Controlling Stroke Size in Fast Style Transfer with Recurrent Convolutional Neural Network

Lingchen Yang\(^1\) \quad Lumin Yang\(^1\) \quad Mingbo Zhao\(^2\) \quad Youyi Zheng\(^1\)\(^\dagger\)

\(^1\) State Key Lab of CAD & CG, Zhejiang University
\(^2\) College of Information Science and Technology, Donghua University

Abstract

Controlling stroke size in Fast Style Transfer remains a difficult task. So far, only a few attempts have been made towards it, and they still exhibit several deficiencies regarding efficiency, flexibility, and diversity. In this paper, we aim to tackle these problems and propose a recurrent convolutional neural subnetwork, which we call recurrent stroke-pyramid, to control the stroke size in Fast Style Transfer. Compared to the state-of-the-art methods, our method not only achieves competitive results with much fewer parameters but provides more flexibility and efficiency for generalizing to unseen larger stroke size and being able to produce a wide range of stroke sizes with only one residual unit. We further embed the recurrent stroke-pyramid into the Multi-Styles and the Arbitrary-Style models, achieving both style and stroke-size control in an entirely feed-forward manner with two novel run-time control strategies.

CCS Concepts

- Computing methodologies → Neural networks; Image processing;

1. Introduction

Rendering an image in the style of another one is a long-standing problem [GG01, SS02, KEBK05]. Inspired by the deep learning algorithms, the seminal work of Gatys et al. [GEB15b, GEB15a] shows that a pre-trained convolutional neural network (CNN) can be used as a feature extractor to achieve this goal. The key idea is to optimize an image with a gradient descend method, so as to match the feature statistics of it to that of the style for texture synthesis, or both the style and the content for image stylization. However, this algorithm is extremely slow in terms of real-time application, as it needs hundreds of iterations to converge every time.

To resolve the problem, significant efforts have been devoted to speeding up neural style transfer. Ulyanov et al. [ULVL16] and Johnson et al. [JAFF16] attempt to train a feed-forward neural network that stylizes the input image in a single forward pass, which turns out to be three orders of magnitude faster than the method of [GEB15b, GEB15a]. Nonetheless, the flexibility is lost: a style transfer network is tied to one specific style. Surprisingly, Ulyanov et al. [DSK16] discover that specializing affine transformation parameters to each specific style is sufficient to model a style. They successfully incorporate different styles into only one generator with negligible extra parameters introduced. There are also many other methods [LFY*17a, ZD17, CYL*17] proposed to address this problem, but they are all restricted to a fixed number of styles. Later on, the works of [HB17, GLK*17, LFY*17b, SLSW18] introduce algorithms to transfer arbitrary style through only one single model. In short, these Fast Style Transfer algorithms can be classified into three categories: One-Style model [ULVL16, JAFF16, UVL17, L-W16], Multi-Styles model [DSK16, LFY*17a, ZD17, CYL*17] and Arbitrary-Style model [HB17, GLK*17, LFY*17b, SLSW18].

The above methods don’t take stroke-size control into account, which is an essential perceptual factor in paintings. Gatys et al. [GEB*17] first design a method to produce large stroke size for the high-resolution image, which can be used to change the stroke size but is apparently very slow and inflexible. Recently, Jing et al. [JLY*18] propose a module named stroke-pyramid to continuously control the stroke size of the generated images in real time. However, this method suffers from the following drawbacks: (1) the number of stroke sizes is fixed, and their method cannot adapt to new size without training the newly-introduced parameters; (2) the method of [JLY*18] needs a lot of parameters when a wide range of stroke sizes are needed, which is very wasteful. What’s more, it remains unclear how effective this kind of structure is when used in Multi-Styles model or even Arbitrary-Style model.

In this work, we develop a novel style transfer method to tackle the aforementioned problems. Our work stems from the intuition that in stroke-pyramid [JLY*18], different residual units might share much degree of computation. However, this sharing will be thrown away by training different residual units as in [JLY*18].
Thus, we design a recurrent convolutional neural subnetwork to replace the original stroke-pyramid, which we call recurrent stroke-pyramid. Consequently, this kind of recurrent computation can automatically increase the receptive field of the generator without introducing extra parameters, making it flexible to generate larger unseen stroke sizes than the training set. Besides, it can save a lot of parameters when a wide range of stroke sizes are needed. Finally, we extend our proposed method to the Multi-Styles and the Arbitrary-Style models to incorporate different styles into one single model while the stroke size of each style can be controlled without any image processing before or after one-shot forwarding.

Our contributions are threefold: (1) We re-analyze the function of each residual unit used in [JLY*18] and propose a novel recurrent convolutional neural subnet to control stroke size in Fast Style Transfer. (2) Experiments demonstrate the high efficiency and flexibility of our method for parameters reduction, ease of training, generalization to unseen larger stroke size, and the capability to produce a wide range of stroke sizes with only one residual unit. (3) We newly design the network architecture to embed this pyramid structure into the Multi-Styles and the Arbitrary-Style models, and present two novel run-time control strategies to concurrently control the style and the stroke size in an entirely feed-forward manner.

2. Background and related work

Below we briefly introduce the background and relevant methodologies in Fast Style Transfer.

**Style Normalization.** Style normalization (SN) is a domain transformation technique applied to each sample image independently, which is commonly used in Fast Style Transfer [UVL16, DSK16, HB17, GLK*17, LFY*17b, SLSW18]. It transforms the current domain \( D_{\text{c}} \) back to the common domain \( \tilde{D} \) and then to the style domain \( D_{i} \) in the feature space of a CNN, where \( \tilde{D} \) contains the basic information, e.g., orientation, but dispels several details, e.g., amplitude. Specifically, let \( x \in \mathbb{R}^{C \times (H \times W)} \) denote the input to the SN layer, then the output of it can be described as:

\[
y = W_c W_c^{-1}(x - \mu_c) + \mu_s,
\]

where \( W_c, W_s \in \mathbb{R}^{C \times C} \) are transformation matrices respectively for \( D_{\text{c}} \) and \( D_{i}; \mu_c, \mu_s \in \mathbb{R}^C \) represent the channel-wise mean. It should be noted that \( W_c \) and \( \mu_c \) are derived from \( x \) while \( W_s \) and \( \mu_s \) data from [UVL16, DSK16] or a style encoder [HB17, GLK*17, LFY*17b, SLSW18].

Among these methods, Instance Normalization (IN) [UVL16] is first proposed to improve the stylization quality, but it only normalizes different images into one single style. When multiple styles are considered, \( W_i \) and \( \mu_i \) should be augmented to independent pairs with each one conditional on a specific style, which is known as Conditional Instance Normalization (CIN) proposed by Dumoulin [DSK16]. In addition, given \( W_i \) and \( \mu_i \) computed from a style encoder, the whole network will be able to transfer arbitrary style according to the works in [HB17, GLK*17]. Later on, Li et al. [LFY*17b] discover that a well-designed SN layer, which they call Whitening and Coloring Transformation (WCT), can be effectively used in the auto-encoder architecture to achieve zero-shot style transfer. Very recently, Sheng et al. [SLSW18] introduce the Style Decoration (SD) module to improve WCT for better quality.
However, these auto-encoder-based methods will find it hard to recover fine-grained style patterns due to the downsampling operations.

**Stroke-size Control.** Stroke size plays an essential part in paintings, which measures how big a corresponding brush is used and how much distance it moves. Thus, stroke size can be defined as the average scale of the similar-sized stroke texons \([JLY^{*}18,Jul81,ZGWX05]\) in an area. There are three primary requirements for stroke-size control: (1) generating semantically distinguishable images no matter which stroke size is used; (2) continuously controlling the stroke size in a feed-forward manner; (3) preserving the stroke consistency \([JLY^{*}18]\) that the generated images of different stroke sizes should exhibit consistent stroke orientation, configuration, etc.

In the Fast Style Transfer literature, the One-Style model \([ULVL16,JAFF16]\) can train multiple models to obtain different stroke sizes. However, it is very uneconomical and inflexible. Wang et al. \([WOZW17]\) design a hierarchical convolutional neural network to paint large strokes in high-resolution images. However, it inherently entails multiple models for different stroke sizes. Furthermore, the works in \([ULVL16,JAFF16,WOZW17]\) dissociate requirement (2) since one model only produce one stroke size. Though Some SN methods \([HB17,GLK^{*}17,LFY^{*}17b,SLSW18]\) can achieve arbitrary-style transfer with one single model, they are not good at controlling stroke size when the receptive field is fixed (see Section 4.2.2). Moreover, they need image processing to achieve different stroke sizes, which is very inflexible. To resolve this dilemma, Jing et al. \([JLY^{*}18]\) recently discover that the generator with larger receptive field can produce larger stroke size and propose a module named stroke-pyramid to endow the network with adaptive receptive fields. It consists of several stroke branches, namely separate residual units, with each one having a larger receptive field than the previous one. As a result, this method perfectly meets the requirements with only one model, becoming the state-of-the-art method.

Nevertheless, another problem occurs that compared with a network without stroke-pyramid, the work in \([JLY^{*}18]\) requires \(L \times 2 \times (9C^{2} + 2C)\) additional parameters, where \(C\) is the number of residual units’ channels and \(L\) the number of stroke branches. Furthermore, if \(C\) is large (e.g., \(C = 128\)), overall parameters will come from the residual units and increase linearly with \(L\). Therefore, it seems wasteful to apply their method to wide-ranging stroke sizes. Besides, their approach can not adapt to new stroke size without retraining the newly-introduced parameters. On the other hand, since they only focus on the One-Style model, it’s unclear how to extend their method to the Multi-Styles and the Arbitrary-Style models.

Note that our work is intrinsically different to \([JLY^{*}18]\) as we propose a novel technique to control the stroke size, and newly design the network architecture and loss function to explore the effectiveness of it for diverse styles in one single model.

3. The Method

3.1. Recurrent Stroke-Pyramid

Our work stems from the intuition that in stroke-pyramid, different residual units have the similar function: fine-tune the feature map from the previous one to let the stroke decoder produce larger stroke size, which means they might share much degree of computation. Therefore, training separate residual units for different stroke sizes will leave this sharing vulnerable to be broken.

Taking this into consideration, we design the recurrent stroke-pyramid which is a recurrent convolutional neural network and inserted in the encoder-decoder architecture (see Section 3.2 for details). The main part of it is a residual unit \([HZRS16a]\) similar to the proposed method of \([HZRS16b]\), where a kind of SN layer is substituted for the Batch Normalization \([IS15]\) layer. This structure has two advantages. First, the parameters are shared across the whole procedure. Second, inserting convolutional computation into the recurrent neural network can endow the generator with adaptive receptive fields, which is the key to controlling the stroke size according to \([JLY^{*}18]\). In each step, the input to the residual unit comes from the output feature map in the previous step (or the input feature map if this is the first step). Thus, after \(R\) steps, our recurrent stroke-pyramid will output a sequence of \(R\) feature maps.

An illustration of the recurrent stroke-pyramid can be as seen in Figure 2. Here, let \(F^{0}\) denote the input feature map to the module and define convolution, style normalization and Relu nonlinearity as \(\phi_{C}(\cdot)\), \(\phi_{SN}(\cdot)\) and \(\phi_{R}(\cdot)\) respectively, then the output feature map \(F^{j}(i = 1, \cdots, R)\) in the \(i\)th step that corresponds to the \(i\)th bigger stroke size \(\Gamma_{i}\), can be computed by:

\[
F^{j} = F^{(i,j)} = F^{(0)} + \sum_{j=0}^{i-1} \phi_{SN}(\phi_{C}(\phi_{R}(\phi_{SN}(\phi_{C}(F^{j+1})))))) = F^{(i,j)} + \sum_{j=0}^{i-1} \phi_{R}(F^{j+1}; w)
\]

where \(w\) denotes the parameters of the residual unit. In this formula, the residual unit will fine-tune the feature map \(F^{j+1}\) by adding the term \(\phi(F^{j+1}; w)\) in the \(i\)th step. Having observed that, we expect this addition will consistently tell the decoder to produce larger stroke size than \(F^{j+1}\) does. Besides, this recurrent computation can enforce stroke consistency (see Section 2) and automatically increase the receptive field of the whole network without introducing extra parameters, making it possible to generate larger unseen stroke size during testing flexibly. It can also save a lot of parameters when a wide range of stroke sizes is needed, which will wastefully induce many separate residual units to be trained in \([JLY^{*}18]\).

3.2. Architecture

Figure 3 shows our proposed architecture. It consists of four parts: encoder, recurrent stroke-pyramid, fusion module and decoder. The convolution layers are all followed by a kind of SN layer to increase the receptive field of the whole network without introducing new extra parameters, making it possible to generate larger unseen stroke sizes while preserving stroke consistency. The padding mode is set to reflect-padding. All the activation functions are Relu nonlinearity except the sigmoid function in the last layer. The encoder is stacked by three convolution layers and two residual units while the decoder consists of two up-sampling layers \([ULVL16]\). Notice that the style control and the stroke-size control are mutually independent thanks to the separation of the convolution and the normalization operations.

**SN Layer.** For single-style transfer, we use the IN layer. For multi-styles transfer, the CIN \([DSK16]\) layer is applied to our network. For arbitrary-style transfer, we use the similar style normalization method as in \([GLK^{*}17]\), where the parameters \(\mu_i\) for each style are computed from the Inception v3 \([SVI^{*}16]\). We normalize \(W_i\) and \(W_e\) as diagonal matrices whose diagonal elements...
A simple mask-out operation.

The stroke size within one image is given a list of masks

\[ \sum_{j=1}^{R} \{ \alpha \} F \]

during training, each style image will be resized to \( R \) different sizes and each size corresponds to one specific stroke size. \( N_{\text{meta}} \) will update the parameters of \( N_f \) based on current style. Details are described in Section 3.2.

Recurrent Stroke-pyramid. According to Section 3.1, this module will receive the feature map from encoder and output a sequence of feature maps. Then the fusion model will receive it and use one of the three functions to generate the output feature map for the decoder. During training, each style image will be resized to \( R \) different sizes and each size corresponds to one specific stroke size. \( N_{\text{meta}} \) will update the parameters of \( N_f \) based on current style. Details are described in Section 3.2.

### 3.3. Loss Function

Content Loss. Based on the works in [GEB15b, ULVL16, JAF-B16], the content loss is formulated as the Euclidean distance between the content target \( I_c \) and the generated image \( I \) in the high-level feature space of the VGG network [SZ14]. Let \( \Psi_l(x) \) denote the feature map of shape \( C_l \times H_l \times W_l \) in the \( l \)th layer of the loss network \( \Psi \) with input image \( x \). The content loss \( L_c \) is defined as:

\[
L_c(I, I_c) = \frac{1}{2C_lH_lW_l} \| \Psi_l(I) - \Psi_l(I_c) \|_2^2. \tag{3}
\]

Style Loss. Let \( G(\Psi_l(x)) \) denote the Gram matrix of \( \Psi_l(x) \) that represents the style statistics [GEB15a], then the style loss \( L_s \) is the Frobenius norm of the difference between \( G(\Psi_l(I)) \) and \( G(\Psi_l(I_c)) \), where \( I_c \) indicates the style target. During training, for one feature map \( F^l \) selected by the fusion module, the style image \( I^l_c \) of stroke size \( \Gamma_l \) is used. Then, the style loss for the output image \( F^l \), generated by feeding \( F^l \) to the decoder, can be described as:

\[
L_s(F^l, I^l_c) = \sum_{l \in \{l\}} \alpha_l \frac{1}{2C_lH_lW_l} \| G(\Psi_l(F^l)) - G(\Psi_l(I^l_c)) \|_2^2, \tag{4}
\]

where \( I^l_c \) is obtained by resizing \( l \) based on \( \Gamma_l, N_l = H_l \times W_l, \{l\} \) is the set of loss network layers used to compute the style loss, and \( \alpha_l \) is the weight for the \( l \)th layer. During training, the weight for high-level layers will increase linearly with current stroke size to enhance the large pattern.
In order to transfer multiple or even arbitrary style, different styles should have their own $W_i$ and $\mu_i$, as discussed in Section 2. We use $N_{meta}$ to abstractly represent the mapping from style image $I_i$ to its corresponding parameters $w_0$. The total loss for $I^Γ_i$ can be then written as:

$$L_Γ = \sum_{I \in \{I_i\}} \lambda_c L_c(I^Γ_i, I_i) + \lambda_a L_a(I^Γ_i, I_i^Γ) + \lambda_t L_t(I^Γ_i), \quad (5)$$

where $I^Γ_i = N_{f}(I_i; w, Γ_i)$; $N_{f}$ denotes the generator; $w = (w_d, w_0)$, $w_d$ represents the parameters shared by different styles, $w_0 = N_{meta}(I_i)$; total variation loss $L_t$ is added for piece-wise smoothness; $\lambda_c$ is set to 1.0 while $\lambda_a$ and $\lambda_t$ are left as hyper-parameters. The Algorithm 1 in the Appendix details our training strategy.

4. Experiments

4.1. Details

We train our proposed network using MS-COCO [LMB14] as content images with a batch size of 8. They are resized and randomly cropped to $256 \times 256$ patches. We let the recurrent time be 5 as default and use 5 scales for each style image to achieve different stroke sizes. During training, the pre-trained VGG-16 network [SZ14] is chosen as the loss network and relu4_2 is used as the content layer while {relu1_1.relu2_1.relu3_1.relu4_1.relu5_1} as the style layers where {relu3_1.relu4_1.relu5_1} are set to be high-level. Adam [KA15] optimizer is applied with a learning rate of $1.0 \times 10^{-3}$. All of our experiments are implemented on TensorFlow†. The content and the style images used in the experiments can be found in Figure 13 and Figure 14 in the Appendix.

4.2. Evaluation of Our Recurrent StrokePyramid

In this section, we focus on the evaluation of the recurrent stroke-pyramid. We use three kinds of evaluation metric: quantitative comparison, qualitative comparison and flexibility comparison. IN is used in our network for a fair comparison with [JLY18].

† https://www.tensorflow.org
4.2.1. Quantitative Comparison

In order to see what is being traded off by incorporating different residual units into only one in a recurrent manner, we use the same training strategy to train our proposed model and the model in [JLY*18], then compare them regarding the convergence speed and the final average loss of different stroke sizes. We set the number of stroke branches in [JLY*18] to be 5 for consistency with our recurrent time. Also, we let their encoders and decoders respectively be the same structures. Table 1 shows the parameters usage comparison between these two models. In this experiment, the parameters of the model in [JLY*18] are more than twice that of ours.

Table 1. Differences between our approach and other methods.

<table>
<thead>
<tr>
<th>Parameters(5 sizes)</th>
<th>Requirements</th>
<th>Receptive Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>[JAFF16]</td>
<td>8.39M</td>
<td>×</td>
</tr>
<tr>
<td>[GLK*17]</td>
<td>3.82M</td>
<td>×</td>
</tr>
<tr>
<td>[LFY*17b]</td>
<td>34.24M</td>
<td>×</td>
</tr>
<tr>
<td>[SLSW18]</td>
<td>7.01M</td>
<td>×</td>
</tr>
<tr>
<td>[JLY*18]</td>
<td>2.27M</td>
<td>✓</td>
</tr>
<tr>
<td>Ours</td>
<td>1.09M</td>
<td>✓</td>
</tr>
</tbody>
</table>

Top two figures in the Figure 5 compare the learning dynamics for each stroke size between two models. The losses are all averaged over 32 random batches of content images and 10 styles. By visual inspection, we can observe that for each stroke size, ours converges as quickly as [JLY*18] with respect to both the style and the content losses. In order to quantify this observation, we show the final loss difference for each stroke size between these two models below. Roughly, the difference is negligible since our proposed model’s average final content loss over 5 different stroke sizes is 0.57% lower than [JLY*18] while average final style loss difference is even less (0.51% higher). What’s more, as stroke size becomes larger, the style loss difference stabilizes (see stroke size $\Gamma_4$ and $\Gamma_5$), which is about 2.7% higher, also very insignificant. Therefore, ours can achieve quantitatively competitive results while improving the parameter-usage efficiency.

4.2.2. Qualitative Comparison

Figure 4 and Figure 8 show several samples of ours and [JLY*18], it is evident that the two models produce closely similar results. Specifically, they can generate semantically clear images with wide-ranging stroke sizes and preserve the stroke consistency that all the strokes change together only in size, namely satisfying the requirement (1) and (3) (Section 2). Note that we use much fewer parameters than [JLY*18] to achieve comparable visual results. Furthermore, our approach is more flexible, as shown in 4.2.3.

We also visually compare our method with [JAFF16, GLK*17, LFY*17b, SLSW18] in Figure 4. To obtain multiple stroke sizes, [JAFF16] needs to wastefully train multiple networks while Arbitrary-Style models [GLK*17, LFY*17b, SLSW18] should first resize input style images to different scales. As we can see, [JAFF16] will collapse the stroke consistency due to the separate learning of different models (e.g., the inconsistent stroke orientations on the left of the house in columns 1 through 3). The style encoder in [GLK*17] can’t correctly parse the style patterns and result in unwanted ones when the stroke size mismatches its fixed and limited receptive field (e.g., the conspicuous discrepancy in overall hue between the original style and the stylized images with $\Gamma_3$ and $\Gamma_5$). Since [LFY*17b] and [SLSW18] employ an auto-encoder structure, when the stroke size is tiny, the decoder fails to completely reconstruct the subtle details due to the pooling layers and generate semantically vague images (see the stylized images with $\Gamma_3$ and $\Gamma_5$). The full curve represents our proposed method while the dashed curve the method of [JLY*18]. Below is the final average loss comparison for each stroke size.

Figure 5. Top two figures compares the training procedure. For one specific stroke size, it is represented by two curves with the same color (red - $\Gamma_1$, yellow - $\Gamma_1$, green - $\Gamma_3$, purple - $\Gamma_4$, black - $\Gamma_5$). The full curve represents our proposed method while the dashed curve the method of [JLY*18]. Below is the final average loss comparison for each stroke size.
trol of stroke size with economical processing cost, and they need much more parameters than ours. Table 1 summaries the above comparisons.

![Figure 6. Comparisons between fine-tuning and learning from scratch in terms of style loss curve and quality.](image)

Figure 6. Comparisons between fine-tuning and learning from scratch in terms of style loss curve and quality.

![Figure 7. Our recurrent stroke-pyramid generalizes well to unseen stroke size without any fine-tuning. Emphasis (red box) has been added on the results for comparison purpose.](image)

Figure 7. Our recurrent stroke-pyramid generalizes well to unseen stroke size without any fine-tuning. Emphasis (red box) has been added on the results for comparison purpose.

4.2.3. Flexibility Comparison

**Learning to Produce Larger Unseen Stroke Size.** In order to produce unseen larger stroke size, [JLY*18] should retrain the network by incrementally learning a new branch on the top of stroke-pyramid. Although this method achieves faster convergence than training the whole network from scratch, it still needs hundreds of iterations to converge [JLY*18]. It should be noted that ours can achieve this goal by lengthening the recurrent time during testing. This is because the decoder learns to be sensitive to the changes in its input feature map caused by different recurrent times. The longer the recurrent time is, the larger stroke size should be achieved. To quantify this observation, we first augment a new larger stroke size upon the pre-trained network by fine-tuning it using the proposed training strategy (see the Algorithm 1 in the Appendix). Then, we compare the loss curve and stylization quality with that of optimizing the network from scratch (Figure 6). While achieving comparable results, fine-tuning nearly converges since the beginning, which indicates the network generalizes well to the larger unseen stroke size. We also let it produce unseen stroke size without any fine-tuning (Figure 7).

**Producing a Wide Range of Stroke Sizes.** In this case, our recurrent stroke-pyramid only needs to lengthen the recurrent time to be much longer (eg., 10 steps) for the continuous control of stroke size during testing. But for the model in [JLY*18], it should use 10 different residual units to achieve comparable results, which seems extremely wasteful and liable to overfitting. Besides, during testing, the computation graph of our model can be adjusted flexibly depending on the actual demand by only changing the recurrent time while for [JLY*18], it seems a little troublesome to do this, especially on the tensorflow platform. Figure 8 shows several results. While achieving comparable results, ours shows less high-frequency noise and saves a lot of parameters, making it easier to train and more suitable for mobile development.

![Figure 8. Visual comparisons of producing a wide span of stroke sizes.](image)

Figure 8. Visual comparisons of producing a wide span of stroke sizes.

4.3. Multi-styles Stroke-controllable Transfer

In this section, we embed our recurrent stroke-pyramid into the Multi-Styles Model. Clearly, we use CIN [DSK16] as our S-N method to transfer multiple styles while the stroke size for each style could be regulated by the recurrent stroke-pyramid. We call our proposed model using IN as 1-style model while that using CIN as N-styles model, where N indicates the number of styles. We first train 10 1-style models and 1 10-styles model, then compare between them. Finally, we execute two run-time control strategies described in Section 3.2, further manifesting the flexibility of it.

4.3.1. Comparison

To validate the effectiveness of the embedding method, we randomly sample 5 styles and compare the final style and content losses for each stroke size between 1-style models and 10-styles model (left column of Figure 9). The losses are all averaged over 32 random batches of content images. Generally speaking, for style loss the 10-styles model achieves around 1.43% ± 5.02% lower losses than its counterparts while for content loss, the difference is more negligible (about 0.64% ± 1.37% higher). Since our N-styles model uses only one model to transfer multiple styles and stroke sizes, this difference can be ignored in the view of much parameters reduction compared to N 1-style models. The right column of Figure 9 shows several visual comparisons between N-styles model and N 1-style models. We can see that both results are qualitatively similar.

4.3.2. Run-time Control

**Strategy 1. Concurrently Interpolating Between Styles and Stroke Sizes.** Similar to [DSK16], interpolating between different styles’ transformation parameters $\theta_i$ and $\mu_i$ can fuse them together. During style fusing, if we use the Function 2 of our fusion module detailed in Section 3.2, then the stroke size of the fused style can
also be continuously controlled, as shown in Figure 10. It can be seen that the stroke size is changed fairly smoothly. While our model is only trained to control the stroke size for the style chosen from the fixed style set, it generalizes naturally to the fused style.

**Strategy 2. Concurrently Executing Style and Stroke Size Spatial Control.** Since the feature statistics in the generator can reflect the style of the output image, applying CIN separately to different regions via region masks can achieve style spatial control. The masks for different layers can be obtained by using nearest neighbor interpolation according to current layer’s stride. By concurrently using the Function 3 of our fusion module detailed in Section 3.2, where the masks are obtained by the same method, ours can achieve both stroke size and style spatial control in a completely feed-forward manner. Figure 11 shows some results.

### 4.4. Arbitrary-style Stroke-controllable Transfer

According to Section 4.3, our recurrent stroke-pyramid can successfully control the stroke size for each style in the fixed style set. Thus, it’s natural to combine the arbitrary style transfer architecture with it to flexibly control the stroke size of arbitrary style without any image pre-processing. Here, we employ the pre-trained network proposed by [GLK*17] and simply add our recurrent stroke-pyramid after its fifth residual unit. The recurrent stroke-pyramid is also followed by a fusion module to control the stroke size of the output image. It should be noted that \( W_s \) and \( \mu_s \) in the recurrent stroke-pyramid are also computed from the pre-trained Inception-v3 [SVI*16], just like the way in [GLK*17]. During training, we use our proposed training strategy to first train the newly-added parameters with others fixed and then fine-tune the whole network, on the DTD dataset [CMK*14]. Figure 1 shows some results (image (e) and (f)). More results can be found in the Figure 12. It can be seen that the network generalizes well to the unseen styles.

![Figure 9](image9.png) Figure 9. The left column is the final average loss comparisons for each stroke size between 1-style models and N-styles model. Different shapes represent different stroke sizes. The right column is the visual comparisons between them.

![Figure 10](image10.png) Figure 10. Interpolation between both styles and stroke sizes.

![Figure 11](image11.png) Figure 11. (a) Content image. (b) The mask for style and stroke size spatial control. The yellow region corresponds to the style image \( 1 \) of Figure 14 in the Appendix and stroke size \( \Gamma_1 \), the red region the style image (b) and stroke size \( \Gamma_3 \), the black region the style image (d) and stroke size \( \Gamma_5 \). (c) The stylized result without stroke size spatial control. (d) The stylized result with stroke size spatial control.

© 2018 The Author(s)
5. Conclusion

In this work, we propose a recurrent convolutional neural subnetwork to control the stroke size of the generated images in Fast Style Transfer, which is a brand-new concept and also a novel technique in the style transfer literature. It can automatically increase the receptive field of the whole network without introducing extra parameters, making it flexible to generate larger unseen stroke size than during training. What’s more, it can save a lot of memories when a wide range of stroke sizes is needed. Experimental results demonstrate that ours can not only achieve comparable results with much fewer parameters but allow more flexibility and efficiency, compared with the original method. In addition, the proposed method can be successfully extended to Multi-Styles model or even Arbitrary-Style model to control both the style and the stroke size in an entirely feed-forward manner. However, the combinational optimization over multiple styles and stroke sizes will compromise the rendering quality (e.g., row 6 and 7 in Figure 12). Thus, further research is needed to generate high-quality images for arbitrary-style stroke-controllable transfer.

Acknowledgments

This work is partially supported by the National Science Foundation of China under Grant no. 61502306, 61601112, the Fundamental Research Funds for the Central Universities, DHU Distinguished Young Professor Program and the China Young 1000 Talents Program.
References


6. Appendix

Figure 13 and Figure 14 show the content images and style images respectively used in Section 4.2 and 4.3. Algorithm 1 details our training strategy.

Algorithm 1 Our Training Strategy

\begin{algorithm}
\begin{algorithmic}
\FOR {number of training iterations}
\STATE Sample a style image $I_s$ from $\{I_s\}$
\STATE $w \leftarrow (w_d, w_y), w_y \leftarrow N_{meta}(I_s)$
\FOR {$i$ in $\{1, 2, ..., R\}$} \DO
\STATE $I_{\Gamma i} \leftarrow \text{Resize}(I_s, \Gamma_i)$
\STATE Sample a mini-batch of content: $\{I_{c j}\}_{m}^1$
\STATE $\{I_{\Gamma i}^j\}_{m}^1 \leftarrow N_f(\{I_{c j}\}_{m}^1; w, \Gamma_i)$
\STATE Compute gradients and update $w$.
\ENDFOR
\ENDFOR
\end{algorithmic}
\end{algorithm}

Figure 13. Diverse content images used in 4.2 and 4.3.

Figure 14. Diverse style images used in 4.2 and 4.3.