– COS 429 Final Report –

Neural Style Transfer Replication Project

Haochen Li Yuyan Zhao
Advisor: Riley Simmons-Edler

1. Introduction

In this project, we will explore the problem of neural style transfer using deep learning. The neural style transfer problem involves taking a content image **C** and a style image **P** and combine them to produce a new image that has the content of **C** and the style of **P**.

There are three major advancements in the area of neural style transfer. In 2015, the paper, A neural algorithm of artistic style [2], proposes an iterative algorithm for neural style transfer. In 2016, the paper, Perceptual losses for real-time style transfer and super-resolution [3], proposes a real-time neural style transfer algorithm. However, for this algorithm, we have to train a seperate neural network for each style. In 2017, the paper, A learned representation for artistic style [1], proposes a real-time neural style transfer algorithm that can transfer a variety of styles with one neural network.

Our project is an implementation-heavy project. In this project, we implemented A neural algorithm of artistic style [2] and Perceptual losses for real-time style transfer and super-resolution [3] from the scratch. We have also done fairly extensive analysis. We did not have time to study A learned representation for artistic style [1].

2. Motivation

We will talk about the motivation of neural style transfer and the motivation of our project separately.

The authors of [2] noted that humans have the ability to create great art through a complex interplay between content and style of an image. However, there is no good algorithmic way of doing it. The authors want to develop an interesting algorithmic way of doing so. Furturemore, the authors hope that their work could help understand how humans create and perceive artistic imagery in an algorithmic way.

A neural algorithm of artistic style [2] proposes an iterative algorithm to solve the problem of neural style transfer. The authors of Perceptual losses for real-time style transfer and super-resolution [3] want to speed up the algorithm.

We want to replicate their algorithms because we want to learn more about deep learning and we think that implementing published deep learning papers from scratch would be a great way of doing so. Moreover, we just think that neural style transfer is a really cool problem.

3. Approach

Both papers made use of pretrained VGG net. Neither paper used the fully connected layers. The first paper(iterative algorithm) uses a pretrained VGG19 net, the second paper(real-time) uses the pretrained VGG16 net.

3.1. A neural algorithm of artistic style

In this paper, the main insight is that the representations of style and content of an image are seperable in convolutional neural network. This paper uses a pretrained VGG19 network. They input three images in the VGG19 network at the same time, a content image **C**, a style image **S**, and an image that is initialized to white noise(or the content image or the style image) **I**. All three images are passed through the network at the same time. Let $layer_i$ denote a layer in the VGG19 network, and let $layer_i(x)$ denote the output of layer i that takes an input x. For content reconstruction, let I_i denote the result of running **I** through VGG19, stopping after the ith layer, let C_i denote the result of running **C** through VGG19, stopping after the ith layer. Pick any layer of the VGG19 net, we can use the output of that layer as the content representation. Suppose we picked layer i, we calculate the content loss by compute the MSELoss between I_i and C_i. For style representation, let S_i denote the result of running **S** through VGG19, stopping after the ith layer. To represent the style, we need to use something called the Gram Matrix. The Gram matrix captures the correlation between different pixels in an image, while ignoring the specific details(content) of the image, thus making it a good candidate to represent style. Let G denote the Gram matrix(for the definition of Gram matrix, please refer to [2]), the style representation of style image using the ith layer be $G(S_i)$. The style loss is calculated by computing the MSELoss between $G(I_i)$ and $G(S_i)$. In the original paper, they used sum of style losses from multiple layers of VGG19 as the final style loss. Furthremore, we specifiy a content weight, C_W, and a style weight, S_W.

$$Total_{loss} = C_W * content_{loss} + S_W * style_{loss}$$

For each iteration, we try to minimize the total loss function by optimizing the values of **I**.

3.2. Perceptual losses for real-time style transfer and super-resolution

In this paper, they train a neural network for each style. Here is the network architecture used in [3](Taken directly from the original paper):

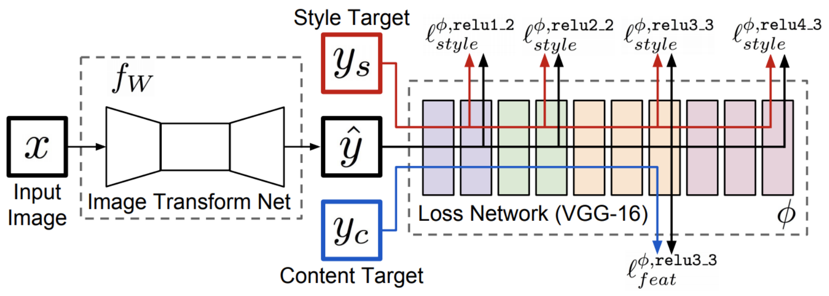
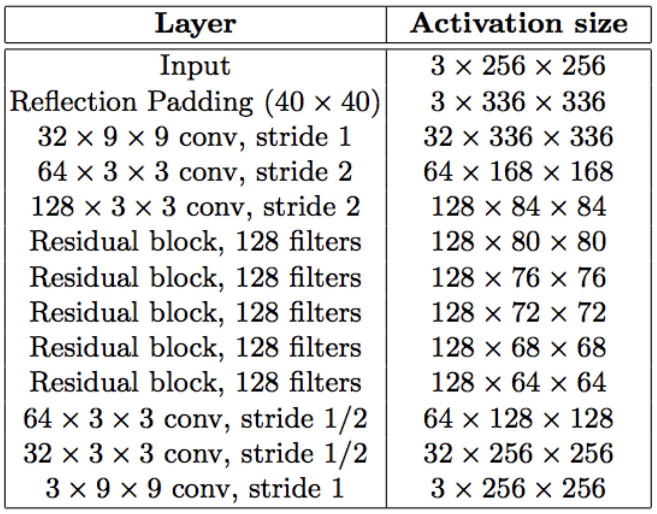

Style Target
$\ell^{\phi,relu1\_2}_{style}$
$\ell^{\phi,relu2\_2}_{style}$
$\ell^{\phi,relu3\_3}_{style}$
$\ell^{\phi,relu4\_3}_{style}$
y_s
f_W
x
$\hat{y}$
Input Image
Image Transform Net
y_c
Loss Network (VGG-16)
ϕ
Content Target
$\ell^{\phi,relu3\_3}_{feat}$

Fig. 2. System overview. We train an *image transformation network* to transform input images into output images. We use a *loss network* pretrained for image classification to define *perceptual loss functions* that measure perceptual differences in content and style between images. The loss network remains fixed during the training process.

Here is the detailed architecture of image transformation network(taken from the supplementary information of this paper):



Layer	Activation size
Input	$3 \times 256 \times 256$
Reflection Padding (40×40)	$3 \times 336 \times 336$
$32 \times 9 \times 9$ conv, stride 1	$32 \times 336 \times 336$
$64 \times 3 \times 3$ conv, stride 2	$64 \times 168 \times 168$
$128 \times 3 \times 3$ conv, stride 2	$128 \times 84 \times 84$
Residual block, 128 filters	$128 \times 80 \times 80$
Residual block, 128 filters	$128 \times 76 \times 76$
Residual block, 128 filters	$128 \times 72 \times 72$
Residual block, 128 filters	$128 \times 68 \times 68$
Residual block, 128 filters	$128 \times 64 \times 64$
$64 \times 3 \times 3$ conv, stride 1/2	$64 \times 128 \times 128$
$32 \times 3 \times 3$ conv, stride 1/2	$32 \times 256 \times 256$
$3 \times 9 \times 9$ conv, stride 1	$3 \times 256 \times 256$

Table 1. Network architecture used for style transfer networks.

In this paper, x, the input image to the image transformation net, is the same as the content target. They pass y_s, $\hat{y}$, and y_c through VGG16 net and calculate the style and content loss as in [2]. However, in this paper, they are optimizing the parameters of the image transformation net. The idea is to have the image transformation net output a good $\hat{y}$ that can minimize the total loss function in the previous section. For the training set, they used the COCO Dataset [4] as content images and a single style image. In this approach, they have to train a separate network for every style image.

4. Data Collection

4.1. A neural algorithm of artistic style

For replicating this paper, we don't need a training set since we are not training any network(we are essentially training the initialized image). We collected 22 style images and 20 content images from Google Images, and we will present some of the results in the analysis section. Here are some examples of the content and style images we collected:

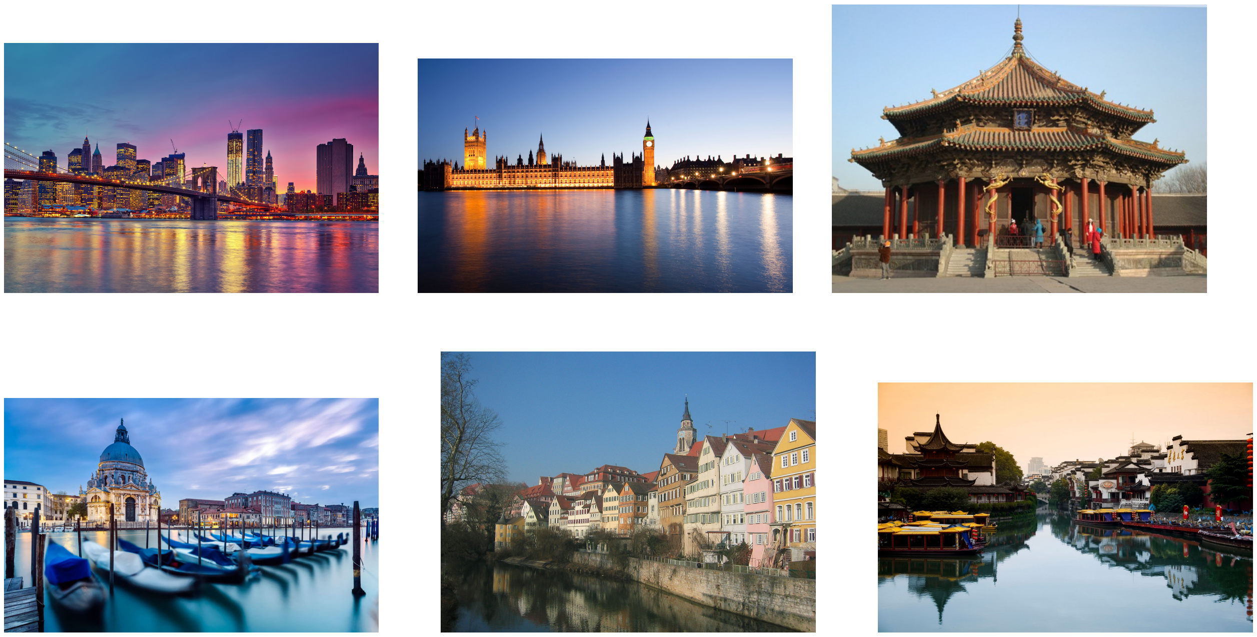


Figure 1: Content Images

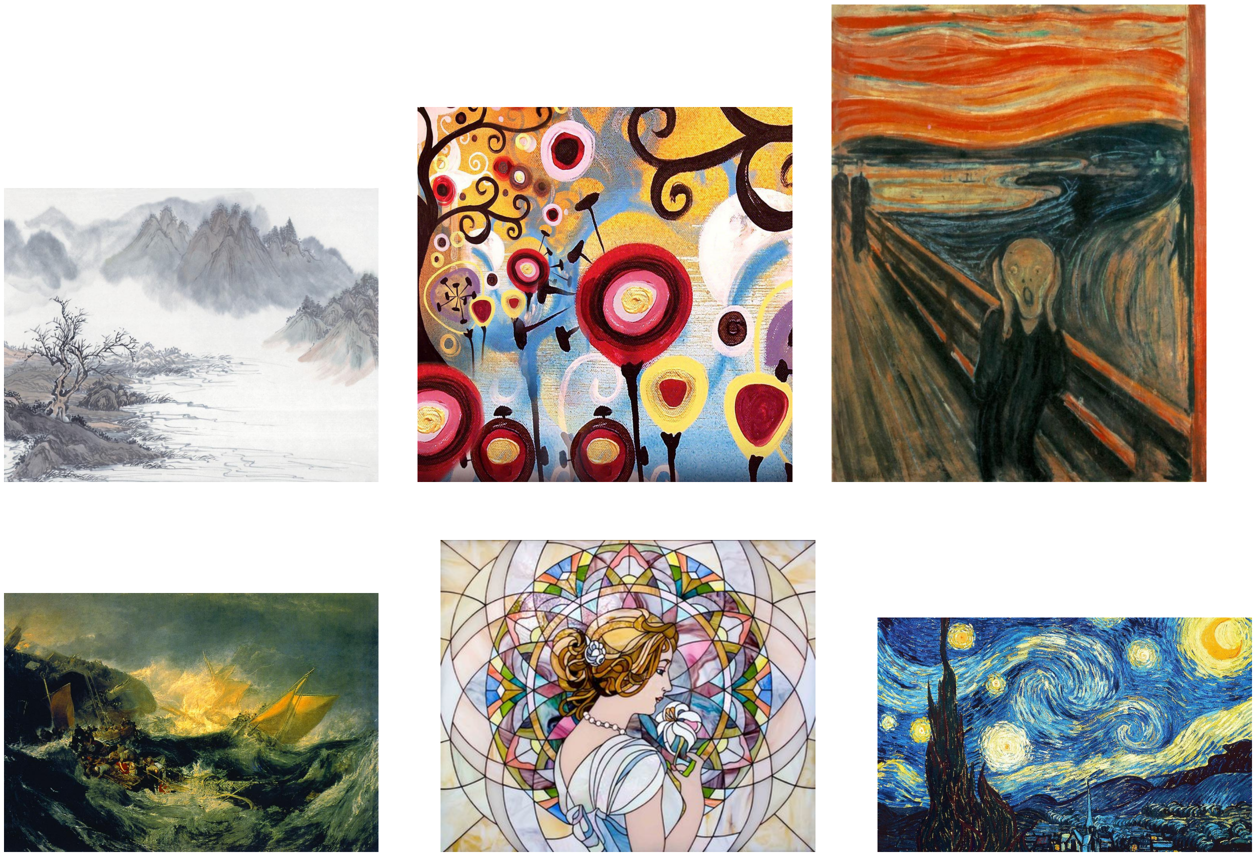


Figure 2: Style Images

4.2. Perceptual losses for real-time style transfer and super-resolution

For replicating this paper, we need the COCO dataset [4]. We first downloaded the COCO dataset [4], then we removed all non-RGB images, and resized all images in the dataset to 256 by 256. We used the same set of 22 style images as above.

5. Summary of Implementation

First of all, we would go over things that are common to both papers we replicated. We used the **p2.xlarge** instance on AWS. We chose to use **p2.xlarge** because we want to use GPU and **p2.xlarge** is the cheapest option that offers a GPU on AWS. We choose PyTorch as our deep learning framework because is enables us to use Python and it generates computational graph dynamically, making it easier to debug. In both projects, we need to upload data we collected to AWS EC2 using FileZilla.

After finishing the project, we think that implementing published research papers from scratch is very tricky as research papers are not written in a way that is very friendly for people who are relatively new to deep learning to replicate. However, we are able to understand deep learning a lot better by understanding published research papers enough to implement it. In addition, by doing this project, we understand PyTorch a lot better.

5.1. A neural algorithm of artistic style

For implementing this algorithm, it takes very little time to upload data to EC2. However, the implementation of this algorithm took us the longest since we don't understand the neural style transfer algorithm and we don't know PyTorch very well. PyTorch is still a relatively new framework and the documentation sometimes is not as comprehensive as we would like them to be.

There are two points in the paper that confused us in the beginning. The first one is that if we are calculating the style and content loss only using a subset of layers in VGG19 net, do we need to pass the images through every layer of the VGG19 Net? After a more detailed reading of the paper and watching a talk given by the author on Youtube, we decide that we should pass the images through all layers of VGG19 net. We also tried only passing the images through a subset of VGG layers(we tried using VGG19 net up to the 8th Convolutional layer), we did not obtain good results at all(many results resemble neither the content nor the style image). We are able to obtain results visually similar to the published paper by using all layers of the VGG19 net.

At the beginning, we were also confused regarding how to optimize the initialized image. We were confused because we have only seen cases where we optimize a set of parameters in some neural network before. By understanding this, we gained a better understanding of how optimizer works in deep learning.

The first step in implementation is to get the images into the formats required by PyTorch. We did so by writing an ImageDataSet class, that turns the images in a certain folder into numpy arrays. We then used dataloader and torch.vision.transforms class that turn the numpy arrays that represent images into Tensors, the datatype required by PyTorch framework. We then used Dataloader that enables us to iterate thorough the set of images we obtained, making it possible to train.

The second step is to implement the process of training(optimizing the initialized image). We encountered some major challenges in this step.

The first challenge is the implementation of Gram Matrix. We were very sure that our implementation is right. However, using my implementation, the style loss decreases during the first one or two iterations, and then the style loss stays at a very high level. The optimizer will optimize the content loss to 0. As a result, the output is always an image that is almost identical to the content image and has almost no resemblance to the style of style image. We spent two days trying to debug and had no progress. We then decided to go online and take the implementation of Gram matrix from this website: Pytorch Tutorial. Here are the lines of code we took:

```
class GramMatrix(nn.Module):

    def forward(self, input):
        # a=batch size(=1)
        a, b, c, d = input.size()
        # b=number of feature maps
        # (c,d)=dimensions of a f. map (N=c*d)

        features = input.view(a * b, c * d)

        # compute the gram product
        G = torch.mm(features, features.t())

        # we 'normalize' the values of the gram matrix
        # by dividing by the number of element in each feature maps.
        return G.div(a * b * c * d)
```

The **input** parameter in the forward function of GramMatrix is a Variable(which is just a wrapper that contains the tensor). Their implementation kept it as a Variable, but our implementation pulled the tensor out of the Variable. We tested the Gram Matrix function independent of the rest of our system. Our implementation and their implementation gave the same numeric results. However, if we use our implementation in the overall system, the output image is almost identical to content image(do not optimize style loss). If we use their implementation, we get very good results. We tried very hard to figure out why this is happening, but we still don't understand why. We would really appreciate some likely explanations since this will help us if we were to do any projects involving deep learning in the future.

We then proceed to implement the training algorithm. One tricky thing is that we have to be aware of in-place operation in PyTorch. If we overwrite some variables that are needed for gradient computations later via some in-place operations, we would mess up the computational graph. We had to fix a few bugs that are related to this issue during our implementation.

Another tricky part during this implementation is to make PyTorch use GPU. Pytorch code uses CPU by default, and Pytorch's official tutorial for GPU is less than comprehensive. CPU is too slow for our purpose. We had to spend a while to figure this out by looking at Youtube tutorials and reading PyTorch Forums. It basically involves moving the input and the network in and out of GPU at the correct time.

The final quirky part we found about PyTorch is that we had to normalize the pixel values of our input images to [0,1]. Otherwise, we get white noise or blank images as output.

5.2. Perceptual losses for real-time style transfer and super-resolution

After implementing the first algorithm, this algorithm seems a lot easier to implement since the two algorithms have a lot of similarities. The new part about this algorithm is that we actually need to build and train a neural network from the scratch. Building the network is not very challenging since the paper gives us the exact network architecture they used. The only thing we need to be careful about is to get the padding right so that we can get the dimensions of output right. However, training the network from the scratch turns out to be tricky. First of all, it takes about 14 hours to train the network from the scratch on a p2.xlarge instance on AWS. Furthermore, a good value of the loss functions does not always guarantee good performance on the test set. We spent about 4 days on the process of hyperparamters tuning and we still don't think that we have enough time to do hyperparamter tuning. We could have used a more powerful instance on AWS, but the cost would be pretty high. We also had a bug during our first implementation, but we only found out after 14 hours of training. From this mistake, we learned the value of incremental training, that is, to train for a certain number of iterations, then view the output to determine whether our code is likely to contain a bug instead of relying solely on the value of loss functions.

6. Qualitative Analysis

6.1. A neural algorithm of artistic style

The hyperparameters in this algorithm include content weight, denoted as C_W, style wright, denoted as S_W, number of iterations trained, denoted as Ite, layers used to represent content, denoted as l_C, and the layers used to represent style, denoted as l_S

Even with the same style and content images, different parameters can lead to very different outputs. The below images illustrate the impact of the content weight to style weight ratio. As expected, the smaller the ratio is, the greater emphasis the output places on style; the bigger the ratio is, the greater emphasis the output places on content.

For all the images below, $C_W = 1, Ite = 15$, l_C = = Conv4_2, l_S = Conv1_1, Conv1_2, Conv2_1, Conv2_2

For the following 4 rows of images, the leftmost two images are content image and style image respectively. The other images are output images generated with different hyperparameters.

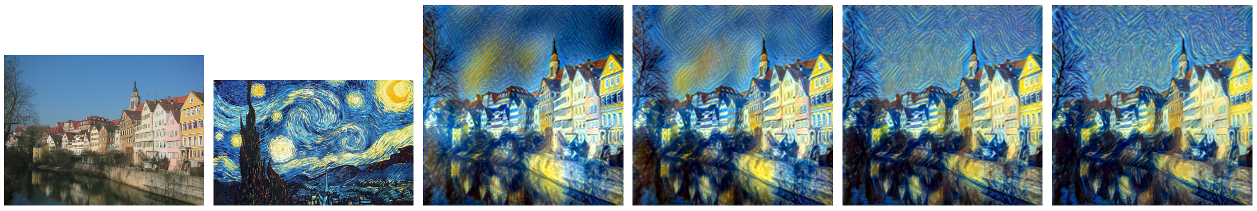


Figure 3: Style Weight = 1000, 2000, 5000, 10000(Left to Right)

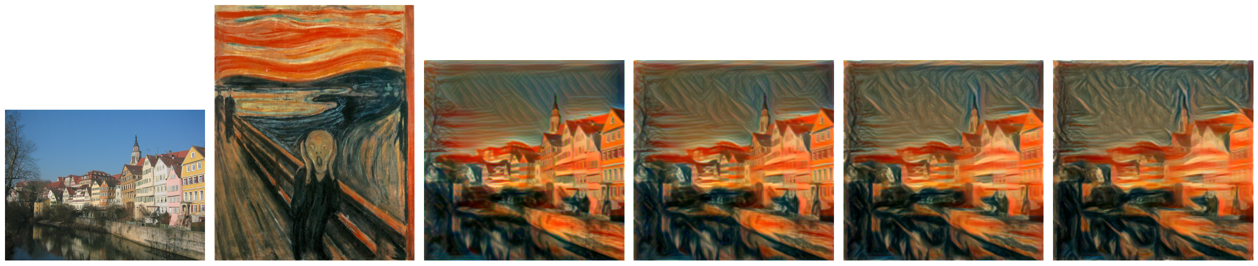


Figure 4: Style Weight = 1000, 2000, 5000, 10000(Left to Right)

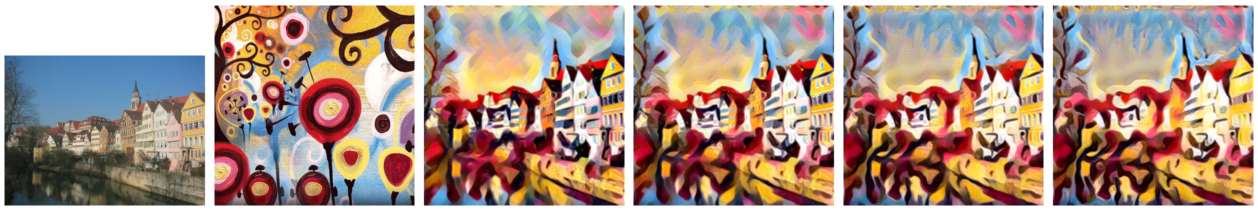


Figure 5: Style Weight = 1000, 2000, 5000, 10000(Left to Right)

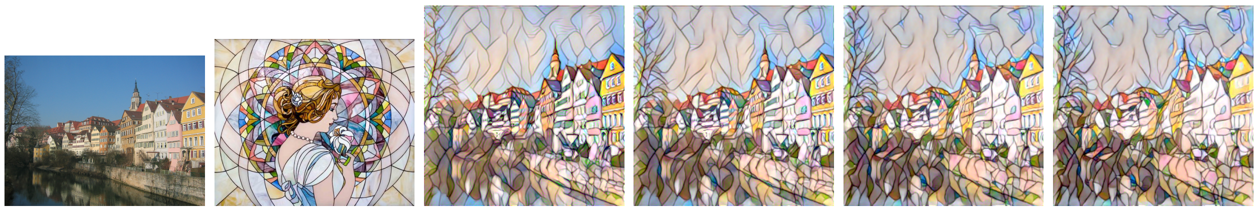


Figure 6: Style Weight = 1000, 2000, 5000, 10000(Left to Right)

Changing the layers with which to represent content or style also has a significant impact on the output. Here we show some examples.

For the belew images, $C_W = 1, S_W = 1000, Ite = 15$, l_C = Conv4_2. The leftmost two images are content and style images respectively. The rightmost two images are output images.

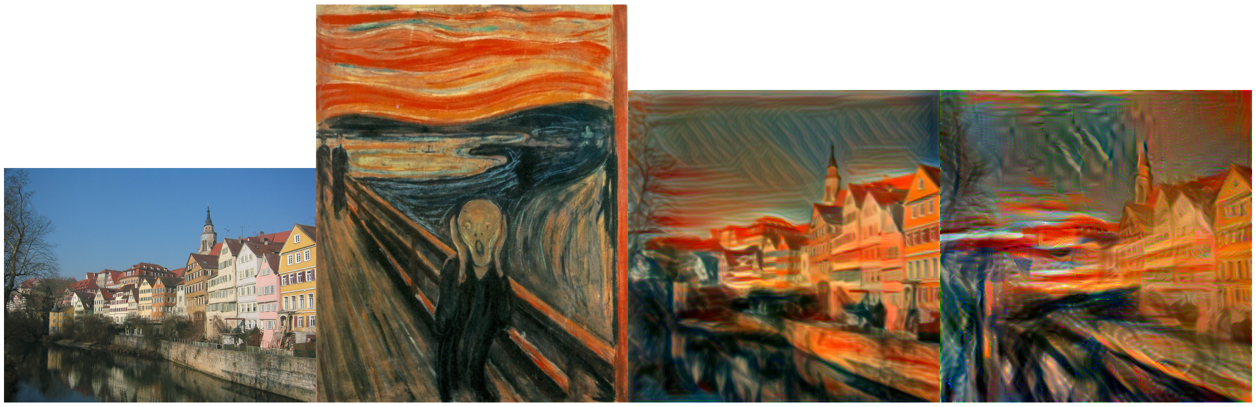


Figure 7: Style layers: Conv1_1, Conv1_2, Conv2_1, Conv2_2(third image from left); Style layers: Conv1_1, Conv2_1, Conv3_1, Conv4_1, Conv5_1(fourth image from left)

For the below images, $C_W = 1, S_W = 2000, Ite = 15$. The leftmost two images are content and style images respectively. The rightmost two images are output images.

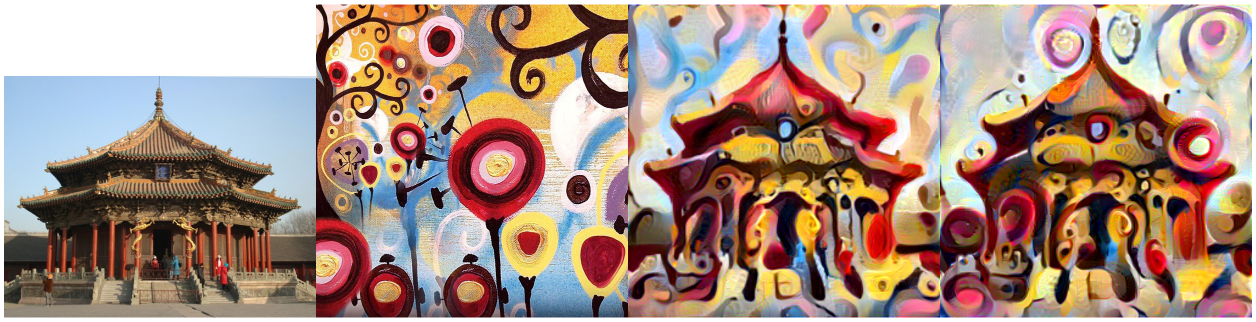


Figure 8: Style layers: Conv1_1, Conv1_2, Conv2_1, Conv2_2, Conv3_1, Conv3_2; Content layer: Conv2_1(third image from left); Style layers: Conv1_1, Conv2_1, Conv3_1, Conv4_1, Conv5_1; Content layer: Conv4_1(fourth image from left)

For the below images, $C_W = 1, S_W = 5000, Ite = 15$. The leftmost two images are content and style images respectively. The rightmost two images are output images.

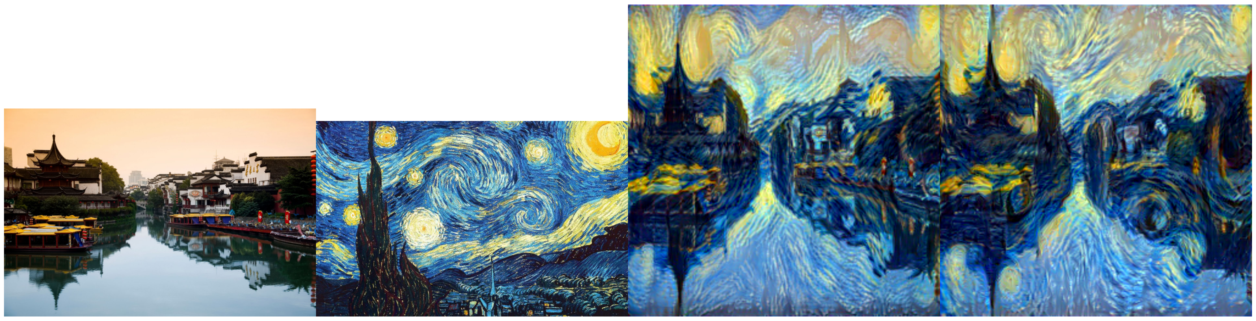


Figure 9: Style layers: Conv1_1, Conv1_2, Conv2_1, Conv2_2, Conv3_1, Conv3_2; Content layer: Conv2_2(third image from left); Style layers: Conv1_1, Conv2_1, Conv3_1, Conv4_1, Conv5_1; Content layer: Conv4_1(fourth image from left)

For the below images, $C_W = 1, S_W = 1000, Ite = 15$. The leftmost two images are content and style images respectively. The rightmost two images are output images.

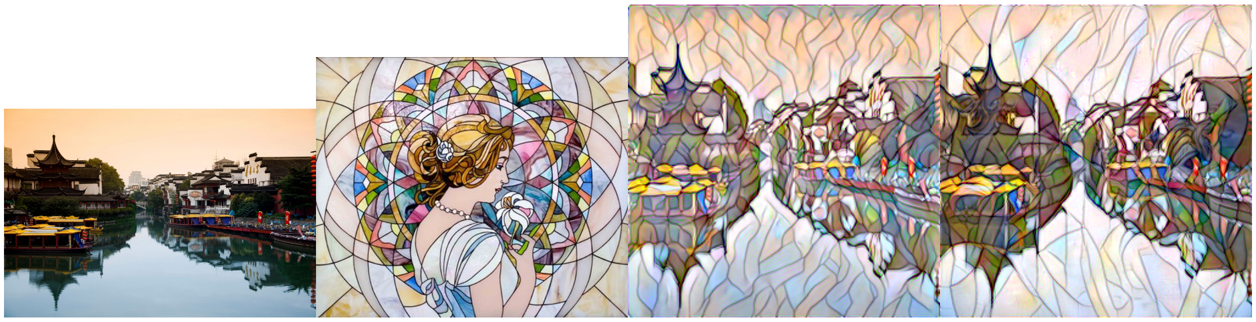


Figure 10: Style layers: Conv1_1, Conv1_2, Conv2_1, Conv2_2; Content layer: Conv2_1(third image from left); Style layers: Conv1_1, Conv2_1, Conv3_1, Conv4_1, Conv5_1; Content layer: Conv2_1(fourth image from left)

Another interesting issue we found out is that although a bad value (large value) of final total loss usually indicates a bad output image, the reverse is not true: a good value of final total loss does not necessarily indicate an artistically satisfying output image. Below is an example.

The parameters are: $C_W = 1, S_W = 5000, Ite = 15$. Style layers: Conv1_1, Conv1_2, Conv2_1, Conv2_2, Conv3_1, Conv3_2. Content layer: Conv2_2.

The final total loss is only 1.5873. However as we can see, the output image has a lot of random noise.

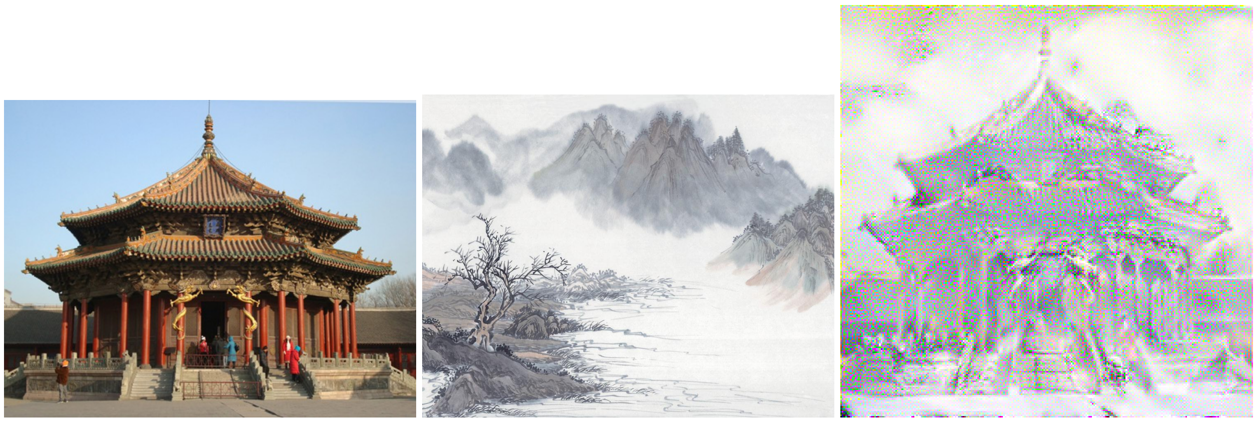


Figure 11: content, style, output

6.2. Perceptual losses for real-time style transfer and super-resolution

During training, this algorithm takes 80000 content images as training set and just one style image. The algorithm looks at this one style image repeatedly. Therefore, it has a tendency to overfit to the style image. For the images below, we used the starry night as style image and we set style weight to be 2000 and content weight to be 1. It is easy to see that the style is being overemphasized in the output. On page 11, we showed the results from first algorithm in which we set style weight to be 2000 and content weight to be 1, and the output images do not overemphasize style. For the neural network below, we trained for 40000 iterations with a batch size 4.

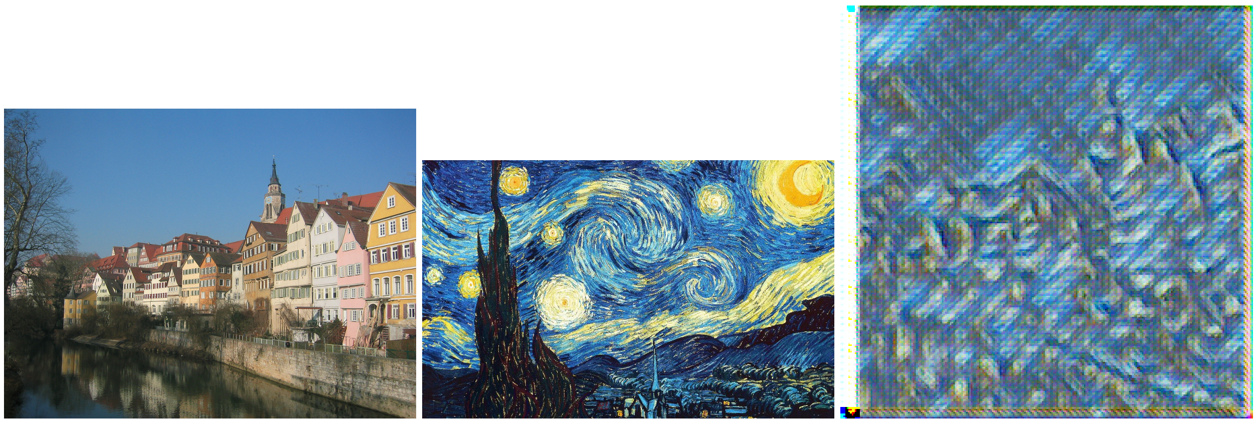


Figure 12: Content, Style, Output Image(left to right)

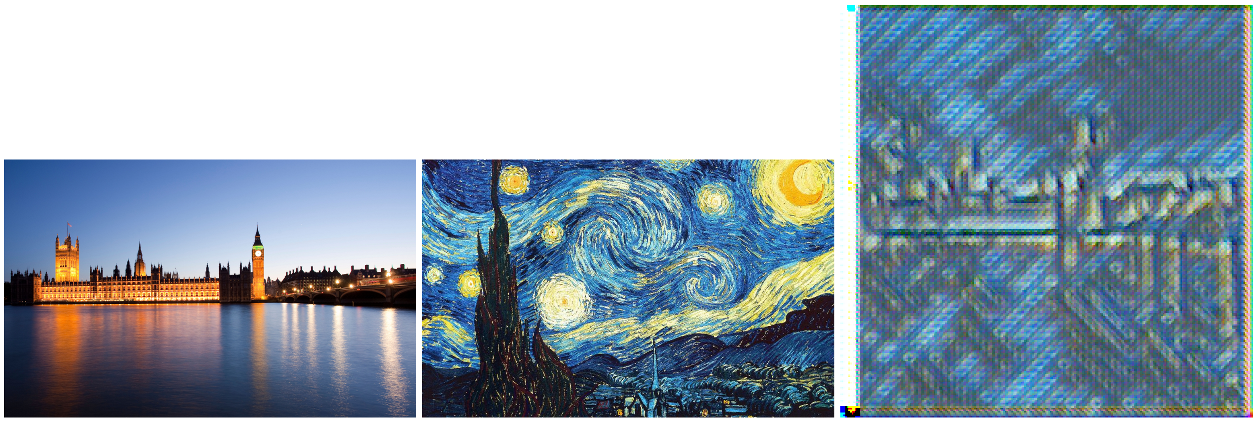


Figure 13: Content, Style, Output Image(left to right)

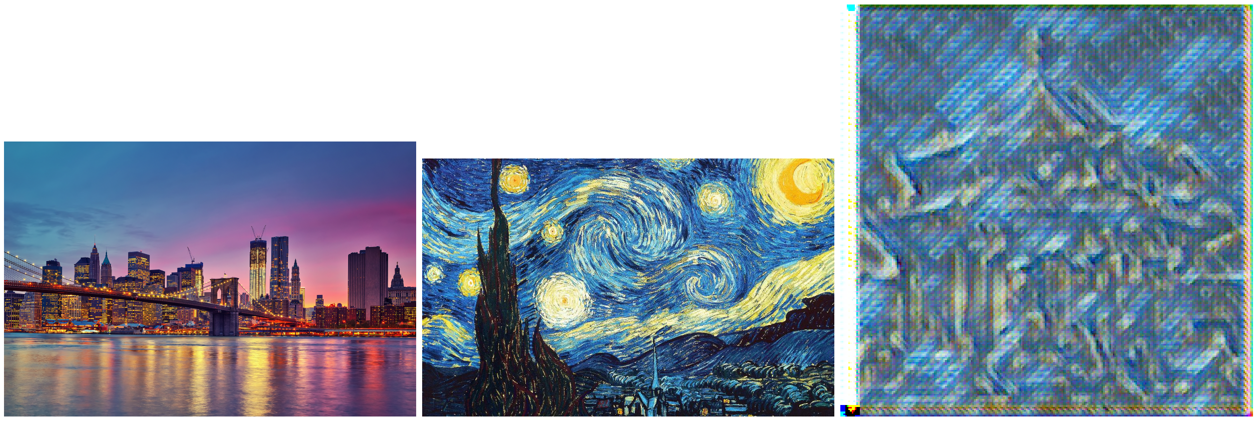


Figure 14: Content, Style, Output Image(left to right)

In the original paper, they trained for 40000 iterations with a batch size of 4. It takes about 14 hours on px2.large on AWS to do so. Since we have limited time and money, we trained for fewer iterations. We noticed that the loss function converges at aroung 10000 iterations. We will also show results from a model trained on the same style image for 10000, 11000, 12000, 13000 iterations with batch size of 4. The leftmost two images are content and style images respectively. The other images are the output images.

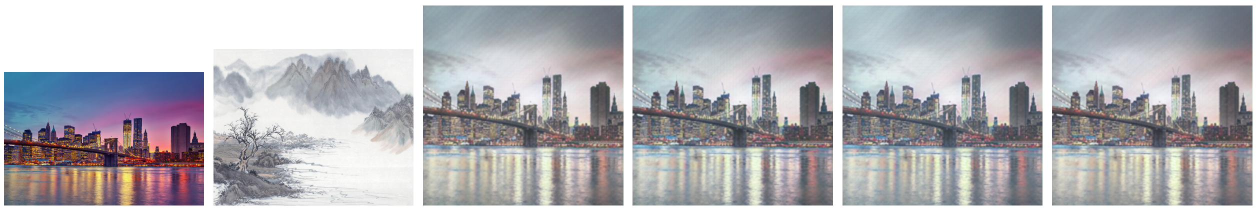


Figure 15: The output images are from models trained for 10000, 11000, 12000, 13000 iterations respectively(left to right)

From the above example, we saw that training for additional iterations do not lead to significant difference in output. Therefore, for future examples, we use models that are trained for 10000 iterations with a batch size of 4 in order to save time and money. We also used a style weight of 1000 and content weight of 1 in order to avoid overemphasizing style in the output. We trained one network for each unique style image displayed below. We used relu2_2 in VGG16 net to compute our content loss and we used relu1_2, relu2_2, relu3_3, and relu4_3 to calculate style loss(as in the original paper). We will now show some examples.

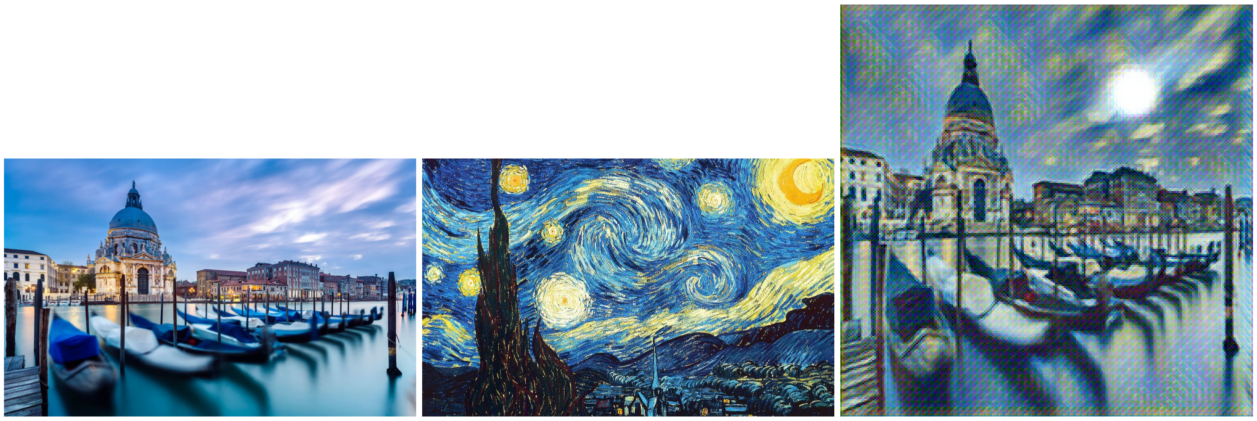


Figure 16: Content, Style, Output(left to right)

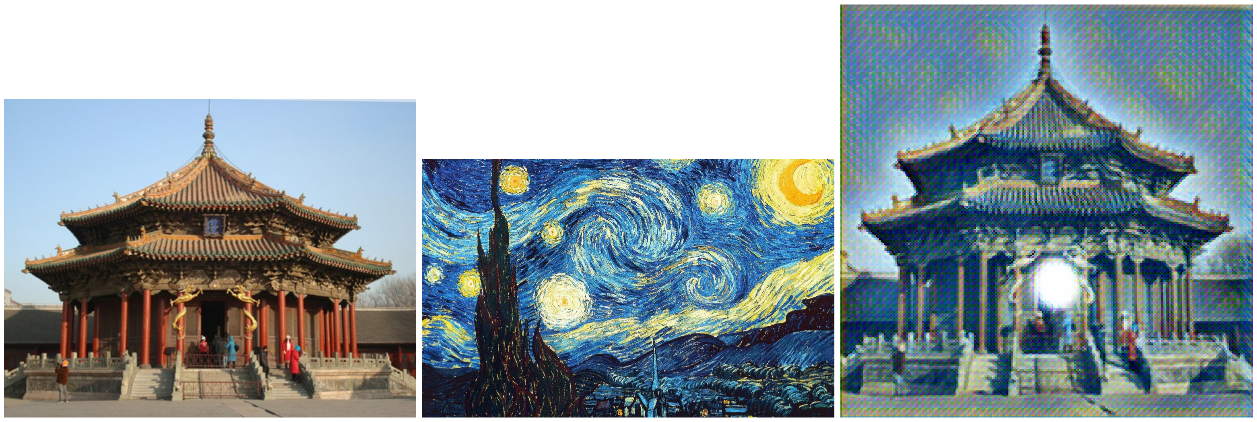


Figure 17: Content, Style, Output(left to right)

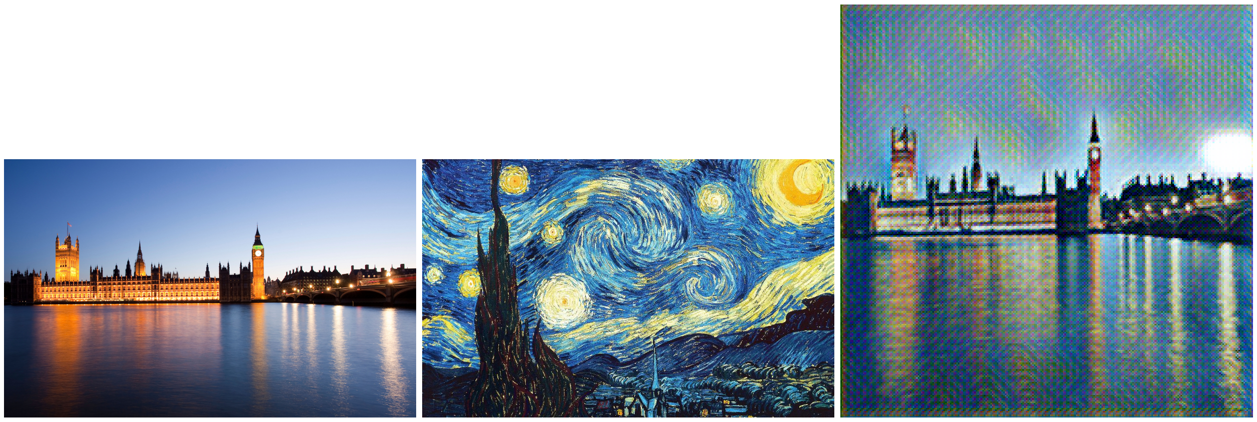


Figure 18: Content, Style, Output(left to right)

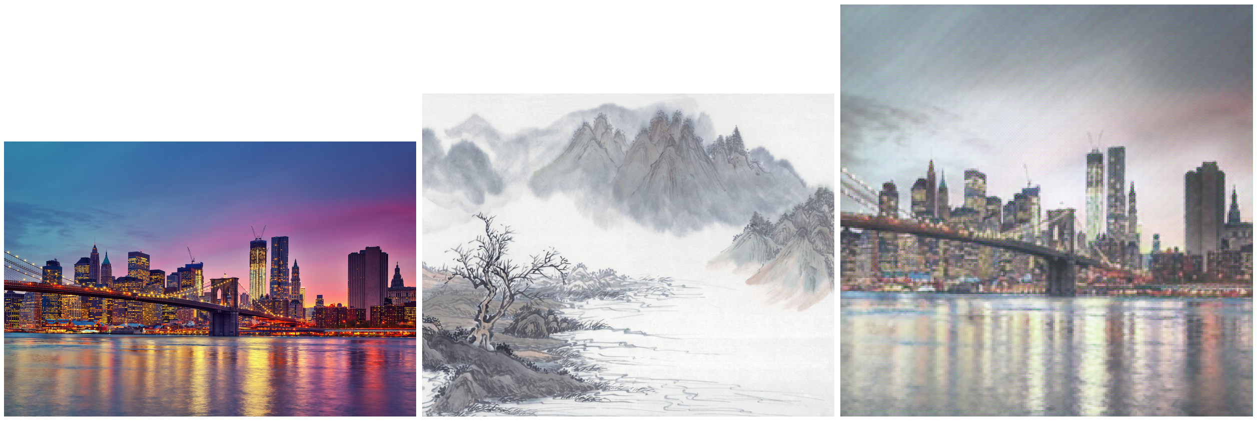


Figure 19: Content, Style, Output(left to right)

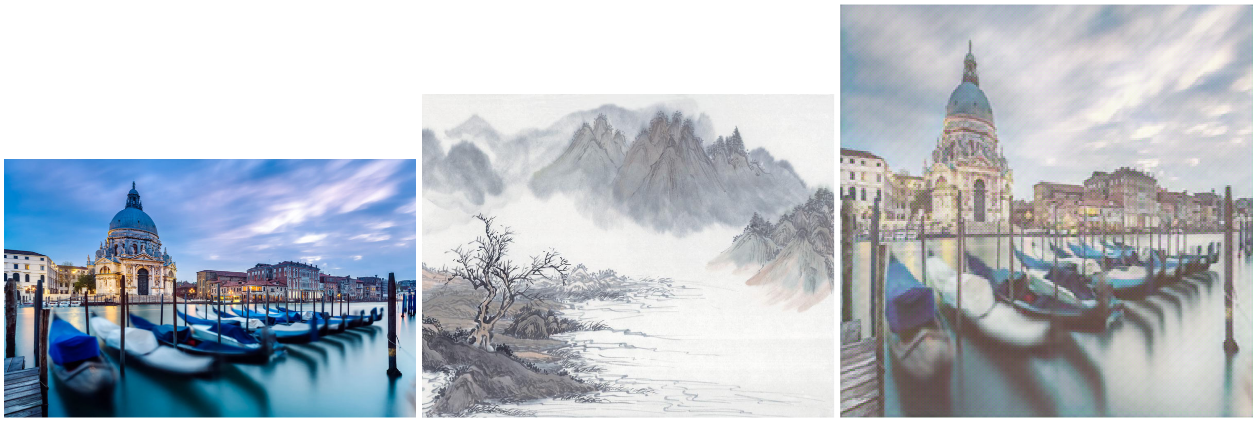


Figure 20: Content, Style, Output(left to right)

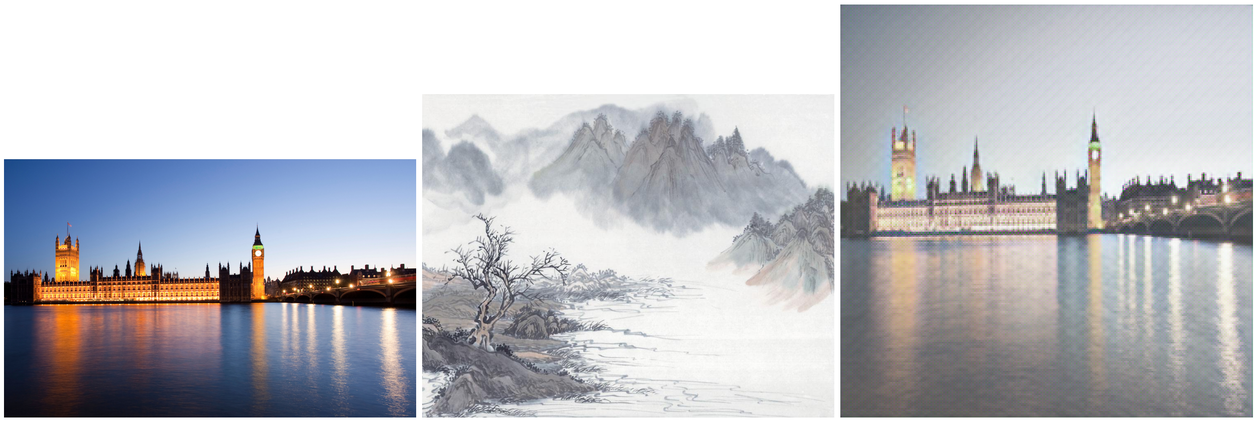


Figure 21: Content, Style, Output(left to right)

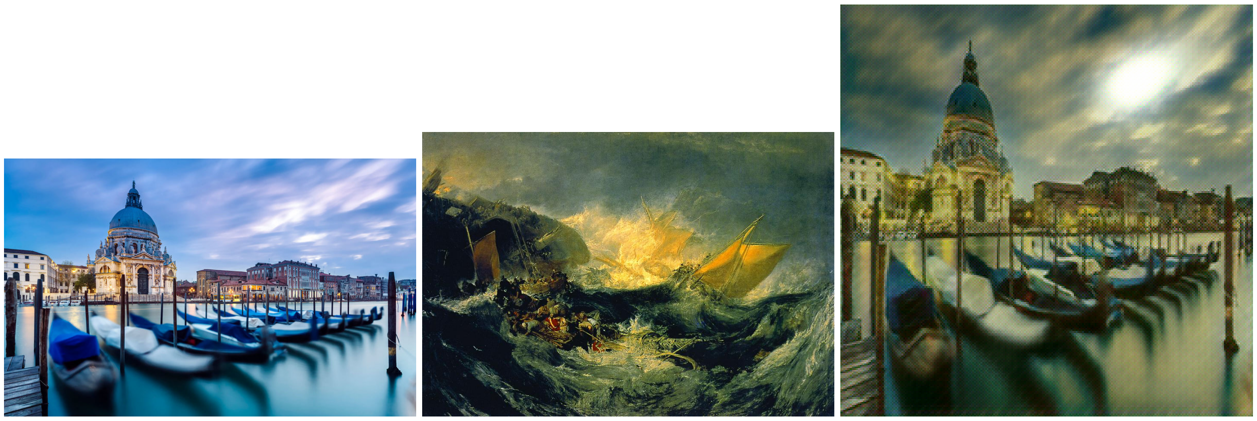


Figure 22: Content, Style, Output(left to right)

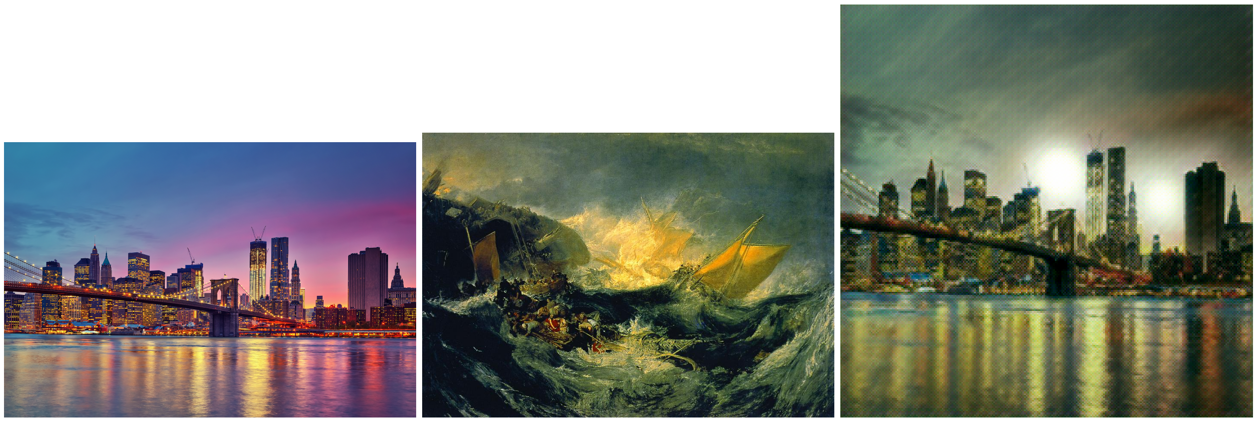


Figure 23: Content, Style, Output(left to right)

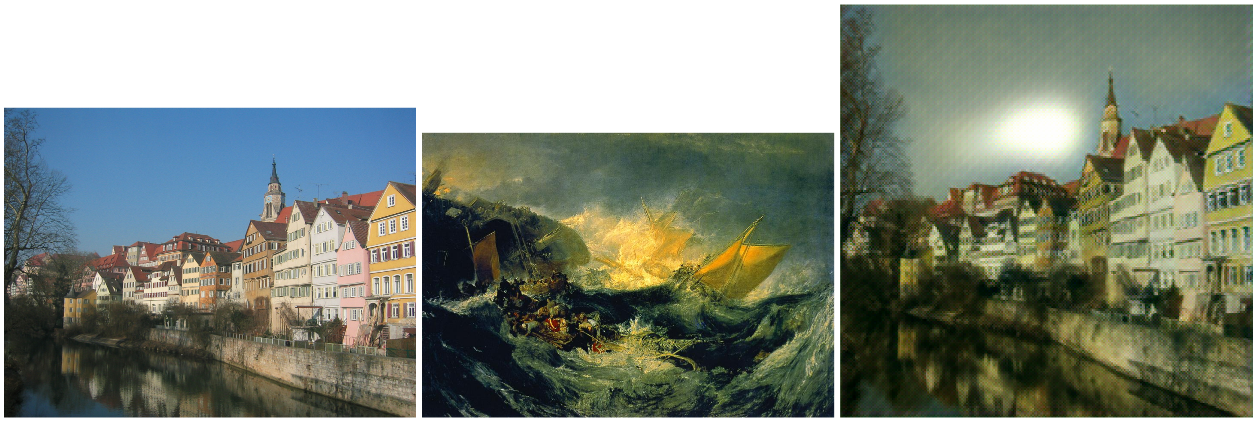


Figure 24: Content, Style, Output(left to right)

As you can see from the above examples, a single trained neural network tends to give output images that look very similar stylistically. If you want to make an adjustment, you have to retrain a network from the scratch, and this is a quite inconvenient feature.

6.3. Comparison between the Two Algorithms

Hyperparameter tuning for the real-time transfer algorithm is much harder than the iterative version because it takes a very long time to train the neural network from scratch, thus taking a very long time to try different sets of hyperparameters. If you want to transfer a few styles to many content images, the real-time transfer algorithm will be a better choice. However, if you want to transfer a variety of styles on a few content images, you should choose the iterative algorithm(3-5 minutes) since the time required to train the neural network in the real-time algorithm is very long(14 hours using p2.xlarge on aws).

7. Quantatitive Analysis

7.1. A neural algorithm of artistic style

One interesting aspect of this algorithm is the usage of pooling layers in VGG19 net. If we use maxpooling, which is the pooling layer used in VGG19 net, the loss function converges aroung 100 iterations. If we replace maxpooling with average pooling(suggested in the original paper) however, the algorithm will converge in around 15-30 iterations.

For all our results, we used average pooling because of the faster training time(time is very valuable as analysis with different hyparameters takes a lot of time). For the version with average pooling, it is risky to train for too many iterations since the loss function might diverge. Here is an example in which we trained for 50 iterations.

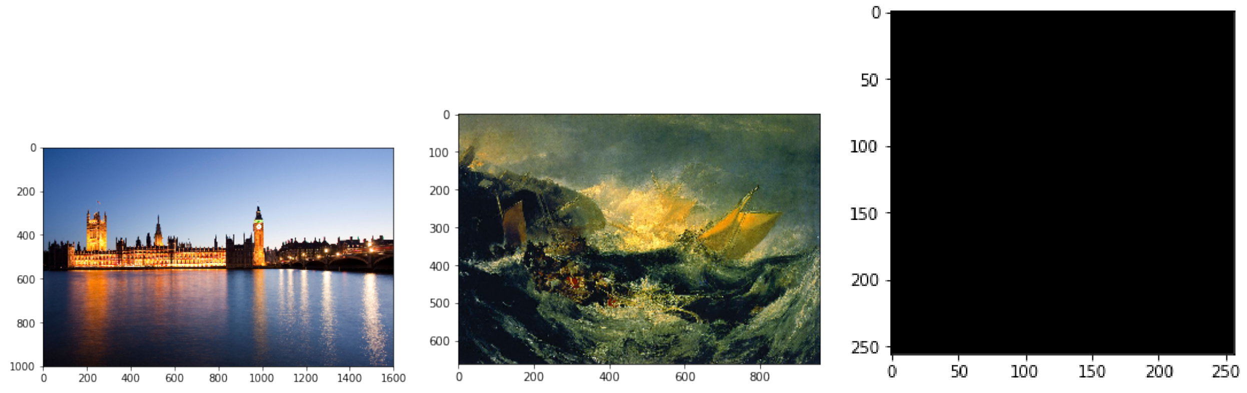


Figure 25: content,style,output

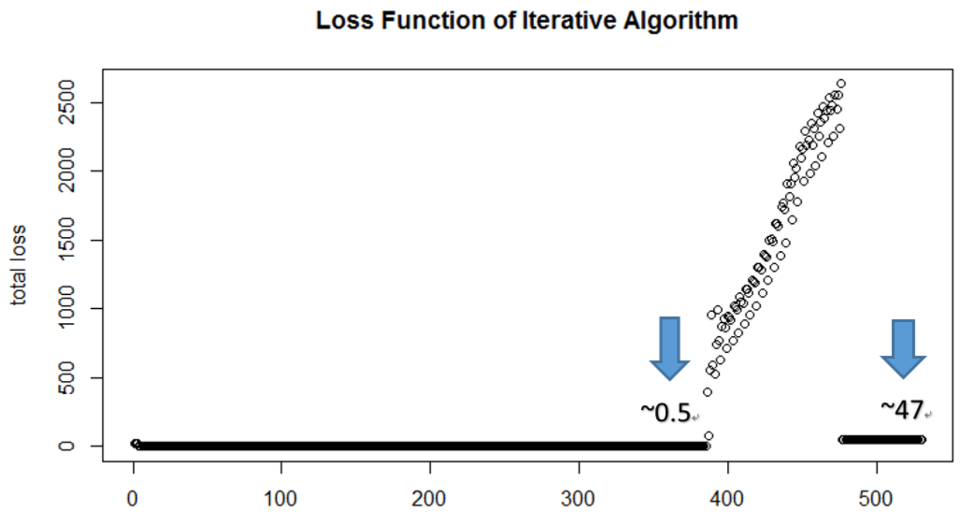


Figure 26: Loss Function

As you can see, the loss function never converges to an optimal value. Instead, it stays at a very high level, leading to a very bad result. The x-axis does not mean that we ran for more than 500 iterations. The LBFGS optimizer we used in our implementation does many rounds of optimization for every step it takes. Although this optimizer gives very good result, it does not seem to be very stable.

7.2. Comparison between the Two Algorithms

We surveyed 20 classmates. We gave each of them 8 images that are output from algorithm 1 and 8 images that are output from algorithm 2. For each of the following two rows of images, we applied the style of the leftmost image to the other 4 images on the same row. In this way, algorithm 1 and algorithm 2 will generate 8 images respectively.

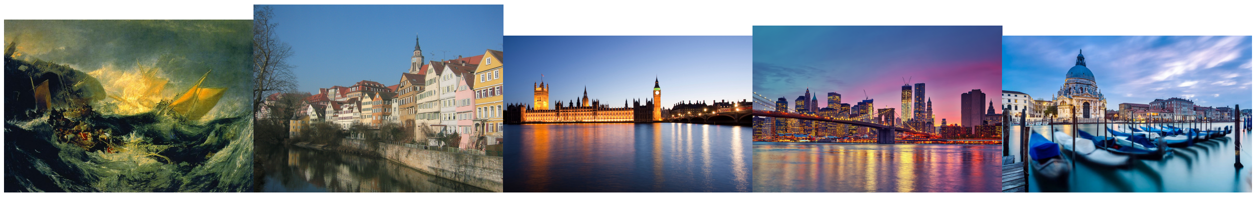


Figure 27: one style and four content images

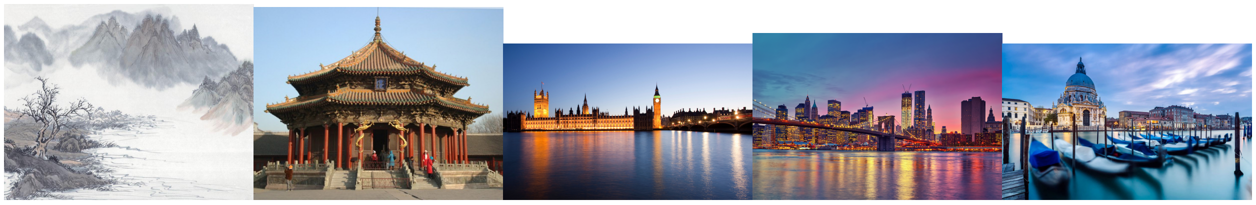
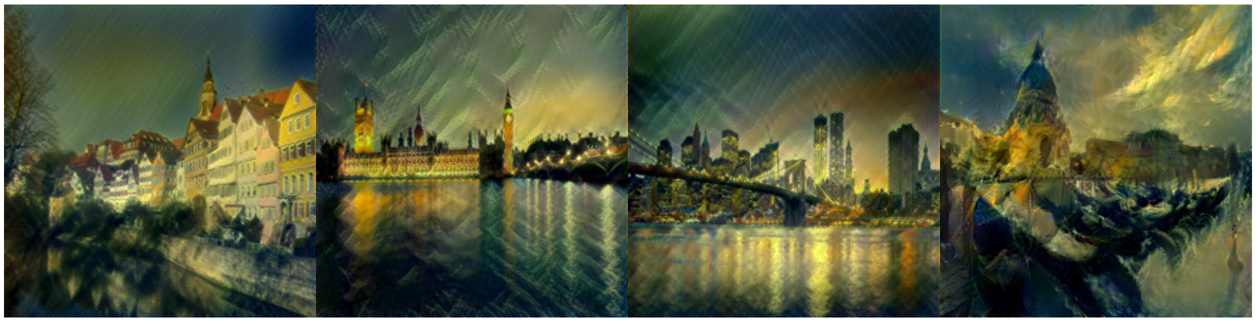
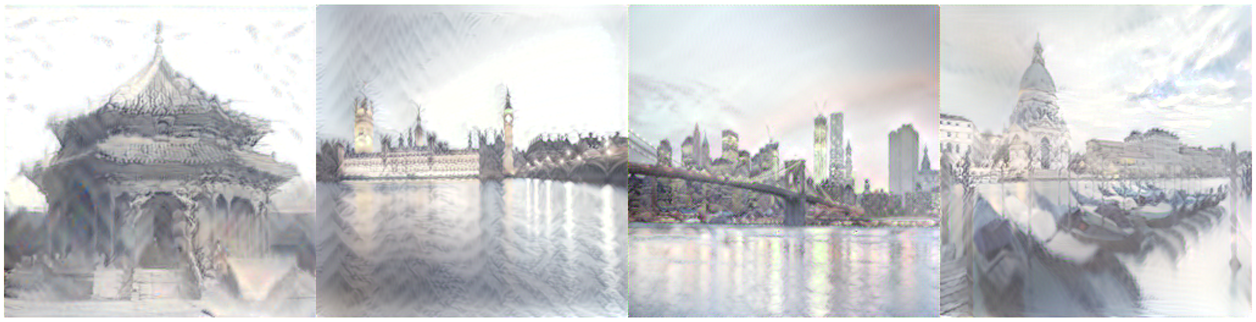


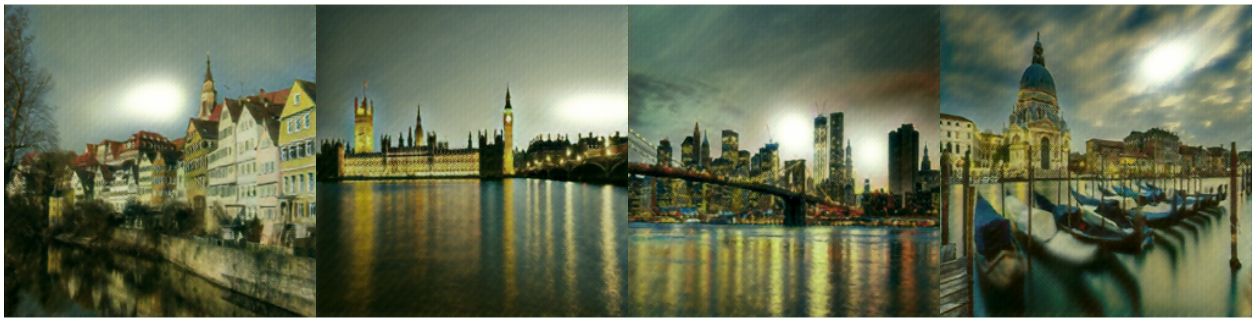
Figure 28: one style and four content images

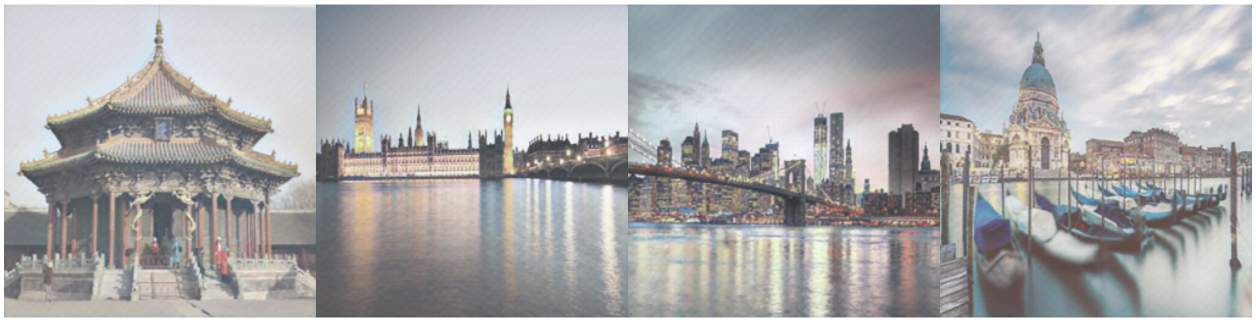
Here are the 8 output images generated by algorithm 1:





Here are the 8 output images generated by algorithm 2:





We conducted the survey in the following way: We put the output images generated from the same pair of content image and style image by algorithm 1 and 2 side by side on a piece of paper and present it to the user. We present 4 pieces of paper to a single user in total(with two images on each piece of paper). We ask the use to pick which image they like better. We randomize the placement of the images(sometimes we place images from algorithm 1 on the left, sometimes we place images from algorithm 1 on the right). Each one of the 20 users voted 4 times, giving us a total of 80 votes. Overall, The images from algorithm 1 get a total of 59 votes, and images from algorithm 2 get a total of 21 votes. Although in our case, most users prefer the images generated from algorithm 1, we should take this result with a grain of salt. We did not have enough time to find the most optimal parameters for algorithm 2, due to the long time required for training. Given enough time and computational resources, we think we should be able to find hyperparameters that will enable algorithm 2 to generate images at least similar in quality to algorithm 1. For example, between the two images below, the output image from algorithm 2 actually got 13 votes and the output from algoritm 1 only got 7 votes.

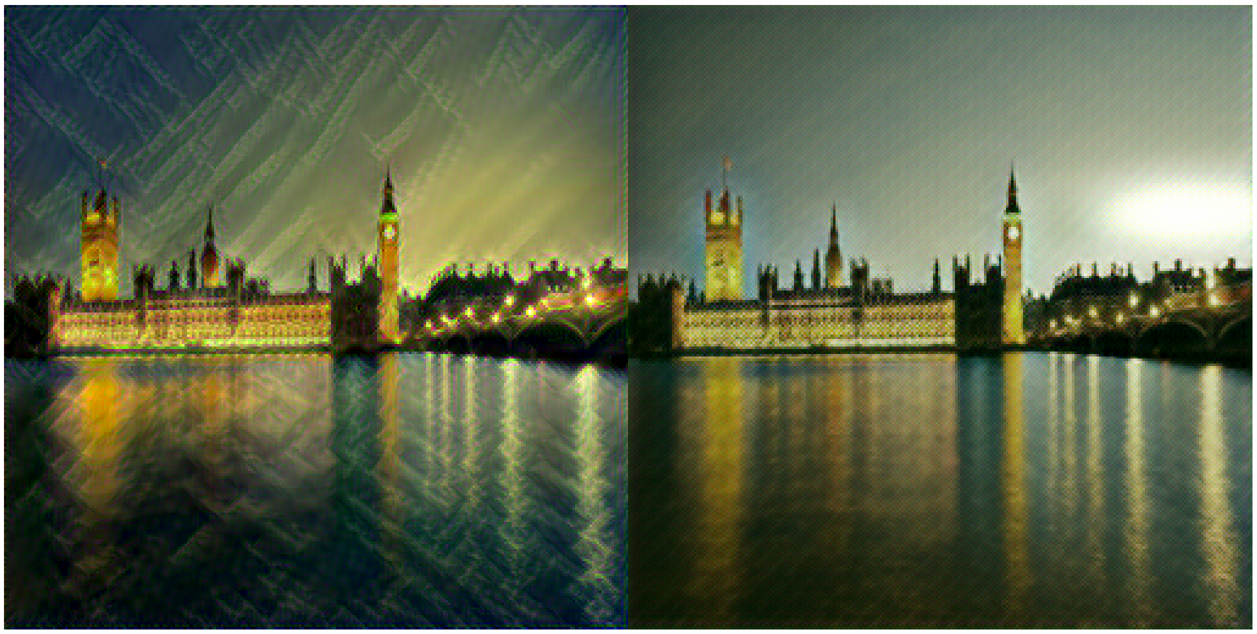


Figure 29: Algorithm 1(left), Algoritm 2(right)

8. Future Work

We would like to explore the algorithm that can do style transfer for a variety of styles with a single neural network in the future and see whether we can improve on that algorithm.

9. Information Regarding Code

The file "Neural Style Transfer.ipynb" is the implementation of algorithm 1(the iterative algorithm). The file "Real Time Neural Style Transfer.ipynb" is the implementation of algorithm 2(the real-time algorithm). In case you have trouble opening Ipython notebook files, we also included "Neural Style Transfer.py" and "Real Time Neural Style Transfer.py". We ran our algorithm on the Deep Learning AMI (Ubuntu) on AWS. We used the EC2 p2.xlarge instance. For detailed instruction regarding how to run our code, please contact us directly(It mainly involves opening up the correct AWS instance, and upload files to EC2, and get the file path correct). We did not include the detailed instruction here since both Professor Russakovsky and Riley told us that it is not very necessary to do so as it is unlikely that there is a need to rerun our code.

10. Honor Code Pledge

I pledge my honor that this project represents my own work in accordance with University regulations.

Signature: Haochen Li, Yuyan Zhao

References

[1] V. Dumoulin, J. Shlens, and M. Kudlur, “A learned representation for artistic style,” *ICLR*, 2017. [Online]. Available: https://arxiv.org/abs/1610.07629

[2] L. A. Gatys, A. S. Ecker, and M. Bethge, “A neural algorithm of artistic style,” *CoRR*, vol. abs/1508.06576, 2015. [Online]. Available: http://arxiv.org/abs/1508.06576

[3] J. Johnson, A. Alahi, and L. Fei-Fei, “Perceptual losses for real-time style transfer and super-resolution,” in *European Conference on Computer Vision*, 2016.

[4] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár, “Microsoft COCO: Common Objects in Context,” *Computer Vision–ECCV 2014. Springer (2014).*