Learning to Simplify: Fully Convolutional Networks for Rough Sketch Cleanup

Edgar Simo-Serra* Satoshi Iizuka* Kazuma Sasaki Hiroshi Ishikawa

Waseda University

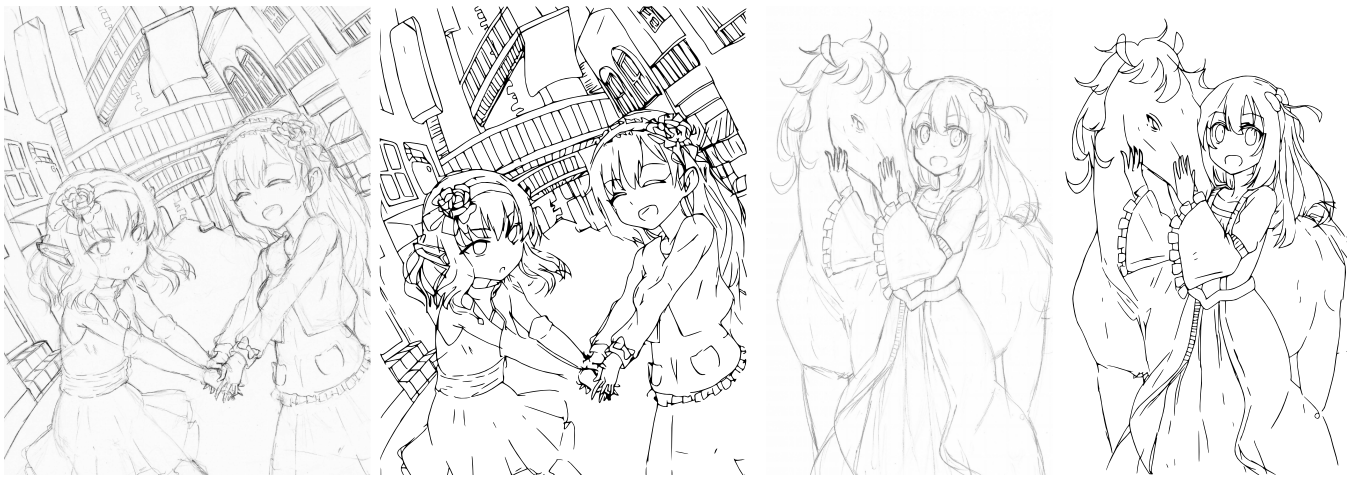


Figure 1: *Example of our sketch simplification results on two different images. Our approach automatically converts the rough pencil sketches on the left to the clean vector results on the right.*

Abstract

In this paper, we present a novel technique to simplify sketch drawings based on learning a series of convolution operators. In contrast to existing approaches that require vector images as input, we allow the more general and challenging input of rough raster sketches such as those obtained from scanning pencil sketches. We convert the rough sketch into a simplified version which is then amendable for vectorization. This is all done in a fully automatic way without user intervention. Our model consists of a fully convolutional neural network which, unlike most existing convolutional neural networks, is able to process images of any dimensions and aspect ratio as input, and outputs a simplified sketch which has the same dimensions as the input image. In order to teach our model to simplify, we present a new dataset of pairs of rough and simplified sketch drawings. By leveraging convolution operators in combination with efficient use of our proposed dataset, we are able to train our sketch simplification model. Our approach naturally overcomes the limitations of existing methods, *e.g.*, vector images as input and long computation time; and we show that meaningful simplifications can be obtained for many different test cases. Finally, we validate our results with a user study in which we greatly outperform similar approaches and establish the state of the art in sketch simplification of raster images.

Keywords: sketch simplification, convolutional neural network

Concepts: •**Applied computing** → **Fine arts;** •**Computing methodologies** → *Neural networks;*

*The authors assert equal contribution and joint first authorship.

SIGGRAPH '16 Technical Paper,, July 24-28, 2016, Anaheim, CA,
ISBN: 978-1-4503-4279-7/16/07
DOI: http://dx.doi.org/10.1145/2897824.2925972

ACM Reference Format
Simo-Serra, E., Iizuka, S., Sasaki, K., Ishikawa, H. 2016. Learning to Simplify: Fully Convolutional Networks for Rough Sketch Cleanup. ACM Trans. Graph. 35, 4, Article 121 (July 2016), 11 pages.
DOI = 10.1145/2897824.2925972 http://doi.acm.org/10.1145/2897824.2925972.

1 Introduction

Sketching is the fundamental first step for expressing artistic ideas and beginning an iterative process of design refinement. It allows artists to quickly render their ideas on paper. The priority is to express concepts and ideas quickly, rather than exhibit fine details, which leads to coarse and rough sketches. After an initial sketch, feedback is used to iteratively refine the design until the final piece is produced. This iterative refinement forces artists to have to continuously clean up their rough sketches into simplified drawings and thus implies an additional workload. The process of manually tracing the rough sketch to produce a clean drawing, as one would expect, is fairly tedious and time-consuming.

In this work we aim at automatically converting rough sketches into simplified clean drawings. Unlike existing methods, **we are able to directly simplify rough raster sketches**, which is fundamental as a large segment of the artist population uses traditional tools such as pencil-and-paper rather than digital tablets. Our approach, based on Convolutional Neural Networks (CNN), consists of processing the image with a series of convolution operations that are able to group the rough sketch lines and output a simplification directly. The kernels used for the convolutions are learnt from a novel dataset of rough images with their associated simplifications which we also present in this work. This data-driven approach has two import advantages: first of all, it learns all the necessary heuristics necessary for sketch simplification automatically from the training data, and secondly, convolutions can be implemented efficiently on the GPU allowing for processing times of under a second for most images. Unlike most standard CNN architectures used for process-

ing images which use layers that are fully connected to the previous layer, ours uses only convolutional layers, based on sparse connections, which allow our approach to process images of any resolution or aspect ratio efficiently.

Once a rough sketch is processed by our model to obtain a simplified sketch, it is then possible to use existing vectorization approaches to convert the raster output image to a vector image. As we will show, directly vectorizing the rough sketch leads to very noisy images, while vectorizing the output of our approach leads to clean images that can be used as is. We show several examples of complicated scenes drawn with pencil converted to vector images with our approach in Fig. 1.

In summary, we present:

- The first sketch simplification approach that is optimized to directly operate on raster images of *rough sketches*.
- A novel fully Convolutional Neural Network architecture that can simplify sketches directly from images of any resolution.
- An efficient approach to learn the sketch simplification network.
- A dataset for large-scale learning of sketch simplification.

2 Related Work

Various approaches have been proposed to simplify sketch drawings of vector images. One of the strategies for simplification is progressive modification during sketching. In this approach, several drawing tools assist the user in adjusting the shapes of the strokes using: mark-based reparametrization [Baudel 1994], geometric constraints among strokes [Igarashi et al. 1997], cubic Bézier curve fitting [Bae et al. 2008], and progressive merging based on proximity and topology [Grimm and Joshi 2012]. Fišer *et al*. [2015] proposed a system for beautification of freehand sketches based on various rules of geometric relationships between strokes, which works with general Bézier curves. These progressive drawing tools generally depend on the stroke ordering and thus are not easily adapted to non-progressive applications. In contrast, our approach is independent of the stroke order and works on general images.

Other approaches simplify line drawings by removing unnecessary strokes. Preim and Strothotte [1995] enable the user control over the amount of lines based on the length, screen position, and density. Deussen and Strothotte [2000] used depth information to draw simplified foliage of trees. Depth and silhouette information obtained from 3D models is often utilized to evaluate the significance of input strokes [Wilson and Ma 2004; Grabli et al. 2004]. Cole *et al*. [2006] proposed item and priority buffers that determine line visibility and line density respectively. The main problem with these methods is that they are only able to remove existing strokes and are unable to add new ones. This is a severe limitation as usually long strokes consist of a series of short strokes in sketch drawings; the best solution is not necessarily any of the strokes that have been drawn, but a new stroke that would be consistent with the smaller ones. Our approach can both remove and add strokes.

In contrast to the stroke reduction that only removes the less significant strokes, several methods to generate new meaningful strokes by grouping drawn strokes have been proposed. Roshin [1994] grouped strokes based on their three aspects: continuation, parallelism, and proximity. Lindlbauer *et al*. [2013] added appearance similarities (*e.g*., thickness) to the above features to improve the perceptual grouping. Barla *et al*. [2005] proposed a morphological property on simplified strokes that prevents them from folding onto themselves. This method was later improved by exploiting the extent of overlapping [Shesh and Chen 2008]. Pusch *et al*. [2007] presented subdivision-based line simplification that recursively subdivide an input image until each sub-box has a single stroke. The

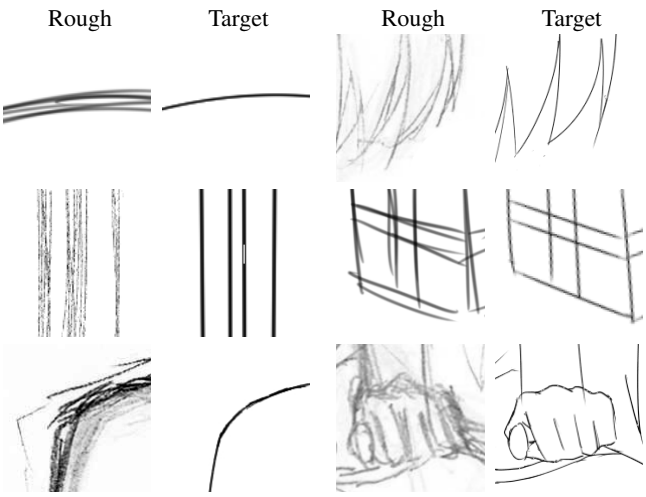


Figure 2: *Examples of the complexity of simplifying rough raster images. We show small examples of rough sketch patches and their corresponding sketch simplifications taken from our dataset. Note how it is common for multiple lines to have to be collapsed into a single line and how the intensity of the different input lines vary greatly even within the same image. Our approach is able to learn how to tackle these extremely challenging using our dataset to then simplify general rough sketches.*

sub-boxes are then connected and B-spline curve fitting is used to generate smooth simple strokes. Orbay and Kara [2011] proposed a sketch beautification method that converts digitally-created sketches into beautified curve segments. They use a supervised stroke clustering algorithm based on geometric relationships between strokes of training sketches. Liu *et al*. [2015] proposed a closure-aware sketch simplification that utilizes closed regions of strokes for semantic analysis of input drawings. However, these stroke reduction approaches still require vector images as input, while our approach can be applied on raster images.

Although the simplification methods of vector images reasonably succeed to generate meaningful simple drawings, the sketch simplification of raster images remains a challenging problem, as neither geometric continuities nor the ordering of vectorized strokes cannot be used. Traditional vectorization approaches are based on line tracing [Freeman 1974], thinning [Zhang and Suen 1984], straight line fitting to anchor points [Janssen and Vossepoel 1997], and cubic Bézier curves fitting [Chang and Yan 1998]. Hilaire and Tombre [2006] proposed a vectorization method that segments line drawings into the most probable graphical primitives such as arcs. These methods use binary images as input and are not suitable for free-hand rough sketches. Bartolo *et al*. [2007] described a simplification and vectorization technique for scribble drawings using Gabor and Kalman filtering. Chen *et al*. [2013] proposed a gradient-based technique for coherence-enhancing filtering, which generates simplified smooth lines via non-oriented gradient fields. However, their method cannot generate detailed structures of sketches such as pencil-and-paper drawings where gradients are subtle and noisy. Noris *et al*. [2013] proposed a vectorization technique for clean drawings, which solves ambiguities near junctions of strokes based on gradient-based pixel clustering and a *reverse drawing* approach that determines the most suitable stroke configurations. However, unlike our method, this method is not applicable for rough sketch simplification as it cannot convert multiple rough lines to a single clean line. Furthermore, none of these approaches have been used on input images as challenging as the ones we present in this work.

While neural networks learnt with back-propagation have been

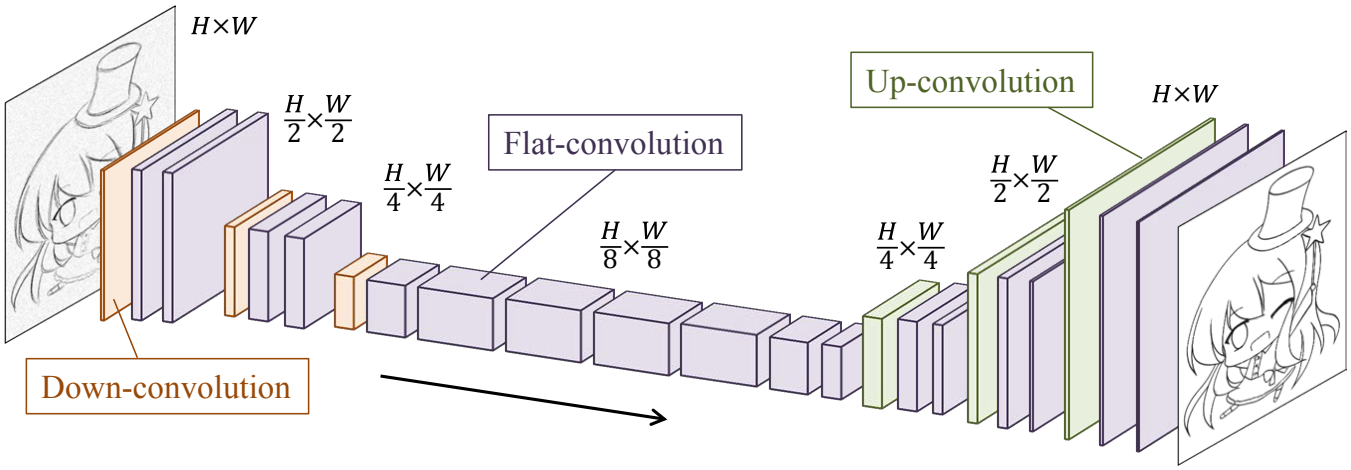


Figure 3: *Overview of our model. Our model is based on convolutional layers of three types: down-convolution, with a stride of* 2 *that halves the image size; flat-convolution, with a stride of* 1 *that maintains the image size; and up-convolution, with a stride of* $1/2$ *that doubles the image size. Initially we decrease the image size with down-convolutions to reduce the data bandwidth and increase the spatial support of subsequent layers, afterwards up-convolutions are used to restore the image to its original size. The depth of each of the convolutional-layer blocks in the figure is proportional to the number of filters it has.*

around for several decades [Rumelhart et al. 1986], only recently has the computational power and data been available to more fully exploit the technique [Krizhevsky et al. 2012]. Originally focusing on classification, in the last few years there have been many different networks proposed for particular tasks. Related to the model we present in this paper are the approaches that output images, such as super-resolution [Dong et al. 2016], semantic segmentations [Long et al. 2015; Noh et al. 2015], contour detection [Shen et al. 2015], and optical flow [Fischer et al. 2015]. Out of these approaches, we can distinguish those that rely on fixed-size image patches [Shen et al. 2015; Dong et al. 2016], and those that rely on up-convolutions [Long et al. 2015; Noh et al. 2015; Fischer et al. 2015]. Our model is inspired by the up-convolutions-based approaches [Zeiler and Fergus 2014; Long et al. 2015; Dosovitskiy et al. 2015], which allow designing networks that downsample to spatially compress the information, and then upsample the data back to the original image size. This also allows training everything in a single end-to-end system unlike the patch-based approaches. In contrast with other methods that use natural images [Long et al. 2015; Noh et al. 2015], we are unable to exploit existing networks as they both require RGB image inputs and are optimized for natural images rather than rough sketches; so we train our network entirely from scratch.

The deep network architecture of Noh *et al*. [2015] is the most similar to our approach: it relies on a fully-convolutional architecture with up-convolutions for semantic segmentation. Yet it still has significant differences due to building off a VGG16 network architecture [Simonyan and Zisserman 2015] and conserving all the pooling layers and the fully-connected layers except the last (treated as convolutions with 1×1 kernels). This results in a network that can only deal with resolutions in 224×224 pixel increments due to using an accumulated 224×224 pixel pooling in their architecture, *i.e*., images between 224×224 and 448×448 pixels without padding will have outputs with 224×224 pixels. In contrast, our architecture uses an accumulated 8×8 pixel pooling (in the form of down-convolutions instead of max-pooling) which allows a much larger range of output image resolutions. By not relying on existing pre-trained networks and designing our architecture from scratch, we are able to completely adapt our network to the rough sketch simplification problem. Furthermore, in order to simplify sketches, we have carefully created a dataset and use a new training procedure which is essential for performance and allows the training of networks from scratch. In particular, without the inverse dataset creation technique we present, it is not possible to train a successful sketch simplification model at all.

In this paper, we overcome the strong limitation of vector input images that existing approaches to sketch simplification have. We are able to handle a variety of practical rough sketches such as scanned pencil-and-paper drawings and detailed sketches with complicated structures as shown in Fig. 2, which cannot be directly vectorized using existing methods. Note how multiple lines are used to represent single lines. Our approach overcomes the difficulty of these images to provide realistic sketch simplifications.

3 Learning to Simplify

We base our model on very deep Convolutional Neural Networks (CNNs) [Krizhevsky et al. 2012; Simonyan and Zisserman 2015] that have a large capacity to learn from data to perform sketch simplification. In order to be able to simplify sketch images, we leverage a large set of recent improvements, *e.g*., batch normalization [Ioffe and Szegedy 2015], ADADELTA [Zeiler 2012], 3×3 convolution kernels [Simonyan and Zisserman 2015], up-convolutions [Long et al. 2015], no explicit pooling [Springenberg et al. 2015], etc., and heavily tailor both the model and the learning approach for the task of sketch simplification. Our contributions include both a novel method for learning and model architecture. An overview of our model can be seen in Fig. 3.

3.1 Convolutional Neural Networks

Convolutional Neural Networks are an extension to Artificial Neural Networks (ANNs) in which the weights are shared across layers [Fukushima 1988; LeCun et al. 1998]. ANNs and their derivatives are a method of approximating a complex unknown function. In our case, this consists of the operation of converting a rough sketch into a simplified drawing. The network consists of several layers of units that can hold real numbers. Each layer can be seen as a multichannel image of the size $h \times w$, where h and w are the height and the width. Let C denote the number of channels, so that the multichannel image is a vector in $\mathbb{R}^{C \cdot h \cdot w}$. The first layer is the input layer, thus its size coincides with the size $(H \times W)$ of the input grayscale image, *i.e*., $h = H, w = W, C = 1$. Similarly, the

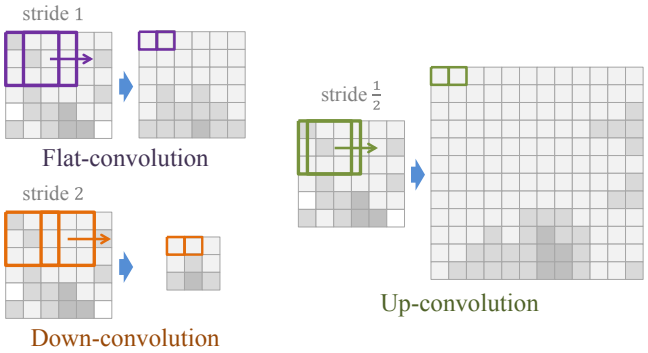


Figure 4: *Upsampling and downsampling using convolutions. We show how using different strides with convolutions allows us to downsample (down-convolution), perform a non-linear mapping (flat-convolution), and upsample the input (up-convolution).*

last layer is the output layer, which also has the same size.

Successive layers are connected by a convolution-with-bias map

$$\text{convadd} : \mathbb{R}^{C \cdot h \cdot w} \longrightarrow \mathbb{R}^{C' \cdot h' \cdot w'} \quad , \tag{1}$$

where (C, h, w) and (C', h', w') are the number of channels, the height, and the width of a layer (L) and the next (L'). For each channel C' of layer L', the map is defined as a convolution with a kernel of the size $C \times k_\text{h} \times k_\text{w}$ followed by the addition of a constant "bias" image. Let $W^{c'}_{c,i,j}$ be the components of the kernel and b'_c the constant bias for channel c' of the layer L', respectively. Then, the value $y_{c',u,v}$ of a specific pixel at (u, v) in channel c' of the layer L' is given by:

$$y_{c',u,v} = b_{c'} + \sum_{i=-k'_\text{h}}^{k'_\text{h}} \sum_{j=-k'_\text{w}}^{k'_\text{w}} \sum_{c=1}^{C} W^{c'}_{c,i+k'_\text{h},j+k'_\text{w}} \; x_{c,u+i,v+j} \;, \tag{2}$$

where $(x_{c,s,t})$ is the multichannel image of layer L, $k'_\text{h} = {}^{(k_\text{h}-1)}/_2$, and $k'_\text{w} = {}^{(k_\text{w}-1)}/_2$.

In the ANN view, this can be seen as synapses connecting the layers where the weights W are independent of the spatial location (u, v) and thus can be seen as shared between synapses related by a parallel translation. Conversely, we can learn the kernel and the bias by back-propagation [Rumelhart et al. 1986] while fixing the shared weights to each other. Thus, the kernel and the bias together give rise to $C \cdot C' \cdot k_\text{h} \cdot k_\text{w} + C'$ learnable weights for each pair of successive layers. Note the number of weights only depends on the kernel size, and the number of channels in the layers.

It is possible to use an increased "stride" to lower the resolution of the output layer. That is, only a subset of positions (u, v) are computed for $y_{c',u,v}$. For example, a stride of 2 would decrease the resolution of the output volume by two as it would only compute $y_{c',u,v}$ for every other pixel. By decreasing the spatial resolution, subsequent convolutions will have an increased spatial support, *i.e.*, the "pixels" in the feature maps will be computed using a larger patch of the original input image. For example, a 3×3 convolution on the original image has a spatial support of 3×3 input pixels for each output pixel. However, if the original image is resized to half the size, the same 3×3 convolution will actually be looking at a 5×5 image patch in the original image. We will construct our model by using increased strides for the first layers to increase the spatial support of subsequent layers. However, increasing the stride decreases the image resolution. In order for the output image to be the same size as the input, we utilize fractional strides to effectively increase the resolution. As an example, using a stride of ${}^1/_2$ will double the resolution of the output layer [Long et al. 2015], as input pixels will be linearly interpolated before being convolved with the convolutional kernel. Our model will use both downscaling and upscaling convolutional layers to increase the spatial resolution with a decreased number of layers, while maintaining an output the same size as the input. An overview of using strides to up- and downsample images is shown in Fig. 4.

After each convolution-with-bias map, a non-linear operation is performed, with the most common one being the Rectified Linear Unit (ReLU) [Nair and Hinton 2010]:

$$\sigma_\text{ReLU}(x) = \max(0, x) \;. \tag{3}$$

Our model also uses the Sigmoid operation for the final layer to have an output in the range $[0, 1]$:

$$\sigma_\text{Sigmoid}(x) = \frac{1}{1 + e^{-x}} \;. \tag{4}$$

The weights of an ANN are learned using back-propagation [Rumelhart et al. 1986] in which given the error of a network, the partial derivative of each weight with respect to the error is computed and used to update the weight by gradient descent. The error of the network is determined by the loss function and the resulting optimization is highly non-convex. Due to the large amount of data used to train these models in combination with a large amount of parameters or weights, stochastic variants of gradient descent are used for optimization, in which each step of the gradient descent algorithm is computed using only a subset of the data known as a batch.

3.2 Model

In contrast with the common CNN models that have fully-connected layers, which do not allow processing images of arbitrary resolution, we focus on exploiting the convolution operation, which allows sharing parameters and processing images of arbitrary resolution. This is inspired by recent approaches [Long et al. 2015; Noh et al. 2015]; however, we opt for designing our architecture from scratch instead of using a pre-trained existing model, as sketch images differ drastically from photographies. We design our model with sketch simplification in mind by having three parts: the first part acts as an encoder and spatially compresses the image, the second part processes and extracts the essential lines from the image, afterwards the third and last part acts as a decoder which converts the small more simple representation to an grayscale image of the same resolution as the input. This is all done using convolutions.

The down- and up-convolution architecture may seem similar to a simple filter banks. However, it is important to realize that the number of channels is much larger where resolution is lower, e.g., 1024 where the size is 1/8. This ensures that information that leads to clean lines is carried through the low-resolution part; the network is trained to choose which information to carry by the encoder-decoder architecture.

For our convolutional layers, we use padding to compensate for the kernel size and ensure the output is the same size as the input when a stride of 1 is used, although the number of channels may change. Instead of using pooling layers, we use convolutional layers with increased stride to lower the resolution from the previous layer. In order for the output of the model to be of the same dimension as the model input, we rely on fractional strides to increase the resolution. Our model is formed by convolutional layers with stride of 1 (*flat-convolution*), 2 (downsampling convolution or *down-convolution*), and ${}^1/_2$ (upsampling convolution or *up-convolution*). An overview of our model can be seen in Fig. 3.

Table 1: *Sketch simplification Convolutional Neural Network architecture. After each convolutional layer, except the last one, there is a rectified linear unit. In the case of the last convolutional layer, there is a Sigmoid layer instead to normalize the output to the* $[0, 1]$ *range. We pad all convolutional layers with zeros such that the output size is the same as the input size when using a stride of* 1*, i.e., 2 pixel padding for* 5×5 *kernels and 1 pixel padding for* 3×3 *kernels. All output sizes reference the original image width* W *and height* H*, as the model can process images of any resolution.*

type	kernel size	stride	output size
input	-	-	$1 \times H \times W$
down-convolution	5×5	2×2	$48 \times {}^{H}/_{2} \times {}^{W}/_{2}$
flat-convolution	3×3	1×1	$128 \times {}^{H}/_{2} \times {}^{W}/_{2}$
flat-convolution	3×3	1×1	$128 \times {}^{H}/_{2} \times {}^{W}/_{2}$
down-convolution	3×3	2×2	$256 \times {}^{H}/_{4} \times {}^{W}/_{4}$
flat-convolution	3×3	1×1	$256 \times {}^{H}/_{4} \times {}^{W}/_{4}$
flat-convolution	3×3	1×1	$256 \times {}^{H}/_{4} \times {}^{W}/_{4}$
down-convolution	3×3	2×2	$256 \times {}^{H}/_{8} \times {}^{W}/_{8}$
flat-convolution	3×3	1×1	$512 \times {}^{H}/_{8} \times {}^{W}/_{8}$
flat-convolution	3×3	1×1	$1024 \times {}^{H}/_{8} \times {}^{W}/_{8}$
flat-convolution	3×3	1×1	$1024 \times {}^{H}/_{8} \times {}^{W}/_{8}$
flat-convolution	3×3	1×1	$1024 \times {}^{H}/_{8} \times {}^{W}/_{8}$
flat-convolution	3×3	1×1	$1024 \times {}^{H}/_{8} \times {}^{W}/_{8}$
flat-convolution	3×3	1×1	$512 \times {}^{H}/_{8} \times {}^{W}/_{8}$
flat-convolution	3×3	1×1	$256 \times {}^{H}/_{8} \times {}^{W}/_{8}$
up-convolution	4×4	${}^{1}/_{2} \times {}^{1}/_{2}$	$256 \times {}^{H}/_{4} \times {}^{W}/_{4}$
flat-convolution	3×3	1×1	$256 \times {}^{H}/_{4} \times {}^{W}/_{4}$
flat-convolution	3×3	1×1	$128 \times {}^{H}/_{4} \times {}^{W}/_{4}$
up-convolution	4×4	${}^{1}/_{2} \times {}^{1}/_{2}$	$128 \times {}^{H}/_{2} \times {}^{W}/_{2}$
flat-convolution	3×3	1×1	$128 \times {}^{H}/_{2} \times {}^{W}/_{2}$
flat-convolution	3×3	1×1	$48 \times {}^{H}/_{2} \times {}^{W}/_{2}$
up-convolution	4×4	${}^{1}/_{2} \times {}^{1}/_{2}$	$48 \times H \times W$
flat-convolution	3×3	1×1	$24 \times H \times W$
flat-convolution	3×3	1×1	$1 \times H \times W$

The basic building block of our model is a convolutional layer (Eq. (2)) followed by a rectified linear unit layer (Eq. (3)). The last layer is special in that it is followed by a Sigmoid unit layer (Eq. (4)) in order to output a grayscale image. We downsample the model three times using convolutional layers with a stride of 2 (*down-convolution*) and upsample three times using convolutional layers with a stride of ${}^{1}/_{2}$ (*up-convolution*). This fully-convolutional approach allows our model to work with any resolution and aspect ratio in contrast to the standard CNN models with fully-connected layers that require fixed input sizes.

We reduce the number of parameters in the full model by relying primarily on 3×3 convolution kernels except for the first convolutional layer, which uses a 5×5 kernel, and the upsampling layers, which use 4×4 kernels. The reasoning behind this is that a 5×5 convolution can be approximated by two consecutive 3×3 convolutions with only ${}^{18}/_{25} = 72\%$ the amount of parameters. Furthermore, using two 3×3 convolutions allow better approximation of non-linearities [Simonyan and Zisserman 2015]. However, when upsampling, a 4×4 kernel is used instead of a 3×3 kernel so that the output size is exactly twice the input size. The full details of our architecture can be seen in Table 1.

3.3 Model Loss

We train the model using training pairs of rough and simplified sketches as input and target, respectively. As a loss, we use the weighted mean square error criterion

$$l(Y, Y^*, M) = \| M \odot (Y - Y^*) \|^2_{\text{FRO}} \ , \tag{5}$$

where Y is the model output, Y^* is the target output, M is the loss map, $\odot$ is the matrix element-wise multiplication or Haddamard product, and $\| \cdot \|_{\text{FRO}}$ is the Frobenius norm. Note that a perfect model ($Y = Y^*$) will have a loss of 0 regardless of the loss map M chosen.

We experimentally tested various loss maps and found that, while they do not change the final performance substantially, the one we describe here can speed-up the learning. We chose a loss map that reduces the loss on the thicker lines, in order to avoid having the model focus on the thicker lines and forego the thinner lines. We construct our loss maps by looking at histograms around each pixel in the ground truth (target) label. The loss map is defined as:

$$M(u, v) = \begin{cases} 1 & \text{if } I(u, v) = 1 \\ \min\left(\alpha \exp\left(H(I, u, v)\right) + \beta, 1\right) & \text{else} \end{cases} \tag{6}$$

where $H(I, u, v)$ is the value of the bin of the local normalized histogram in which the pixel $I(u, v)$ falls into. The histogram is constructed using all pixels within d_h pixels from the center using b_h bins.

An example of the optimization with and without the loss map can be seen in Fig. 5. We can see how the loss map attenuates the eyebrows of the figure in the sketch to allow the model to learn all parts of the image in a more equal fashion. Notice how, after

Figure 5: *Visualization of the optimization process with and without the loss map. The main purpose of the loss map is to decrease the importance of thick lines when training to speed-up the learning process. If we train the model using the single image without the loss map, the optimization focuses on the thicker lines while ignoring the other thinner lines, as we can see in the top row. However, when we add the loss map, a balance is struck between the thick and the thin lines as shown in the bottom row. We can see how the eyebrows that have very thick lines are detected and modulated to have a weaker loss. We use values of* $\alpha = 6$*,* $\beta = 3$*,* $d_h = 2$*, and* $b_h = 10$ *for computing the loss map.*

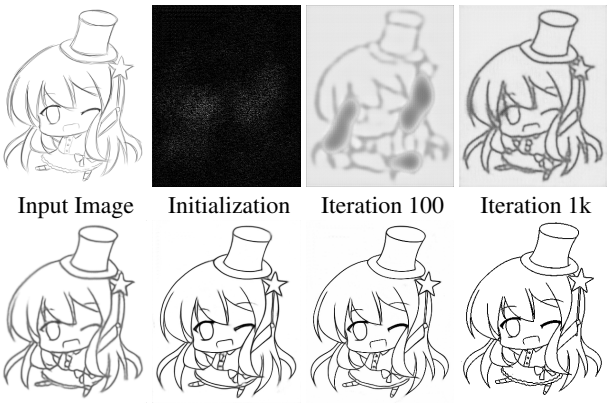


Figure 6: *Visualization of the training of the model. It is initialized randomly and as training proceeds we can see how the figure becomes clear and polished. The model initially focuses on joining the lines into a single blurry line (Iteration 1000) and then progressively learns to refine the simplified line until it converges.*

200 iterations, the output lines in the model optimized with the loss map can be clearly seen, in comparison with the case when it is not used, where the output is still a set of blurry blobs.

3.4 Learning

One of the main recent innovations that have allowed the training of such deep models as the one we present from scratch are batch normalization layers [Ioffe and Szegedy 2015]. They consist of simply keeping a running mean and standard deviation of the input data, and using them to normalize the input data. The output of these layers roughly has a mean of 0 and a standard deviation of 1. The running mean is only updated during training and is kept fixed during evaluation. Batch normalization layers also have two additional learnable parameters that serve to scale the normalized output and add a bias:

$$y_{BN}(x) = \frac{x - \mu}{\sqrt{s^2 + \epsilon}}\gamma + \eta \,, \tag{7}$$

where μ is the running mean, s is the running standard deviation, ϵ is a constant for numerical stability, and γ and η are learnable parameters. We use these layers after all convolutional layers except for the last one during training. Once the model is trained, these additional layers can be folded into the previous convolutional layer to not add any overhead during inference. This is done by simply reducing Eq. (7) to a linear transformation (note that everything except x is constant during evaluation), and merging this linear transformation with the linear transformation of the preceding convolutional layer. That is, the weights of the convolutional layer get multiplied by $\gamma/\sqrt{s^2+\epsilon}$ and $\gamma\mu/\sqrt{s^2+\epsilon} - \eta$ is subtracted from the bias. Without these temporary layers, learning is not possible in a reasonable amount of time.

For learning the weights of the models, we rely on the ADADELTA algorithm [Zeiler 2012]. The main advantage of this approach is that it does not require explicitly setting a learning rate, which is a non-trivial task. ADADELTA has been shown to generally converge to similar solutions as other algorithms, however, it will take a longer time to converge in comparison with optimally tuned learning rates. For training a sketch model, we tried various other optimizers and found that the result did not change significantly, while other optimizers have the added complexity of choosing a learning rate scheduler. ADADELTA consists of keeping a running mean of the square of the gradients and the square of the updates, which are used to determine the learning rate. An update of the parameters of the model θ then becomes,

$$\theta_{t+1} = \theta_t + \Delta\theta_t = \theta_t - \frac{\text{RMS}[\Delta\theta]_{t-1}}{\text{RMS}[\delta\theta]_t}\delta\theta_t \,, \tag{8}$$

where $\Delta\theta_t$ is the parameter update, and $\delta\theta_t$ is the gradient of the parameters given the loss for a given iteration t. The update is done by computing the Root Mean Square (RMS) of the running averages. Note that this approach automatically adjusts the learning rate independently for all the different weights of the model.

We perform extensive data augmentation in order to combat overfitting of the model and improve the generalization. We train with constant size 424×424 image patches extracted randomly from the full image. We first extend the dataset by downscaling it by $^7/_6$, $^8/_6$, $^9/_6$, $^{10}/_6$, $^{11}/_6$, $^{12}/_6$, $^{13}/_6$, and $^{14}/_6$. Note that we do not use the downscaled images if they are smaller than the training image patch size. This results in roughly nine times the original amount of image pairs, although they are heavily correlated. We then threshold the simplified sketch images so that all pixels, which are in the $[0, 1]$ range, with a value below 0.9 are set to 0. This normalization is critical for learning as all output images will have similar tones. The resulting images are randomly rotated in the range of $[-180, 180]$ degrees and also randomly flipped horizontally. When training, we sample larger images more frequently based on the number of pixels in comparison with smaller images to compensate that smaller images contribute more to the learning when extracting patches. Thus, patches from a 1024×1024 image will be four times more likely to appear than patches from a 512×512 image. Furthermore, with a probability of 10%, we change the input image to be the same as the target image, *i.e.*, we try to teach the model that clean images should not be modified. Training is done until the convergence of the loss.

Figure 7: *Effect of vectorization on the output of our model. We vectorize our model using automatic publically available tools with default parameters. As the output of our model is a clean simplified image, such a simple approach yields excellent results. The vectorization process consists of a high-pass filter followed by a binary thresholding. Afterwards polygons are fitted to the binary image and converted to Bézier curves. An example result of vectorization is shown on the right.*

3.5 Vectorization

We employ simple techniques to vectorize our model output as it is already a clean simplified line drawing in order for the result to be directly usable by graphical artists. We automate the approach by performing a simple high-pass filter and thresholding using the publicly available Potrace software [Selinger 2003] with default parameters. We show the result of vectorizing the output of our model in Fig. 7. Note that this vectorization is, like our model, fully automated and requires no user intervention.

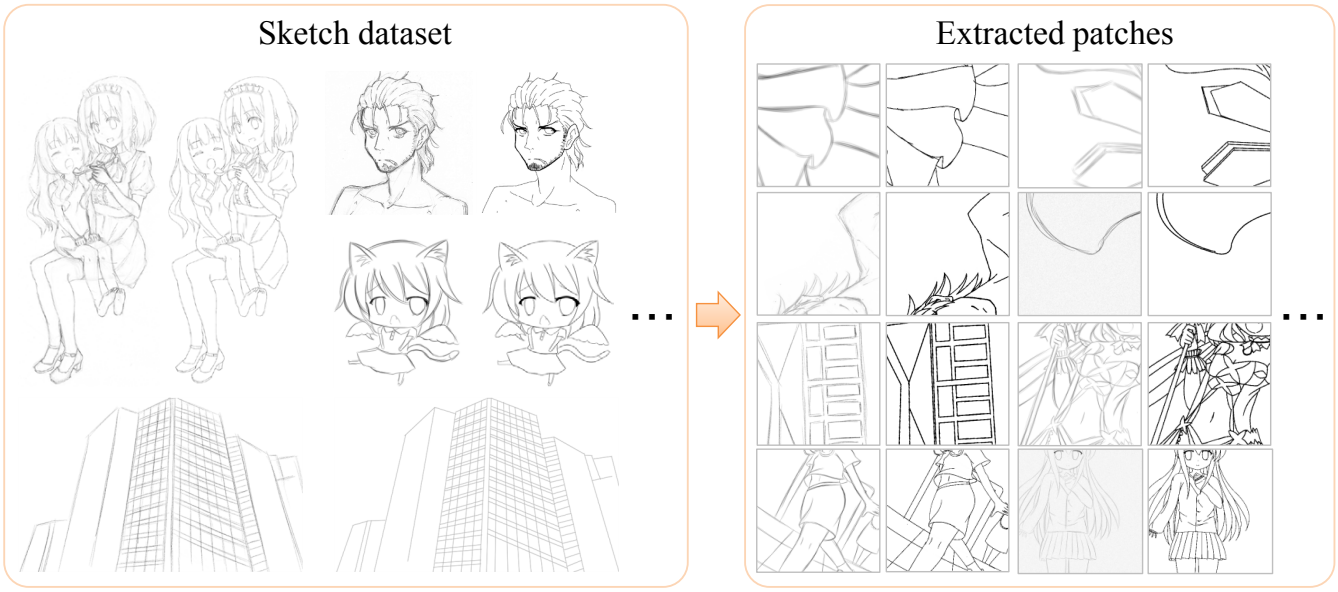


Figure 8: *Examples from our sketch simplification dataset used for training our model. The left of each pair shows the rough sketch while the right shows the corresponding simplified sketch. We use the rough sketches as the input of our model and the simplified sketches as the target output when training our model. Note that these patches are randomly extracted during training and not fixed.*

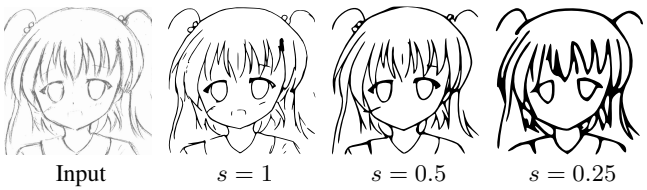


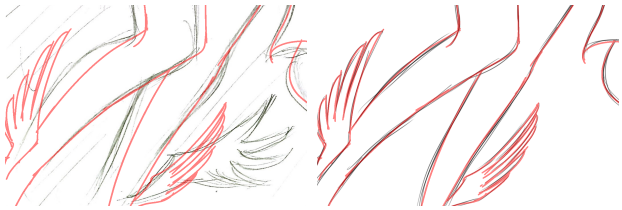
Figure 9: *Sketch simplification on scaled versions of the input image. The scaling done with respect to the input image on the left is denoted with s, where $s = 1$ denotes no scaling. By down-scaling the input image, it is possible to obtain more simplified sketches.*

3.6 Controlling Simplicity by Scaling

While our approach is fully automatic and requires no user-intervention, it is possible to tweak the results in various ways. The most straight-forward way is to scale the input image. Down-scaling the input images will result in more simple output images. On the other hand, up-scaling the images will result in more conservation of fine details. By changing the amount of scaling, it is possible for the user to control the degree of simplification of the algorithm. An example of the effect of scaling can be seen in Fig. 9.

4 Rough Sketch Dataset

To teach our model to simplify sketches, we build a dataset using the *inverse dataset construction* which consists of creating rough sketches from clean sketches and results critical to be able to train a successful sketch simplification model. Our dataset is formed by 68 pairs of training images drawn by 5 different artists. These images consist of pairs of rough and simplified images and have different resolutions with an average of 1280.0×1662.7 pixels. The smallest image is 630×630 pixels and the largest is 2416×3219 pixels. Some examples from the dataset and patches used for learning our model can be seen in Fig. 8.

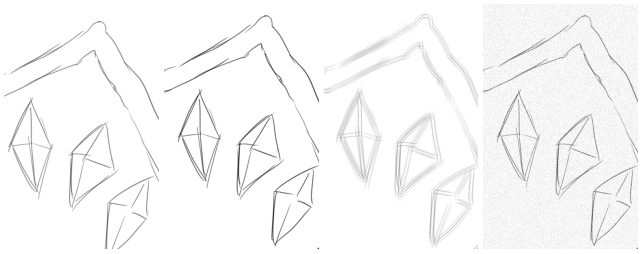


(a) Direct (b) Inverse

Figure 10: *Comparison of direct and inverse dataset construction approaches. For both cases, the input image is shown in original grayscale, while the target image is shown overlaid in red. If we attempt to create rough images and their simplifications in a direct way, we get results such as the one shown on the left. As we can clearly see, even when aligned, the artist has taken various liberties to change different parts of the original sketch. If we attempt to use this for training, our model will not be able to learn this mapping. However, if we ask the artist to once again create a rough sketch based on the clean sketch obtained in the direct approach, we get the result shown on the right. Notice how the input and target images are very well aligned. We call this dataset construction approach the inverse dataset construction approach, as we are generating input images from target images. Data created with this approach is suitable for training deep neural networks.*

4.1 Inverse Dataset Construction

Dataset quality and quantity is critical for the performance of Deep Convolutional Neural Networks such as the one proposed in this work. We found that the standard approach, which we denote as *direct dataset construction*, of asking artists to draw a rough sketch and then produce a clean version of the sketch ended up with a lot of changes in the figure, *i.e.*, output lines are greatly changed with respect to their input lines, or new lines are added in the output. This results in very noisy training data that does not perform well. In order to avoid this issue, we found that the best approach is the *inverse dataset construction* approach, that is, given a clean simplified sketch drawing, the artist is asked to make a rough version of that sketch. While this does result in additional overhead



(a) input (b) tone (c) slur (b) noise

Figure 11: *We further augment the dataset by changing the tone of the input image (b), slurring the input image (c), and adding random noise (d). By using these additional training images, we can make our model generalize better to other images.*

to the artist, the quality of the dataset is significantly superior and training using this data results in much better models. An example of the difference between the traditional direct and inverse dataset construction approach can be seen in Fig. 10. We use this approach in the creation of the 68 pairs of training images.

4.2 Data Augmentation

Due to the relatively low number of training images and the high diversity of rough sketches found in the wild, we further augment the dataset to have four times the images. We employ Adobe Photoshop to perform this data augmentation and perform three augmentations: tone change, image slur, and noise. For an example of these augmentations, refer to Fig. 11.

Tone change is done by using the *Auto Tone* tool with default parameters. This automatically sets the exposure, contrast, highlights, shadows, whites and blacks of the image as shown in Fig. 11-(b). It requires no parameters and is done in a fully automated approach.

Image slur is done by using the *Fragment* tool with default parameters. This blurs the image, duplicates it, and puts the duplicates together with an offset. The resulting image has a more rough, although somewhat blurry, appearance in comparison with the raw input. This can be seen in Fig. 11-(c).

Noise is done by the *Noise-Uniform* tool with default parameters. This adds noise to all the pixels in the image using a uniform distribution. An example can be seen in Fig. 11-(d). This gives a result that is similar to that of low-quality digital scans of paper-and-pencil drawings and helps our model be robust to those.

By manually augmenting all the training images, we obtain a training dataset with four times more images than the original, which improves the quality of the results.

5 Experimental Results and Discussion

We have performed extensive analysis of our model and showed that it is robust and suitable for all types of rough sketches. For all images, we first subtracted the mean gray value of the training dataset. For computing the loss map, we used the values of $\alpha = 6$, $\beta = 3$, $d_h = 2$, and $b_h = 10$ as depicted in Fig. 5. We trained our model for 600,000 iterations with a batch size of 6. This takes roughly three weeks using a Nvidia TITAN X GPU. We use the same model for all the experiments in the rest of the section.

5.1 From Pencil and Paper to Vector Images

While digital sketching has taken force with the appearance of many high-quality digital tablets, many artists still prefer to initially draft the sketch with pencil-and-paper. Afterwards, the sketch is scanned and vectorized manually using a digital tablet. Our approach intends to replace this manual step and allow the sketch to be directly imported as a vector image which the artist can then modify and colorize. In order to test our model, we directly input rough sketches drawn with pencil and visualized the vector image results. We evaluated on various rough sketches obtained from different artists in Fig. 12. Note that we did not perform any sort of preprocessing on the input images. We directly input them into our model and performed simple vectorization on the result to obtain vector images as output.

We can see that, despite the complexity and differences between the various images, our model in general is able to perform accurate and meaningful line simplifications. We note that other existing sketch simplification approaches require vector inputs and vectorization approaches are unable to handle such complicated rough sketches as the ones we consider in this work.

Table 2: *Results of our user study comparing our approach with commercial vectorization software. We processed 15 images with our model, Potrace, and Adobe Live Trace. For Potrace and Adobe Live Trace, we manually set the threshold for each image to obtain the best results, while our approach is fully automatic. We show the absolute score on the scale of 1 to 5 for each model and relative comparisons, i.e., which model is better in the last three rows. We can see that our approach significantly outperforms the vectorization approaches.*

	Ours	Live Trace	Potrace
Score	**4.53**	2.94	2.80
vs Ours	-	2.5%	2.8%
vs Live Trace	**97.5%**	-	30.3%
vs Potrace	**97.2%**	69.7%	-

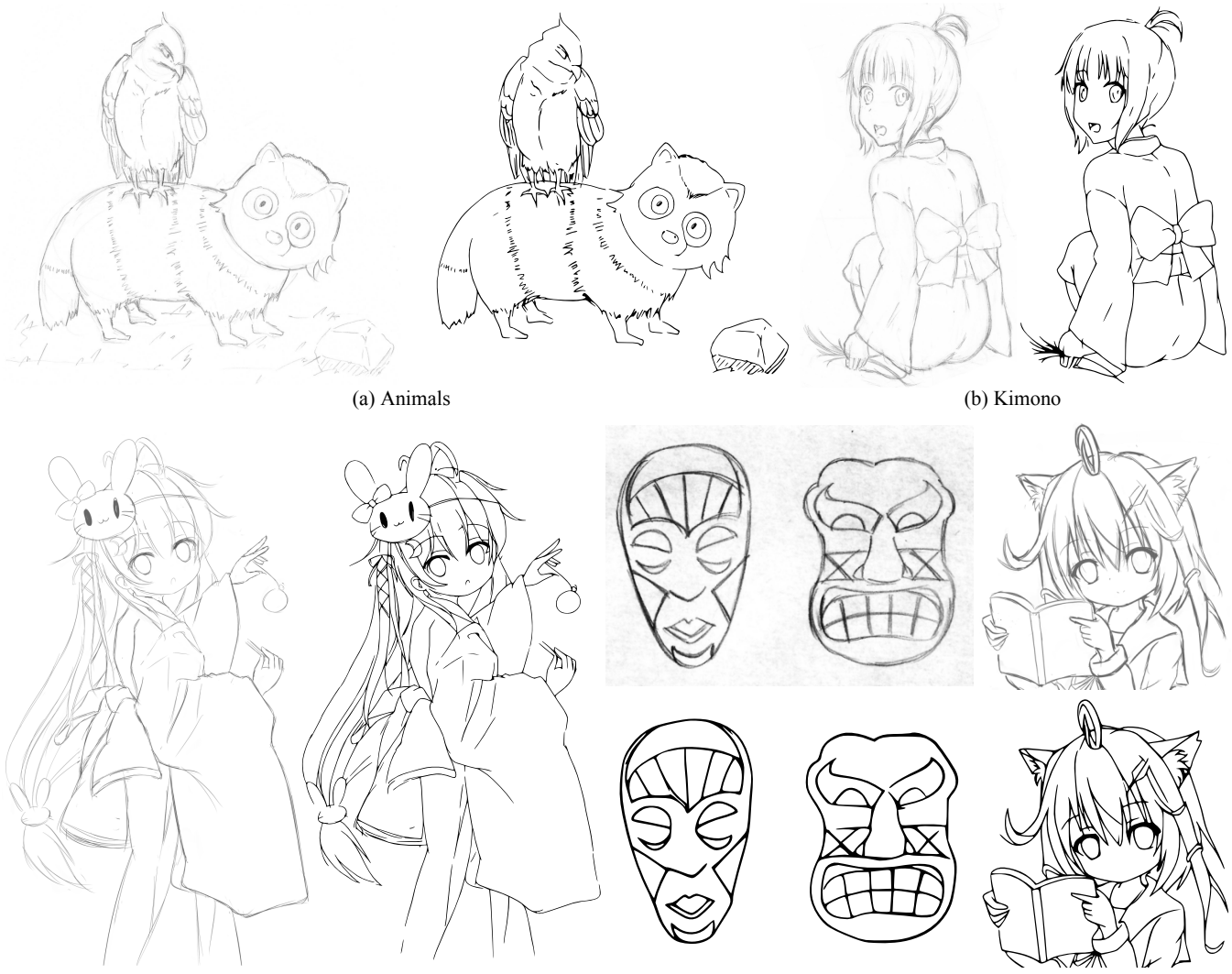
5.2 Comparison with the State of the Art

We compare against the state of the art [Liu et al. 2015] in sketch simplifications on several images. However, note that the state of the art requires vector images as input, while our approach does not. For the purpose of evaluation, we fed a vector image to [Liu et al. 2015] and its rasterized version to our model. The comparison can be seen in Fig. 13. We can see that, in general, our performance is on par despite not being limited to vector image inputs. We further note that, as these images were rasterized from the original vector images and thus are fairly different from the images on which we train our model, *i.e.*, they already have much cleaner dark lines in comparison to the dirty real sketches from Fig. 12.

We also performed comparisons against commercial vectorization software which can process raster input images. In particular, we compared with the Potrace [Selinger 2003] and Adobe Live Trace. We used the default parameters except we manually set the threshold of both approaches to 0.9 in order to obtain the best results. We show the results in Fig. 14. We can see that, due to the complexity of the images we evaluate on, vectorization approaches failed to give good results. This is especially visible on parts of the sketch that have overlapping multiple lines. Our approach was able to elegantly fuse these lines into a single clean line, while performing vectorization directly either conserves multiple lines or, in the case they are faint, fails to conserve them at all.

5.3 User Study

We also performed a user study to evaluate our model. We compared our model against Potrace and Adobe Live Trace. We se-



(a) Animals (b) Kimono

(c) Matsuri (d) Masks (e) Book

Figure 12: *Results of our approach on different pencil-and-paper images. Note the variety and the coarseness of the different sketches. Despite the complexity, our approach is able to obtain reliable line simplifications. We note in particular how clean the result of (d) is despite the very challenging dirty input image. All sketches come from three different artists.*

lected 15 images for evaluation and processed them with all three approaches. In the case of Potrace and Adobe Live Trace, we manually set the threshold for each image to obtain best results. Comparison was done in two formats: (a) comparing two processed images to see which is better, and (b) ranking a processed image on a scale of 1 to 5.

We used 19 users for both cases, 10 of whom had significant experience in sketch drawing. Results are shown in Table 2. We can see that, when compared to the other approaches, our model was considered better in over 97% of the cases. Furthermore, in absolute terms, our approach was ranked 4.53 on a scale of 1 to 5. We found no significant differences between the naïve and expert users.

5.4 Computation Time

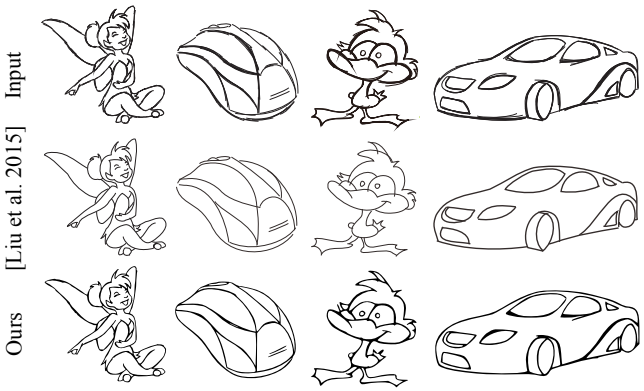
Evaluation time depends heavily on the resolution of the input image. Our model can be run both on the GPU and CPU, although best performance is obtained on the GPU, allowing for near real-time performance. In comparison, [Liu et al. 2015] take various minutes depending on the number of strokes. We test for various square images of different sizes initialized randomly and show the mean results for 100 evaluations in Fig. 3. For evaluation, we use an Intel Core i7-5960X CPU at 3.00 GHz with 8 cores and NVIDIA GeForce TITAN X GPU. We note that using a GPU gives nearly a $50\times$ speedup. As we can see, our approach is suitable for real world usage.

Table 3: *Analysis of computation time for our model. We notice a significant speedup when using the GPU that drives computation times to under a second even for large input images.*

Image Size	Pixels	CPU (s)	GPU (s)	Speedup
320×320	102,400	2.014	0.047	$42.9\times$
640×640	409,600	7.533	0.159	$47.4\times$
1024×1024	1,048,576	19.463	0.397	$49.0\times$

5.5 Limitations

The main limitation of our approach is that it has a strong dependency on the quality and quantity of the training data. However, we



(a) Fairy (b) Mouse (c) Duck (d) Car

Figure 13: *Comparison with the state of the art. Note that, while [Liu et al. 2015] uses fairly clean vector images as input, our model directly uses raster images.*

show that with a small dataset we are still able to generalize fairly well to many different images. Given additional training data, it is likely that we would be able to obtain better performance and generalization. Additionally, while the inference of the proposed model is very fast, the learning process is computationally very expensive and relies on high-end GPUs in order to finish in a reasonable amount of time.

6 Conclusions

We have presented a novel automated end-to-end system that takes rough raster sketches and outputs high quality vectorized simplifications. Our model is based on stacked convolution operations for efficiency, and is able to handle very challenging pencil-and-paper scanned images from various sources. Furthermore, our proposed fully-convolutional architecture is optimized for the simplification task and can process images of any resolution. We also present a novel dataset carefully designed for the task that, in combination with our learning method, can be used to teach our model to simplify sketches. Our approach is fully automatic and requires no user intervention. Our results show that our approach is able to outperform the state of the art in sketch simplification despite not sharing the severe limitations of only being able to process vector images while maintaining a computation time of under a second. We also corroborate with a user study that processing images with our model gives significantly better results in comparison with commercial vectorization software. We believe our proposed approach is an important step towards being able to integrate sketch simplification into artist's everyday work flow.

7 Acknowledgements

This work was partially supported by JST CREST.

References

BAE, S.-H., BALAKRISHNAN, R., AND SINGH, K. 2008. Ilovesketch: As-natural-as-possible sketching system for creating 3d curve models. In *ACM Symposium on User Interface Software and Technology*, 151–160.

BARLA, P., THOLLOT, J., AND SILLION, F. X. 2005. Geometric clustering for line drawing simplification. In *ACM SIGGRAPH 2005 Sketches*.

BARTOLO, A., CAMILLERI, K. P., FABRI, S. G., BORG, J. C., AND FARRUGIA, P. J. 2007. Scribbles to vectors: Preparation of scribble drawings for cad interpretation. In *Eurographics Workshop on Sketch-based Interfaces and Modeling*, 123–130.

BAUDEL, T. 1994. A mark-based interaction paradigm for free-hand drawing. In *ACM Symposium on User Interface Software and Technology*, 185–192.

CHANG, H.-H., AND YAN, H. 1998. Vectorization of hand-drawn image using piecewise cubic bézier curves fitting. *Pattern Recognition 31*, 11, 1747 – 1755.

CHEN, J., GUENNEBAUD, G., BARLA, P., AND GRANIER, X. 2013. Non-oriented mls gradient fields. *Computer Graphics Forum 32*, 8, 98–109.

COLE, F., DECARLO, D., FINKELSTEIN, A., KIN, K., MORLEY, K., AND SANTELLA, A. 2006. Directing gaze in 3d models with stylized focus. In *Eurographics Conference on Rendering Techniques*, 377–387.

DEUSSEN, O., AND STROTHOTTE, T. 2000. Computer-generated pen-and-ink illustration of trees. In *Conference on Computer Graphics and Interactive Techniques*, 13–18.

DONG, C., LOY, C. C., HE, K., AND TANG, X. 2016. Image super-resolution using deep convolutional networks. *PAMI 38*, 2, 295–307.

DOSOVITSKIY, A., SPRINGENBERG, J. T., AND BROX, T. 2015. Learning to generate chairs with convolutional neural networks. In *CVPR*.

FISCHER, P., DOSOVITSKIY, A., ILG, E., HÄUSSER, P., HAZIRBAS, C., GOLKOV, V., VAN DER SMAGT, P., CREMERS, D., AND BROX, T. 2015. Flownet: Learning optical flow with convolutional networks.

FIŠER, J., ASENTE, P., AND SÝKORA, D. 2015. Shipshape: A drawing beautification assistant. In *Workshop on Sketch-Based Interfaces and Modeling*, 49–57.

FREEMAN, H. 1974. Computer processing of line-drawing images. *ACM Comput. Surv. 6*, 1, 57–97.

FUKUSHIMA, K. 1988. Neocognitron: A hierarchical neural network capable of visual pattern recognition. *Neural networks 1*, 2, 119–130.

GRABLI, S., DURAND, F., AND SILLION, F. 2004. Density measure for line-drawing simplification. In *Pacific Conference on Computer Graphics and Applications, 2004*, 309–318.

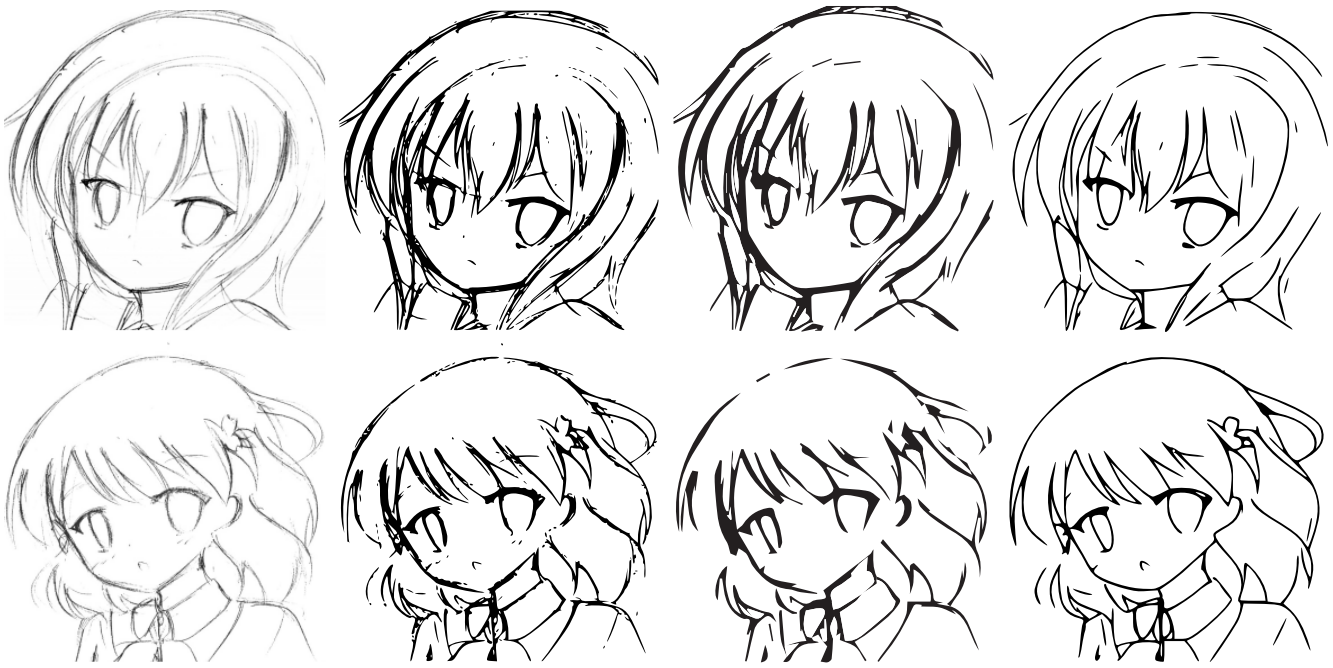
GRIMM, C., AND JOSHI, P. 2012. Just drawit: A 3d sketching system. In *nternational Symposium on Sketch-Based Interfaces and Modeling*, 121–130.

HILAIRE, X., AND TOMBRE, K. 2006. Robust and accurate vectorization of line drawings. *PAMI 28*, 6, 890–904.

IGARASHI, T., MATSUOKA, S., KAWACHIYA, S., AND TANAKA, H. 1997. Interactive beautification: A technique for rapid geometric design. In *ACM Symposium on User Interface Software and Technology*, 105–114.

IOFFE, S., AND SZEGEDY, C. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*.

JANSSEN, R. D., AND VOSSEPOEL, A. M. 1997. Adaptive vectorization of line drawing images. *Computer Vision and Image Understanding 65*, 1, 38 – 56.



(a) Input (b) Potrace (c) Adobe Live Trace (d) Ours

Figure 14: *Comparison with commercial tools for vectorization. We used the default parameters when possible for all tools. However, as Adobe Live Trace requires manually setting a threshold parameter, we set it to the best visual result value of 0.9. We also set the potrace threshold to 0.9 otherwise most of the input image gets erased. In general, directly vectorizing the image gives poor results, not fully simplifying the lines or erasing vital parts of the image. In contrast, our approach gave the most accurate simplification of the input image.*

KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. 2012. Imagenet classification with deep convolutional neural networks. In *NIPS*.

LECUN, Y., BOTTOU, L., BENGIO, Y., AND HAFFNER, P. 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE 86*, 11, 2278–2324.

LINDLBAUER, D., HALLER, M., HANCOCK, M. S., SCOTT, S. D., AND STUERZLINGER, W. 2013. Perceptual grouping: selection assistance for digital sketching. In *International Conference on Interactive Tabletops and Surfaces*, 51–60.

LIU, X., WONG, T.-T., AND HENG, P.-A. 2015. Closure-aware sketch simplification. *ACM Trans. Graph. 34*, 6, 168:1–168:10.

LONG, J., SHELHAMER, E., AND DARRELL, T. 2015. Fully convolutional networks for semantic segmentation. In *CVPR*.

NAIR, V., AND HINTON, G. E. 2010. Rectified linear units improve restricted boltzmann machines. In *ICML*, 807–814.

NOH, H., HONG, S., AND HAN, B. 2015. Learning deconvolution network for semantic segmentation. In *ICCV*.

NORIS, G., HORNUNG, A., SUMNER, R. W., SIMMONS, M., AND GROSS, M. 2013. Topology-driven vectorization of clean line drawings. *ACM Trans. Graph. 32*, 1, 4:1–4:11.

ORBAY, G., AND KARA, L. 2011. Beautification of design sketches using trainable stroke clustering and curve fitting. *IEEE Trans. on Visualization and Computer Graphics 17*, 5, 694–708.

PREIM, B., AND STROTHOTTE, T. 1995. Tuning rendered line-drawings. In *Winter School in Computer Graphics*, 227–237.

PUSCH, R., SAMAVATI, F., NASRI, A., AND WYVILL, B. 2007. Improving the sketch-based interface. *The Visual Computer 23*, 9-11, 955–962.

ROSIN, P. L. 1994. Grouping curved lines. In *Machine Graphics and Vision 7*, 625–644.

RUMELHART, D., HINTON, G., AND WILLIAMS, R. 1986. Learning representations by back-propagating errors. In *Nature*.

SELINGER, P. 2003. Potrace: a polygon-based tracing algorithm. *Potrace (online), http://potrace. sourceforge. net/potrace. pdf (2009-07-01)*.

SHEN, W., WANG, X., WANG, Y., BAI, X., AND ZHANG, Z. 2015. Deepcontour: A deep convolutional feature learned by positive-sharing loss for contour detection. In *CVPR*.

SHESH, A., AND CHEN, B. 2008. Efficient and dynamic simplification of line drawings. *Computer Graphics Forum 27*, 2, 537–545.

SIMONYAN, K., AND ZISSERMAN, A. 2015. Very deep convolutional networks for large-scale image recognition. In *ICLR*.

SPRINGENBERG, J. T., DOSOVITSKIY, A., BROX, T., AND RIEDMILLER, M. A. 2015. Striving for simplicity: The all convolutional net. In *ICLR Workshop Track*.

WILSON, B., AND MA, K.-L. 2004. Rendering complexity in computer-generated pen-and-ink illustrations. In *International Symposium on Non-photorealistic Animation and Rendering*, 129–137.

ZEILER, M. D., AND FERGUS, R. 2014. Visualizing and understanding convolutional networks. In *ECCV*.

ZEILER, M. D. 2012. ADADELTA: an adaptive learning rate method. *CoRR abs/1212.5701*.

ZHANG, T. Y., AND SUEN, C. Y. 1984. A fast parallel algorithm for thinning digital patterns. *Commun. ACM 27*, 3, 236–239.