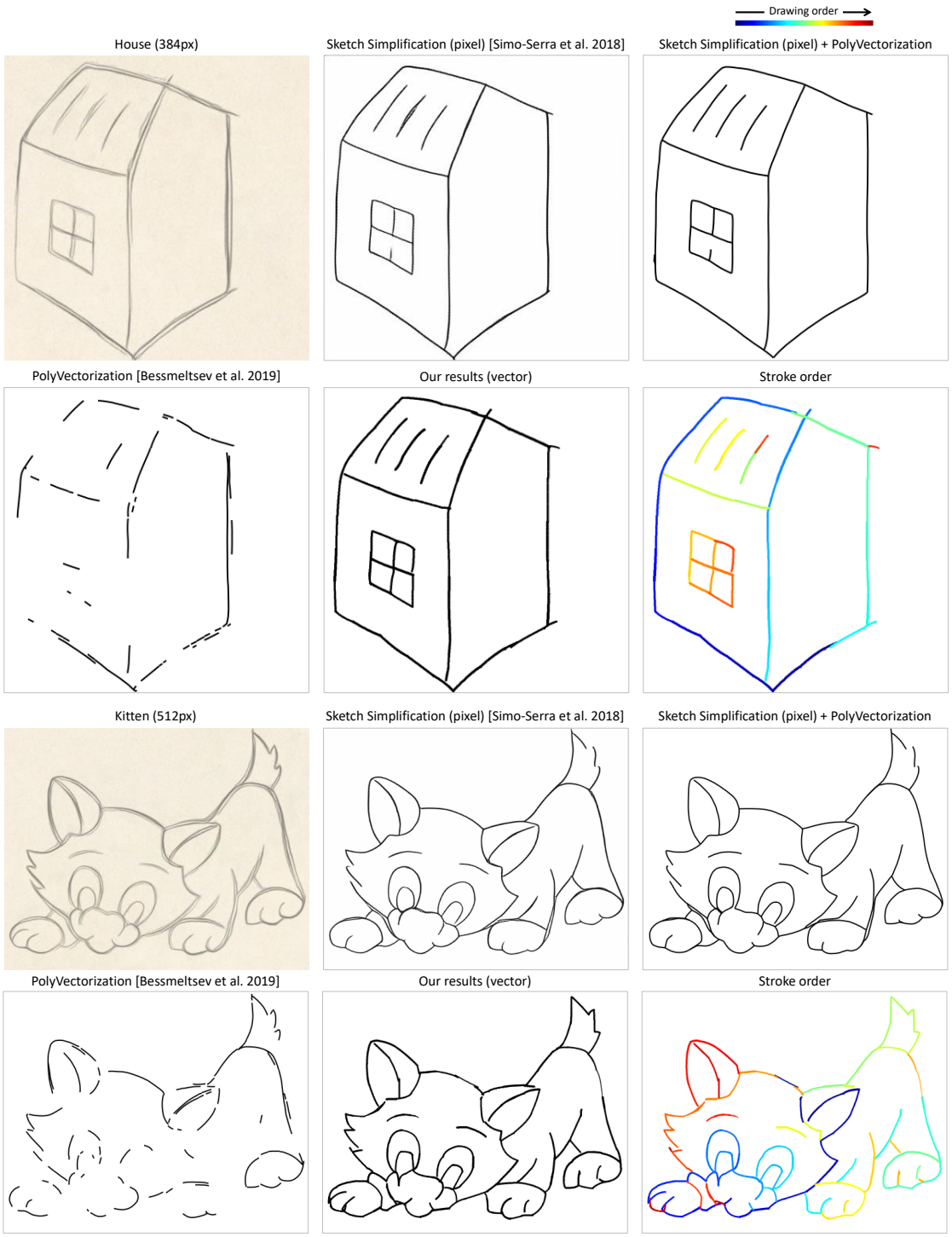

Fig. 33. Comparisons with existing approaches on rough sketch simplification.

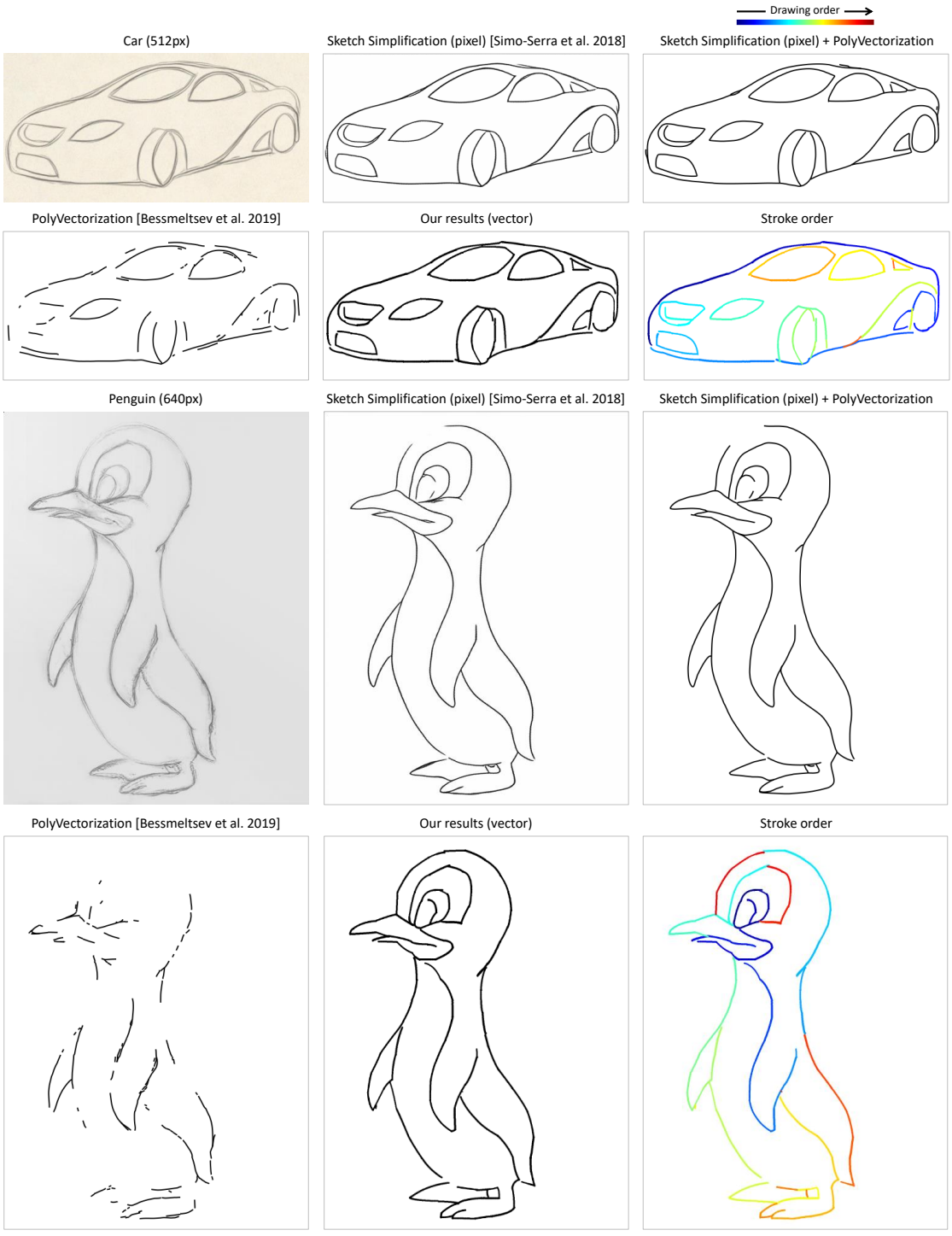

Fig. 34. Comparisons with existing approaches on rough sketch simplification.

4.8 Photograph to Line Drawing

More results and comparisons with existing methods are shown in Fig. 35 and Fig. 36.

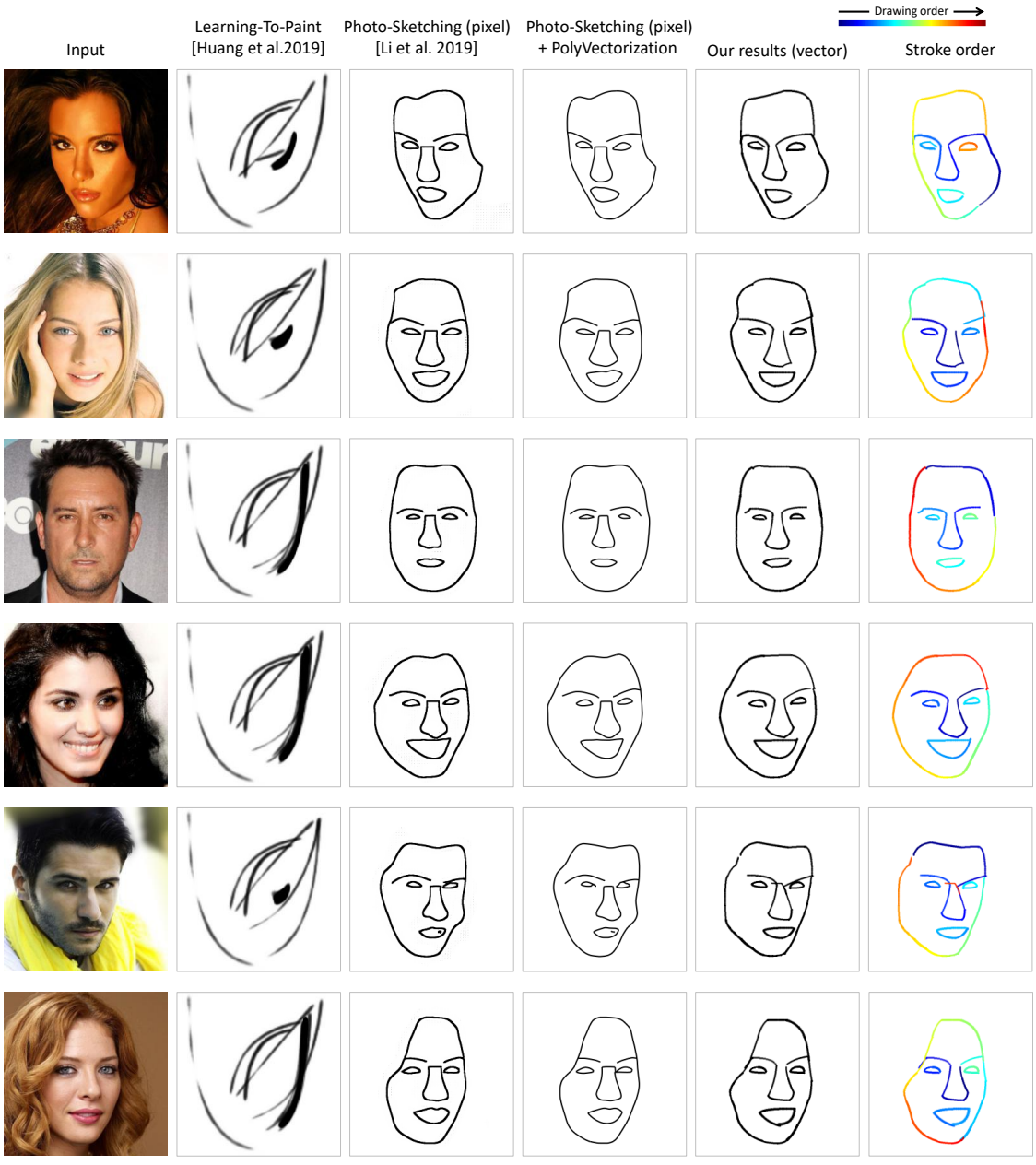

Fig. 35. Comparisons with existing approaches on photograph to line drawing.

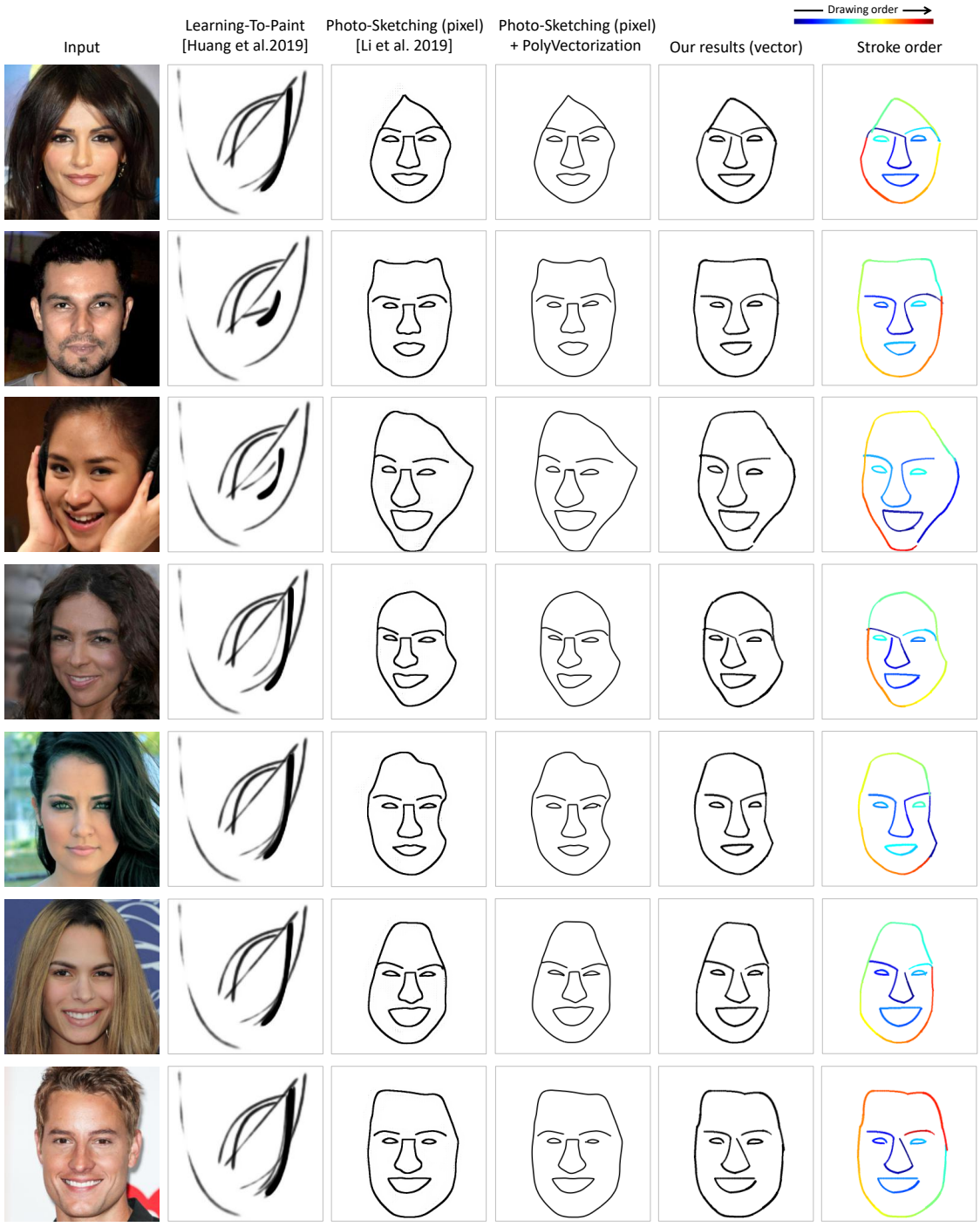

Fig. 36. Comparisons with existing approaches on photograph to line drawing.

REFERENCES

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

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.

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)*.

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

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

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