Spatial Control in Neural Style Transfer

Tom Henighan
Stanford Physics
henighan@stanford.edu

Abstract

Recent studies have shown that convolutional neural networks (convnets) can be used to transfer style with stunning results [5]. To expand upon this, it is desirable to enable spatial control over this transfer of style [6]. Here I explore methods for achieving spatial control in neural style transfer. I focus on two distinct regimes. The first is one which aims to slowly transition between styles in different regions of the image. The second regime is sharp boundaries between different styles, for instance between foreground and background. I propose a method, receptive averaging, which gives qualitatively good results in both regimes, with minor artifacts present in the second regime. I also show that the use of gradient masking in the second regime surprisingly gives more significant artifacts near the boundary between different styles. Finally, I show that artifact-free results and be achieved in the second regime by simply training two images separately with different styles, and then stitching them together.

1. Introduction

Texture is known to play a key role in the human visual system to allow, for instance, distinguishing an orange from a tennis ball. Algorithms which effectively distinguish and parametrize texture are thus useful for visual recognition as well as synthesis of realistic visual scenes. Work in this area prior to the emergence of convnets achieved impressive results, as demonstrated by authors who effectively synthesized images whose texture matches a target image [4, 15, 3], used texture to fill in occluded regions of an image [4], used texture-matching to stitch together different images [11], and transfer the texture of one image onto another [3, 8, 2, 12]. Building upon this last task, authors showed they could transfer "style" by transferring the texture of, for instance, a painting onto a photograph [8, 2, 12]. While the results are impressive, they are restricted to using relatively low-level features to interpret the style transfer.

Convolutional neural systems allow parametrization of high level image features, which has led to improved state of the art performance in computer vision tasks including object recognition [7], segmentation [13], and image captioning [14]. It stands to reason that clever use of high level semantic information would allow for more realistic style transfer algorithms. Following this line of thinking, Gatys et al using the convolutional filters of the 19 layer VGG-net pre-trained on ImageNet achieve stunning style transfer results [5]. In follow up work, the same authors presented methods for transferring different styles to different spatial regions of the output image, among other things [6]. The authors specifically focus on having sharp boundaries between spatial regions of one style and another. I here attempt to expand upon this work by exploring alternative methods for spatial control of style transfer. I focus on two distinct regimes: smooth transitions between different styles, and very sharp boundaries between different style regions. For the first regime, I achieve the desired qualitative result using receptive averaging to generate soft masks which are applied when calculating the gram matrices of the output image activation layers. For the second regime, I show that receptive averaging still works surprisingly well, better than masking of the gradients during backpropagation. However, this performance is superseded by simply training the entire image on different styles and then stitching the regions together.

2. Base Algorithm Summary

The goal of the base algorithm is to create an image which has the "style" of one image (the style image) and the content of another (the content image) [5]. This is achieved using a dual loss function, where minimization of the content loss ensures the content of the output image closely matches that of the content image, and the style loss ensures the style of the output image matches that of the style image. The output image is then produced by adjusting the output image to minimizes the combined loss.

2.1. VGG-net

In the algorithm of Gatys et al, all three images (style, content, and output) are all fed through the 19 layer version

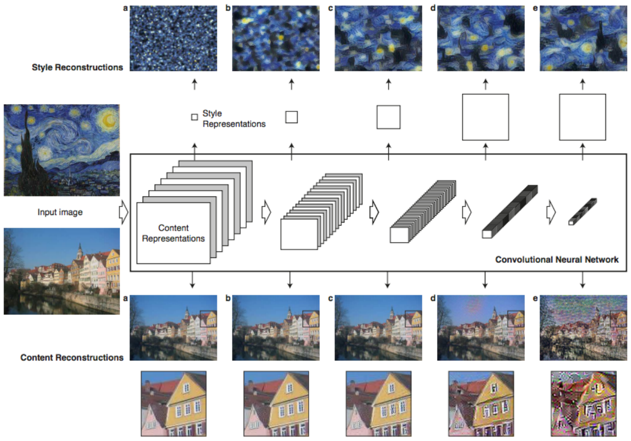

Figure 1. From Gatys et al [5]. Image representations in a Convolutional Neural Network (CNN). A given input image is represented as a set of filtered images at each processing stage in the CNN. While the number of different filters increases along the processing hierarchy, the size of the filtered images is reduced by some downsampling mechanism (e.g. max-pooling) leading to a decrease in the total number of units per layer of the network. **Content Reconstructions**. We can visualise the information at different processing stages in the CNN by reconstructing the input image from only knowing the networks responses in a particular layer. We reconstruct the input image from from layers conv1_2 (a), conv2_2 (b), conv3_2 (c), conv4_2 (d) and conv5_2 (e) of the original VGG-Network. We find that reconstruction from lower layers is almost perfect (ac). In higher layers of the network, detailed pixel information is lost while the high-level content of the image is preserved (d,e). **Style Reconstructions**. On top of the original CNN activations we use a feature space that captures the texture information of an input image. The style representation computes correlations between the different features in different layers of the CNN. We reconstruct the style of the input image from a style representation built on different subsets of CNN layers (conv1_1 (a), conv1_1 and conv2_1 (b), conv1_1, conv2_1 and conv3_1 (c), conv1_1, conv2_1, conv3_1 and conv4_1 (d), conv1_1, conv2_1, conv3_1, conv4_1 and conv5_1 (e). This creates images that match the style of a given image on an increasing scale while discarding information of the global arrangement of the scene.

of VGG-vet [1]. The convolution or pool at each layer gives a spatial representation of some high-level features in the input image. The losses will be constructed by comparison of these feature maps of the content, style, and output images. Following the notation of Gatys et al, let's denote different layers by subscript l and call the number of filters and number of pixels in a given layer N_l and M_l, respectively. We then store the response of the output image at layer l as $F^l \in \mathcal{R}^{N_l \times M_l}$. So F^l_{ij} is the activation of the ith filter at position j in layer l for the output image. We then denote P^l and S^l as the responses of the content and style images, respectively, at layer l. Note that we are flattening the 2D images to one dimension.

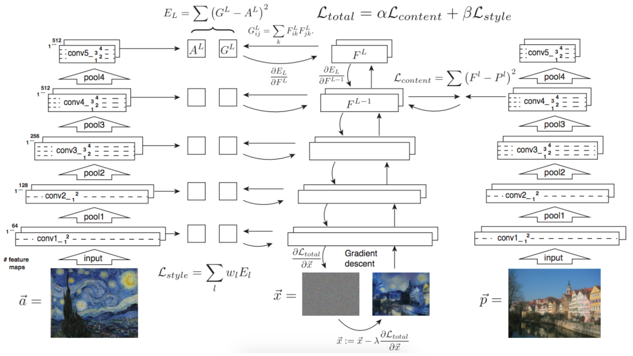
$E_L = \sum (G^L - A^L)^2$
$\mathcal{L}_{total} = \alpha\mathcal{L}_{content} + \beta\mathcal{L}_{style}$
$G^L_{ij} = \sum_k F^L_{ik} F^L_{jk}$.
512
conv5_3
A^L
G^L
F^L
pool4
$\frac{\partial E_L}{\partial F^L}$
$\frac{\partial E_L}{\partial F^{L-1}}$
$\mathcal{L}_{content} = \sum (F^l - P^l)^2$
pool4
512
conv4_3
F^{L-1}
conv4_3
pool3
pool3
256
conv3_3
conv3_3
pool2
pool2
128
conv2_1
conv2_1
pool1
pool1
64
conv1_1
conv1_1
# feature maps
input
$\frac{\partial \mathcal{L}_{total}}{\partial \vec{x}}$
Gradient descent
input
$\mathcal{L}_{style} = \sum_l w_l E_l$
$\vec{a} =$
$\vec{x} =$
$\vec{p} =$
$\vec{x} := \vec{x} - \lambda \frac{\partial \mathcal{L}_{total}}{\partial \vec{x}}$

Figure 2. From Gatys et al [5]. Style transfer algorithm. First content and style features are extracted and stored. The style image is passed through the network and its style representation A^l on all layers included are computed and stored (left). The content image is passed through the network and the content representation P^l in one layer is stored (right). Then a random white noise output image is passed through the network and its style features G^l and content features F^l are computed. On each layer included in the style representation, the element-wise mean squared difference between G^l and A^l is computed to give the style loss $\mathcal{L}_{\text{style}}$ (left). Also the mean squared difference between F^l and P^l is computed to give the content loss $\mathcal{L}_{\text{content}}$ (right). The total loss $\mathcal{L}_{\text{total}}$ is then a linear combination between the content and the style loss. Its derivative with respect to the pixel values can be computed using error back-propagation (middle). This gradient is used to iteratively update the output image until it simultaneously matches the style features of the style image and the content features of the content image (middle, bottom).

2.2. Content Loss

The content loss is then given by

$$\mathcal{L}_{\text{content}} = \frac{1}{2} \sum_{ij} (F^l_{ij} - P^l_{ij})^2 \tag{1}$$

Where the layer l used above is a hyperparameter. It is empirically found that for small l, minimizing the above produces output images which closely match the content image. As l is increased, the output image becomes an increasingly distorted version of the content image.

2.3. Style Loss

In contrast to the content image, the mapping between the output image and the style image is not spatially direct. For instance, if the style image has a sun in the top right, there need not be a sun in the top right corner of the output image to match the style of the style image. Thus it seems we want to match *delocalized* features of the style image. A

[1] I will be using the 16 layer version, at least to start to reduce the number of parameters and convergence time.

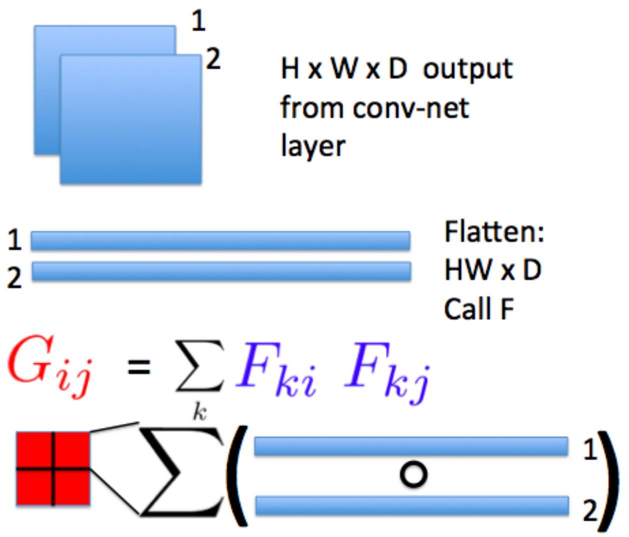
1
2
H x W x D output from conv-net layer
1
2
Flatten: HW x D
Call F
$G_{ij} = \sum_k F_{ki} F_{kj}$
1
2

Figure 3. Illustration of Gram matrix calculation. In this toy example, the convolutional layer of interest has two filters, giving two images of size H × W. The spatial dimensions are flattened onto a single dimension. Elements of the gram matrix are then given by the dot product of these vectors, where the ijth component of the matrix is given by the dot product of the output from filters i and j. Here I have written out the dot product as an element-wise multiplication followed by a summation to allow for easy visualization of masking in future sections.

simple way to retain information of the high-level features of the convnet layers while ignoring direct spatial information is to take a dot product along the image dimension. Doing this for all filter pairs produces the $\in \mathcal{R}^{N_l \times N_l}$ Gram matrix, which I've illustrated in Fig. 3. The Gram matrix for the output image is then given by

$$G^l_{ij} = \sum_k F^l_{ik} F^l_{jk} \tag{2}$$

and for the style image by

$$A^l_{ij} = \sum_k S^l_{ik} S^l_{jk} \tag{3}$$

The contribution of layer l to the total loss is then

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{ij} (G^l_{ij} - A^l_{ij})^2 \tag{4}$$

Matching to higher l leads to matching of style image features of increasing size. To preserve the style at several scales, the style loss is defined as a weighted sum of the loss at different layers

$$\mathcal{L}_{\text{style}} = \sum_l w_l E_l \tag{5}$$

where the w_l are the weights.

2.4. Total Loss

The total loss is then given by a weighted sum of the content and style loss

$$\mathcal{L} = \alpha \mathcal{L}_{\text{content}} + \beta \mathcal{L}_{\text{style}} \tag{6}$$

where the ratio of α to β is a scalar hyperparameter which allows one to tune the relative importance of style and content matching.

2.5. My Implementation

I have implemented the 16-layer of VGG-net (VGG-16) in tensorflow using pre-trained weights, as well as the image class labels, found on the internet [1]. The same link provided an implementation of vgg-16 in tensorflow which was useful as a guide. However I found it advantageous to write my own VGG-16 implementation to make it more concise and integrate into the rest of my codebase. Using a few test images I confirmed the network correctly classifies the examples, indicating the network is implemented correctly.

Here we are doing full batch training. As such, I found L-BFGS-B [16] training to converge faster than Adam [10] optimization. I used the Scipy [9] implementation of L-BFGS-B which is part of the experimental/volatile module of tensorflow [2]. However, manipulation of the gradients used in the optimization was challenging with the experimental scipy optimizer. In section 4.1 where gradient manipulation was desired, I instead used adam for this reason.

Style images were re-sized to have the same number of pixels as the content image, but retain the same aspect ratio. If the content image had more than 250,000 pixels, it was resized to have approximately this many pixels while retaining the same aspect ratio. The output image always had the same dimensions as the content image. Max pooling layers were replaced with average pooling layers, as suggested by Gatys et al [5].

Found in Fig. 4 are intermediate images produced when training on the content loss alone (β=0), where the output image was initialized as gaussian noise. Upon training, this gaussian noise begins to look more and more like the style image (in this case, a picture of a weasel). On a personal note, I would like to say that the intermediate images of this training image can look kindof creepy, particularly in the early stages.

Found in Fig. 5 are the images produced when training on the style loss alone (α=0) for only a single layer (i.e. w_l is 1 for a single l and zero for all others). This illustrates that the network is indeed capturing the "style," with deeper layers capturing higher-level and larger-scale features.

[2] Specifically, I used tensorflow.contrib.opt.python.training.external_optimizer.ScipyOptimizerInterface. For more details, see https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/opt/

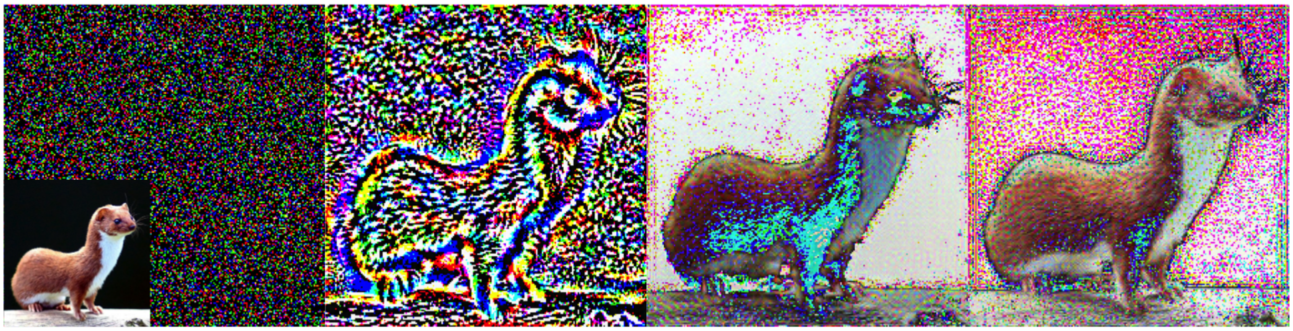

Figure 4. Results of minimizing content loss alone at various stages of training, where the images on the right side are produced from more training iterations. The image is initialized as gaussian noise. The content image used is seen in the inset in the bottom left.

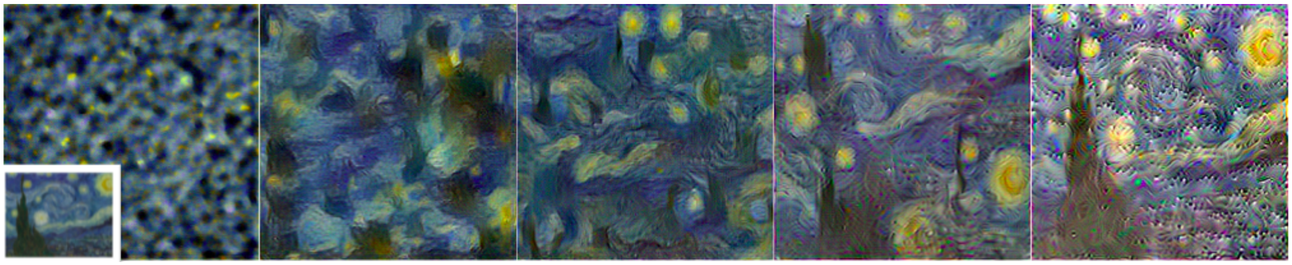

Figure 5. Results of minimizing style loss alone (α=0) for different convolutional layers, specifically layers conv1_1, conv2_1, conv3_1, conv4_1, and conv5_1 from left to right. Images were initialized with gaussian noise and ”Starry Night” by Van Gogh was used as the style image (seen in inset). Deeper layers seem to capture higher-level and larger scale features. It is interesting to note that the right-most image nearly reconstructed the style image.

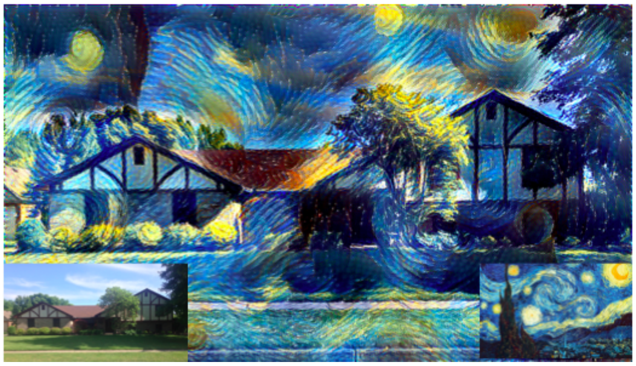

Figure 6. Example output of my implementation of neural style transfer using VGG-16. The content image is a photo of my parents' house in Ohio, while the style image is ”Starry Night” by Van Gogh. Here the ration α/β was 1e-2.

With α and β both non-zero, the content image is rendered in the style of the style image as desired. An example output is shown in Fig. 6. In all examples used here, the style weights w_l are zero for all layers except conv1_1, conv2_1, conv3_1, conv4_1, and conv5_1, for which they are equal and sum to 1. Layer conv3_2 was used as the content matching layer.

3. Gram Matrix Masking

As discussed in Section 2.3, we seek to match the gram matrices of the convolutional layers of the output and style images. The Gram matrix is found by taking inner products over all spatial pixels of different filters in a given layer. A natural way to to achieve spatial control is then to introduce a mask at this stage, so the dot product of is no longer spatially uniform. This is illustrated in Fig. 8. This idea is central to the spatial control proposed by Gatys et al [6]. Given a mask for layer l, $T^l \in \mathcal{R}^{N_l}$, the masked Gram matrix for the output image is then given by

$$G^l_{ij} = \sum_k F^l_{ik} F^l_{jk} T^l_k, \tag{7}$$

as illustrated in Fig. 8.

However, as the authors point out, given the desired mask for the output image, the choice of appropriate masks T^l at the convolutional layers is not obvious. This is because pixels in the later convolutional layers have large receptive fields. Thus the rest of this section focuses on how to choose the T^l given the desired soft mask for the style on the output image.

python/training/external_optimizer.py

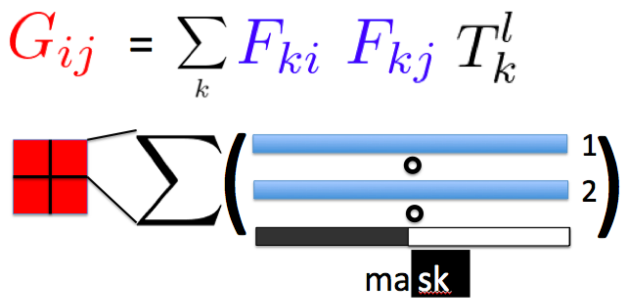

Figure 7. Illustration of masked Gram matrix calculation. Instead of simply taking an element-wise product over all pixels in the two images, one *also* multiplies this with a mask, to weigh how much each pixel contributes to the Gram matrix. After element-wise multiplication, the values are summed as before.

3.1. Inside Guidance

In [6], the authors focus on binary masking for hard boundaries between segmented parts of the image. Their solution is to unmask a pixel in the Gram matrix calculation only if it has *zero* pixels in the unmasked region of the output image in its receptive field. This is known as "inside guidance." However, doing this alone leaves the boundary regions unstylized. To remedy this, the authors use an eroded or dilated version of inside guidance. However they also mention that the step of erosion/dilation requires some tuning, as the parameters for best results vary.

3.2. Receptive Averaging

Here I propose a similar method which focuses on the case of smooth transitions between styles rather than hard boundaries. Thus our mask for the output image is a soft mask (values between 0 and 1) as opposed to a binary mask. We must then find the appropriate soft masks for the Gram matrices at each convolutional layer. My approach is to feed the initial soft-mask for the output image through a drastically simplified version of the network. Specifically, at each layer the convolutional kernels have the same height (H) and width (W) as the original filter, but only one filter (depth of 1), and the values of the kernel are all 1/(HW). Additionally, all non-linearities are removed. This yields the *average value* of the mask over all pixels in the receptive field of a single pixel in the activation map. Note that the convolutional operations are now equivalent to average pooling. I call this method of mask propagation "receptive averaging." It should be noted this is distinct from the method of re-sizing the mask at each layer, which is mentioned in passing by Gatys et al [6]. Note that if simply resizing, a convolution which preserves spatial dimensions would have the same mask, whereas the method proposed here (receptive averaging) would not in general.

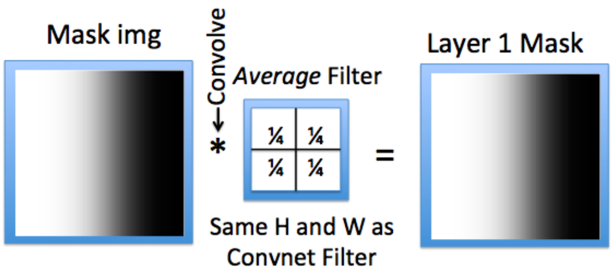

Figure 8. Illustration of receptive averaging method used to define soft masks for Gram matrices in convolutional layers. Given the desired soft mask for the output image, we convolve it with an averaging kernel of the same height and width of the kernels in the first layer of the convnet used (VGG-16 in this case). This produces a soft mask for layer 1, where the value of each pixel in the activation map is the average over the pixels within its receptive field. Note this is equivalent to replacing the convolutional operation with an average pooling of the same size and stride. The soft mask for layer 2 would be made using the same procedure, with the layer 1 mask as the input and using an average kernel with the same height and width of the layer 2 kernel in the original convnet. Non-linearities in the original network are removed. Average/max pooling layers are kept but not pictured here.

3.3. Results

Examples using receptive averaging can be found in Fig. 9. In these examples, two styles and one content image are used, with the style smoothly transitioning from one style to the other horizontally across the image. Indeed, it appears the desired results are achieved.

4. Alternative Methods for Binary Masking

While the spatial-control results of Gatys et al [6] are impressive, they require some tuning at the erosion/dilation step. A method which can produce sharp boundaries between different styles without tuning is therefore desirable. I propose and try a few simple methods here. The best results are achieved with arguably the simplest method: training two separate images with different styles and stitching them together. Unexpectedly, receptive averaging comes in second, achieving a sharp boundary between styles with only minimal artifacts. Finally, the method of gradient masking also give sharp boundaries, but also more noticeable artifacts near the boundary.

4.1. Gradient Masking

During optimization, one calculates the gradient of the loss, given in Eq. (6). Since the total loss is given by a superposition of the content loss and style loss, we could calculate these gradients independently and then add them (weighting by α and β) to get the gradient of the total loss. Thus if we did *not* want to transfer style to one part of the

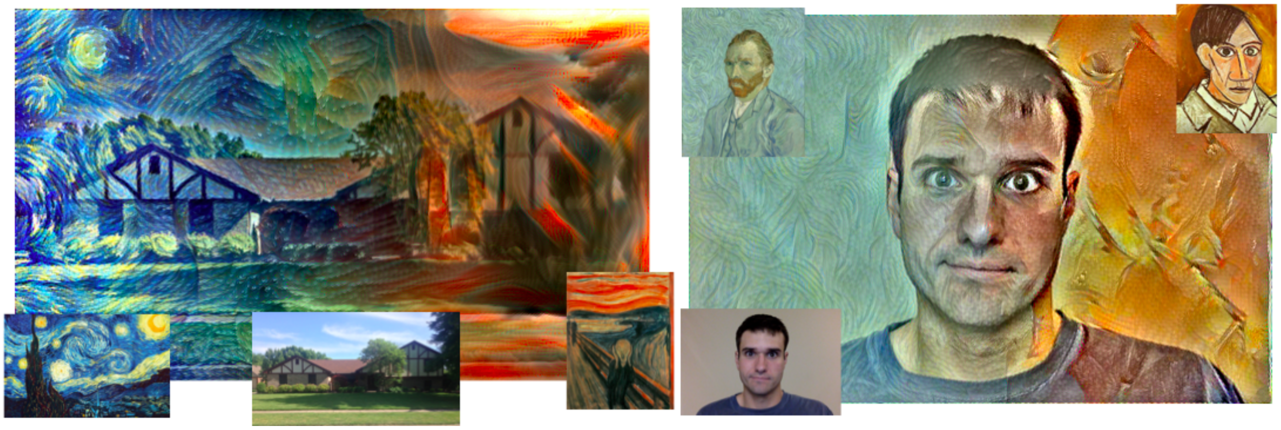

Figure 9. Example outputs of Receptive averaging. This allows for mixing of two different styles in one image, where we can have smooth transitions from one style to another. In the images above, a horizontal sigmoid soft mask was used for the right style, and its complement (one minus that mask) was used for the left style. The left content image is a photo of my parents' home in Ohio, with the style of ”Starry Night” by Van Gogh on the left, and ”The Scream” by Edvard Munch on the right. The right content image is a photo of me with the style of self-portraits of Van Gogh and Picasso on the left and right respectively.

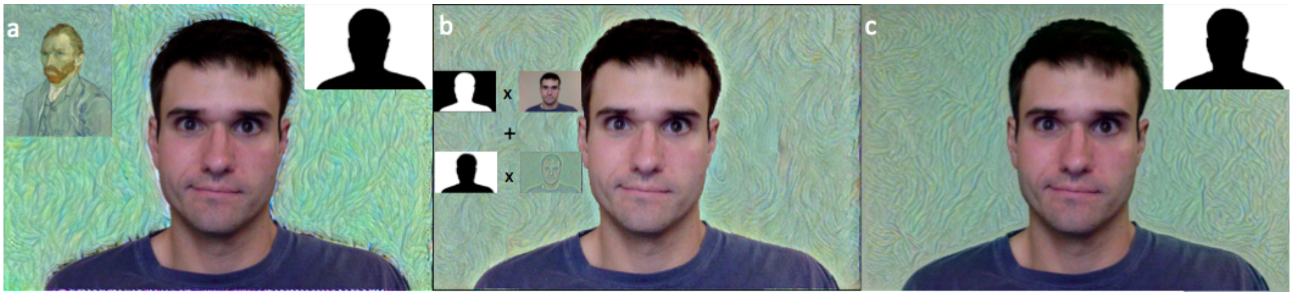

Figure 10. Example outputs for different methods of style transfer with binary masking. In all three images, the goal is to stylize the background with the Van Gogh portrait in the top left inset, while the foreground matches the content image. The results of gradient masking are found in subfigure a, with the mask found in the inset (black is zeros, white ones). The output image was initialized with the content image. In b, the background from an unmasked stylized image is stitched together with the foreground of the content image. The results of naively using receptive averaging are found in c. For c the output image was initialized to the content image.

image, one could artificially set the gradients of the style loss with respect to those pixels to zero, add them to the gradients of the content loss (again, weighting by α and β), and use this gradient for optimization. Thus instead of masking the activation maps when calculating the loss, one simply masks the gradients of the output image after doing backpropagation. From here forward I will call this procedure “Gradient Masking.”

An example output using gradient masking is found in Fig. 10 a. The style image is found in the top left inset while the binary mask is found in the top right. The image was initialized to the content image. While the content of the foreground was effectively unaltered as desired, the regions just outside the boundary show artifacts. I must say I do not fully understand the origins of these artifacts. Naively, I would have expected that we are effectively restricting the optimization to fewer dimensions, but that over time, the pixels which we allow to vary would converge to something matching the style of the style image in all unmasked regions.

On a technical note: it was not obvious how one could manipulate the gradients during optimization when using the experimental implementation of the scipy optimizer in tensorflow. For this reason, training was done on this image with the adam optimizer.

4.2. Post-Optimization Stitching

An even simpler alternative is apply neural style transfer to a single content image several times, each with different styles, and then stitch these together. Fig. 10 b shows an example output of this procedure. While uninventive, the result is as desired. Unsurprisingly, there are no artifacts near the foreground-background boundary.

4.3. Receptive Averaging

One can also try using receptive averaging (introduced in Section 3). Propagating the mask forward will produce soft masks (i.e. not binary, but between 0 and 1) at each layer. However, doing so will not necessarily produce the desired hard boundary. Activation map pixels whose receptive fields include both masked and unmasked pixels in the output image will provide channels for gradients to flow back to some of the masked pixels, blurring the boundary.

In Fig. 10 c, one finds an example output using receptive averaging with the mask seen in the top-right inset. This result is not perfect, but much better than I would have expected, with minimal artifacts. The faint artifacts that are there are present on both sides of the style boundary.

5. Conclusions

Here I explore methods for spatial control in neural style transfer.

First with a focus on soft masks allowing for smooth transitions between styles in a single image, I present receptive averaging. In receptive averaging, activation-layer masks based on the desired mask for the output image are used in the Gram matrix calculations. The masks for each layer are produced by propagating the desired output mask through a simplified version of the convnet, where the nonlinearities are removed and convolutions are replaced by average pooling using a kernel of the same height, width, and stride as those in the convnet. This method shows promising results for its intended purpose: creating smooth transitions between different styles within the output image.

Second, I show results for three different methods of creating sharp boundaries between different styles in the output image with binary masks. The first is to mask the gradients during backpropagation rather than masking the loss function itself. While this does create a sharp boundary, it produces artifacts near the edge. The second is using receptive averaging as was done for soft masks in the first part of the paper. Despite the fact that this method does not block gradients from the style of one region leaking into the other, this method gives relatively good results, with minimal artifacts near the border. The third and final method simply stitches together two pre-optimized images: where the entirety of each image was rendered with only one style. Although boring, I would say this gives the best results as there are no artifacts near the border separating the regions of the two styles.

References

[1] Vgg in tensorflow. `https://www.cs.toronto.edu/~frossard/post/vgg16/`. Accessed: 2017-06-12.

[2] N. Ashikhmin. Fast texture transfer. *IEEE Computer Graphics and Applications*, 23(4):38–43, 2003.

[3] A. A. Efros and W. T. Freeman. Image quilting for texture synthesis and transfer. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 341–346. ACM, 2001.

[4] A. A. Efros and T. K. Leung. Texture synthesis by non-parametric sampling. In *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, volume 2, pages 1033–1038. IEEE, 1999.

[5] L. A. Gatys, A. S. Ecker, and M. Bethge. Image style transfer using convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2414–2423, 2016.

[6] L. A. Gatys, A. S. Ecker, M. Bethge, A. Hertzmann, and E. Shechtman. Controlling perceptual factors in neural style transfer. *arXiv preprint arXiv:1611.07865*, 2016.

[7] K. He, X. Zhang, S. Ren, and J. Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition. In *European Conference on Computer Vision*, pages 346–361. Springer, 2014.

[8] A. Hertzmann, C. E. Jacobs, N. Oliver, B. Curless, and D. H. Salesin. Image analogies. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 327–340. ACM, 2001.

[9] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed 2017-06-12].

[10] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[11] V. Kwatra, A. Schödl, I. Essa, G. Turk, and A. Bobick. Graphcut textures: image and video synthesis using graph cuts. In *ACM Transactions on Graphics (ToG)*, volume 22, pages 277–286. ACM, 2003.

[12] H. Lee, S. Seo, S. Ryoo, and K. Yoon. Directional texture transfer. In *Proceedings of the 8th International Symposium on Non-Photorealistic Animation and Rendering*, pages 43–48. ACM, 2010.

[13] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3431–3440, 2015.

[14] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and tell: A neural image caption generator. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3156–3164, 2015.

[15] L.-Y. Wei and M. Levoy. Fast texture synthesis using tree-structured vector quantization. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 479–488. ACM Press/Addison-Wesley Publishing Co., 2000.

[16] C. Zhu, R. H. Byrd, P. Lu, and J. Nocedal. Algorithm 778: L-bfgs-b: Fortran subroutines for large-scale bound-constrained optimization. *ACM Transactions on Mathematical Software (TOMS)*, 23(4):550–560, 1997.