Neural Style Transfer Implementation and Applications

Final Project Report of GR5242

Prepared by Hongyu Li (hl3099), Jia Zheng (jz2891),

Peifeng Hong (ph2534) and Di Wu (dw2794)

Contents

1 Introduction

1.1 Background

Creativity is something we closely associate with what it means to be human and a lot of artists are pride of using skills to compose painting or arts. But with the development of digital technology, it seems that machines could also be creative and compose arts like humans by learning. Neural style transfer is one of the examples.

Neural style transfer is a deep learning technique used to combine the artistic style of one image with the content of another image. It was first described by Leon A. Gatys in 2015[1]. Based on this idea, a lot of applications has been released. One of the coolest applications you might know is Prisma which is an app that allows you to transform your photos in the style of other with the help of convolutional neural network. After that, much progress also has been made on style transfer to make style transfer training and inference faster[2] [3]and to extend the style transfer technique from static images to videos and even audio and other mediums.

1.2 Related Work

In 2015, Gatys et al. [1] for the first time proposed the combination of content loss and style loss based on a pretrained convolutional network. Their methods produced high-quality results, but is computational expensive due to hundreds of gradient descent iterations.

To overcome this computational burden, Johnson et al. [2] in 2016 train a feed-forward image transformation network to quickly approximate solutions to their optimization problem. However, for each new style, it needs to train a corresponding image transformation network, which is time consuming and the style is encoded in the network parameters by enormous iterations of stochastic gradient descent.

Shen et al. [3] in 2017 built a meta network which takes in any given style image and directly produces a corresponding image transformations network, which is real time efficient.

1.3 Our Goals

In this project, we have three goals. Firstly, we want to implement two style transfer algorithms: the original 1 to 1 neural style transfer algorithm (fixed style for a certain image) that was came up with by Gatys[1] and fast neural style transfer algorithm (fixed style for arbitrary image) that was proposed by Johnson[2]. Secondly, we want to apply the fast algorithm in real-time off a webcam. Lastly, we are also planning to build an online demo in browser so that users could upload their own images and design their own styled photos. By doing these, we are trying to get a closer look at style transfer algorithms with TensorFlow eager execution and bring more possibilities for this area based on our understanding.

2 Methods

The basic idea of neural style transfer is to take the feature representations learned by a pre-trained deep convolutional neural network (which is known as **transfer learning**) to obtain separate representations for the style and content of any image. Once these representations are found, we can then try to **minimize the loss** that combines the loss of content and style.

2.1 Transfer Learning

Transfer learning is a machine learning method where a model developed for a task is reused as the starting point for a model on a second task. This learning method assumes that if this original dataset is large enough and general enough, then the spatial feature hierarchy learned by the pre-trained network can effectively act as a generic model of our visual world, and hence its features can prove useful for many different computer vision problems, even though these new problems might involve completely different classes from those of the original task[4].

It is currently very popular in the field of deep learning because it allows us to build accurate models in a timesaving way[5]. What's more, we don't need a lot of data in this case. It's just like the deep learning version of Chartres' expression '*standing on the shoulder of giants*'. Due to its massive advantages, Andrew NG even think that it will be -- after supervised learning -- the next driver of ML commercial success[6].

2.2 Artistic Loss

The key idea of style transfer is to decompose the loss into three parts: content loss, style loss and variation loss [1].

$$L_{\text{total}} = \alpha L_{\text{content}} + \beta L_{\text{style}} + \gamma L_{\text{variation}}$$

where α, β and γ are weights for content loss, style loss and variation loss.

2.2.1 Content Loss

The content loss is the Euclidean distance between feature representations of the content and combination image. This is defined as below.

$$L_{\text{content}} = \sum_{i,j} \left(\mathrm{F}_{\mathrm{ij}}^{\mathrm{l}} - \mathrm{C}_{\mathrm{ij}}^{l}\right)^2$$

where F_{ij} is the input image's feature representations in layer *l* and C_{ij} is the content image's feature representations in layer *l*. This formula is a little bit different from Gatys' paper because we found that the normalization term (0.5 used in the paper) do not have much impact on the performance of the algorithm in our implementations.

2.2.2 Style Loss

Style loss is the Euclidean distance between gram matrix of the style image and combination image. Gram matrix is the inner product of feature maps which captures information about which features tend to activate together. By only capturing these aggregate statistics across the image, they are blind to the specific arrangement of objects inside the image. This is what allows them to capture information about style independent of content[7]. The style loss and gram matrix are defined as below.

$$\mathrm{G}_{\mathrm{ij}}^{\mathrm{l}} = \sum_{k} (F_{ik}^{l} F_{kj}^{l})$$

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} \left(\mathrm{G}_{\mathrm{ij}}^{\mathrm{l}} - \mathrm{S}_{\mathrm{ij}}^{l}\right)^2$$

$$\mathrm{L}_{\mathrm{style}} = \sum_{l} \mathrm{w}_{\mathrm{l}} E_l$$

where $\mathrm{G}_{\mathrm{ij}}^{\mathrm{l}}$ stands for the gram matrix of combination image in layer *l*, $\mathrm{S}_{\mathrm{ij}}^{\mathrm{l}}$ stands for the gram matrix of style image in layer *l*, N_{l} stands for the number of color channels, M_{l} stands for the multiply of width and height and w_{l} stands for the style weights for each layer. This definition is given by Gatys[1]. Johnson's definition of style loss[2] is a little bit different in the normalization term. However, based on our experimentations, we thought normalization terms do not influence the results greatly.

2.2.3 Variation Loss

Variation loss is a regularization term that encourages spatial smoothness. [2] This loss is defined as below.

$$\mathrm{L}_{\mathrm{variation}} = \sum_{i,j} \left(x_{i,j+1} - x_{ij}\right)^2 + \left(x_{i+1,j} - x_{ij}\right)^2$$

where x_{ij} stands for the combination image.

2.3 Implementation Details

We implemented our code by using **TensorFlow eager execution** which is an imperative programming environment that evaluates operations immediately, without building graphs. Eager execution would be default in TensorFlow 2.0, so we use this mode to implement our code. As for the implementation of one-to-one neural style transfer algorithm, we trained our model on Colab. However, as for the implementation of faster neural style transfer algorithm, we trained our model by using GCP due to the computation complexity.

3 Implementation and Results

3.1 1 to 1 Neural Style Transfer

As mentioned before, the basic idea of 1 to 1 neural style transfer is to combine transfer learning with optimization. Our workflow to implement it has been shown in Fig 3-1.

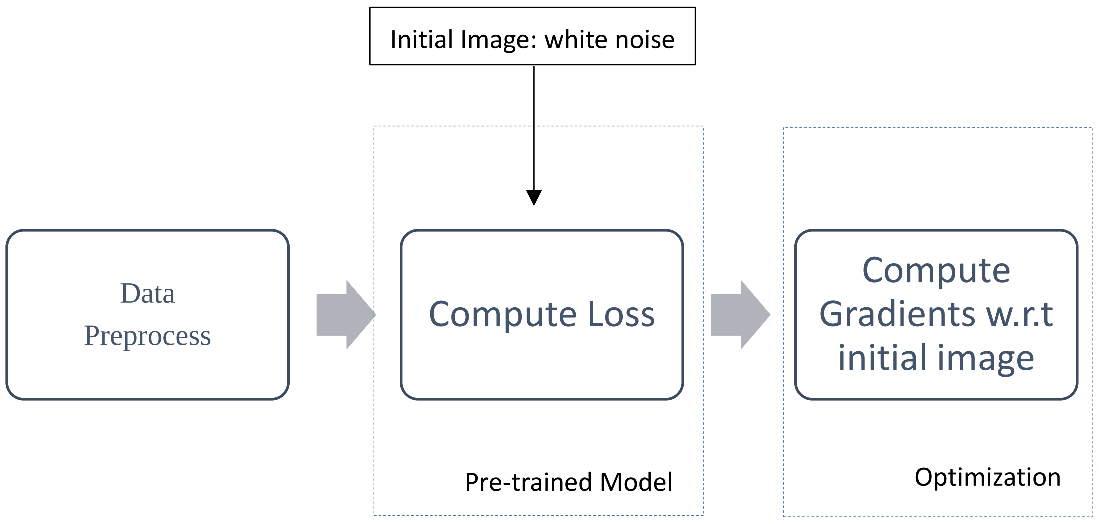

Initial Image: white noise
Data Preprocess
Compute Loss
Compute Gradients w.r.t initial image
Pre-trained Model
Optimization

Fig 3-1: Workflow of Implementation

3.1.1 Network Architecture

After specifying our style transfer model, we are going to choose a pre-trained neural network for our further implementation. In our implementation, we would use VGG net to perform transfer learning. However, VGG net is only used to assist in our calculation of loss function. Thus we won't use the whole network. Instead, we only use the top few layers to help us extract features from an image.

VGG net was designed by Simonyan and Zisserman in 2014[8]. This network was trained on ImageNet dataset and characterized by its simplicity, using only 3*3 convolutional layers stacked on top of each other in increasing depth. Reducing volume size is handled by max pooling. Two fully-connected layers, each with 4,096 nodes are then followed by a softmax classifier. There are two widely used version of VGG net: VGG-16 and VGG-19.

In our implementation, we would use VGG-16 net rather than VGG-19 used in the original paper because some researches showed that VGG-16 could be as good as VGG-

19 for style transfer task but VGG-16 is simpler than VGG-19[2]. VGG-16 has 16 weight layers in the network. The visualized structure of VGG-16 is shown in Fig 3-2.

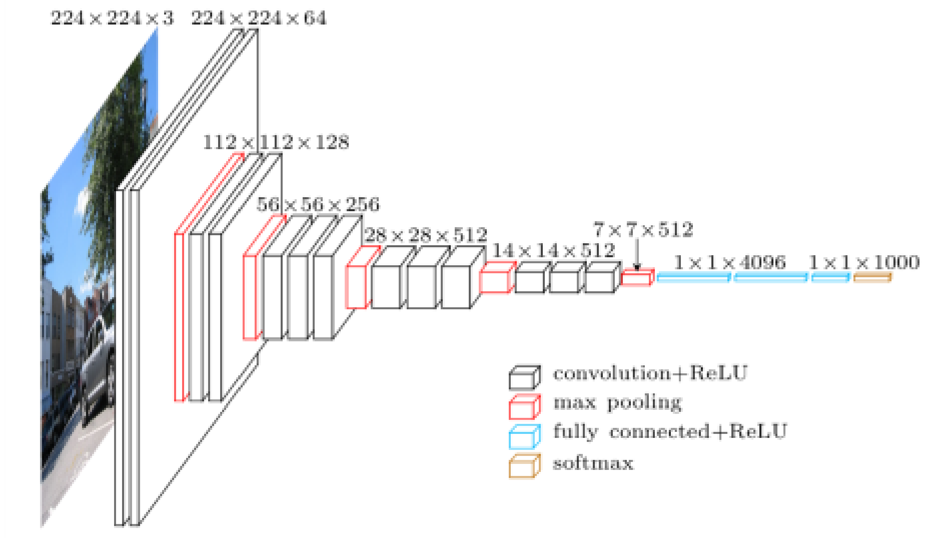


Fig 3-2: VGG16 Architecture

3.1.2 Dataset and Preprocessing

Dataset. For this implementation, we only used two images: *central park.jpg from CNN.com (content)* and *starry night.jpg from Vincent van Gogh, 1889 (style)*.

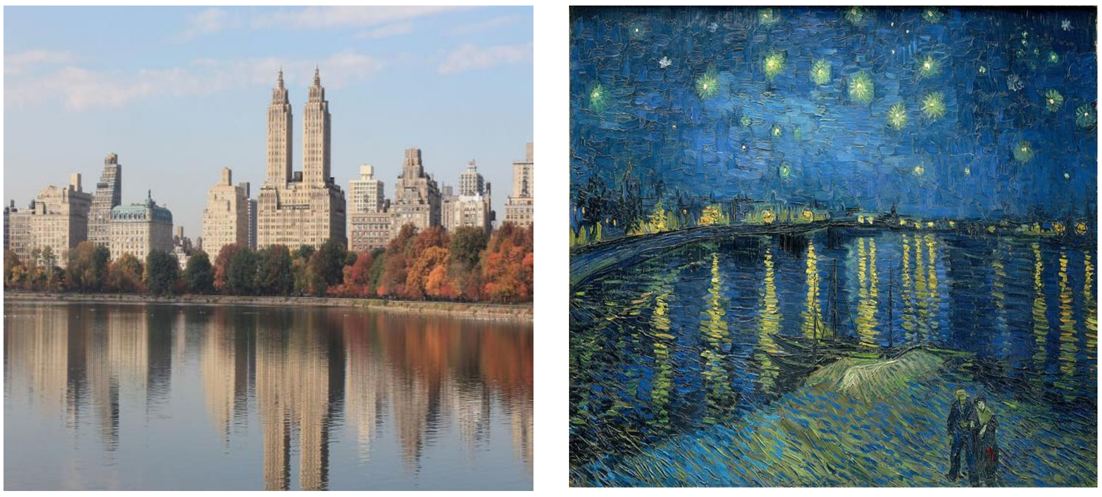


Fig 3-3: Content Image (Left) and Style Image (Right)

Data Preprocessing. As stated in 2.1, transfer learning has many advantages, but we should be very careful about some details while using pre-trained model, especially in data preprocessing. That’s because most of pre-trained models are trained on ImageNet dataset and their data preprocessing methods are different. For example, inputs for VGG-16 should be subtracted the mean RGB value so as to somehow adjust the color difference between content image and style image while inputs for ResNet should be rescaled into 0 to 1. Inputting accurate data into pre-trained model has a great impact on the performance of transfer learning which was found during our implementation. Thus, we resized our images into 512*512 and also subtracted the mean RGB value ([103.939, 116.779, 123.68] is used in paper) before inputting these two images into the VGG-16.

3.1.3 Parameters Tuning

In our implementation, we came across a couple of hyper parameters that need tuning process to decide the best value.

The first thing we have to decide is that which layer in the VGG-16 net we should choose to calculate our content and style loss. To visualize the outcome of each layer, we set style/content weight to 0 to see the effect of content/style reconstruction.

Content Reconstruction. We set the weights of style loss and variation loss to 0 to see the effect of each layer. From Fig 3-4, we can see that as the layer goes deeper, the color and context of the image are decomposed and the spatial structure remains.

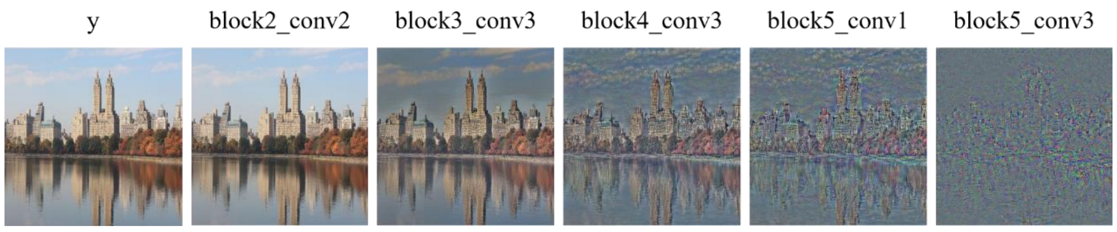


Fig 3-4: Content Reconstruction

Style Reconstruction. We set the weights of content loss and variation loss to 0 to reconstruct the style feature from each layer. Different from content reconstruction, in style reconstruction we sum up the style loss of the current layer and all the former layer. For example, the style reconstruction of block2_conv2 is calculated by summing up the style loss of block1_conv2 and block2_conv2. Fig 3-5 shows that as the layer goes deeper, the spatial structure of our style image is decomposed and the color and texture style of the style image remain.

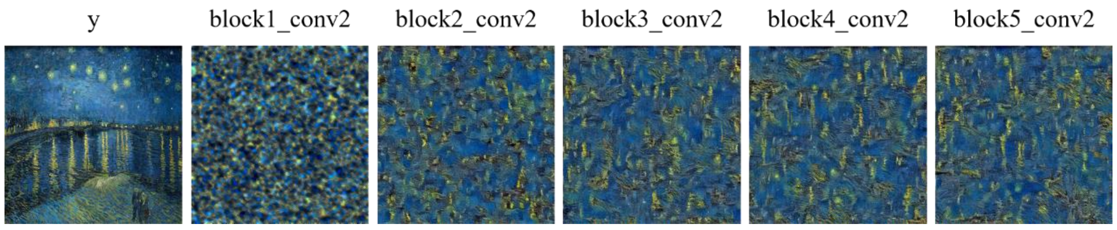


Fig 3-5: Style Reconstruction

Content and style reconstruction demonstrates the function of content loss and style loss. Thus we can step into the next step: decide which layer we would use in our training process. Fig 3-6 shows that the content structure barely exists in the combination image when the content layer goes as deep as block4_conv3. After comparing these 25 images, we thought the image at the left bottom corner has the best visual effect. So we decided to use block2_conv2 as content layer and block1_conv2, block2_conv2, block3_conv2, block4_conv2 and block5_conv2 as style layers. This is different from the paper of Gatys which chose block4_conv2 as content layer and block1_conv1, block2_conv1, block3_conv1, block4_conv1 and block5_conv1 as style layers[1].

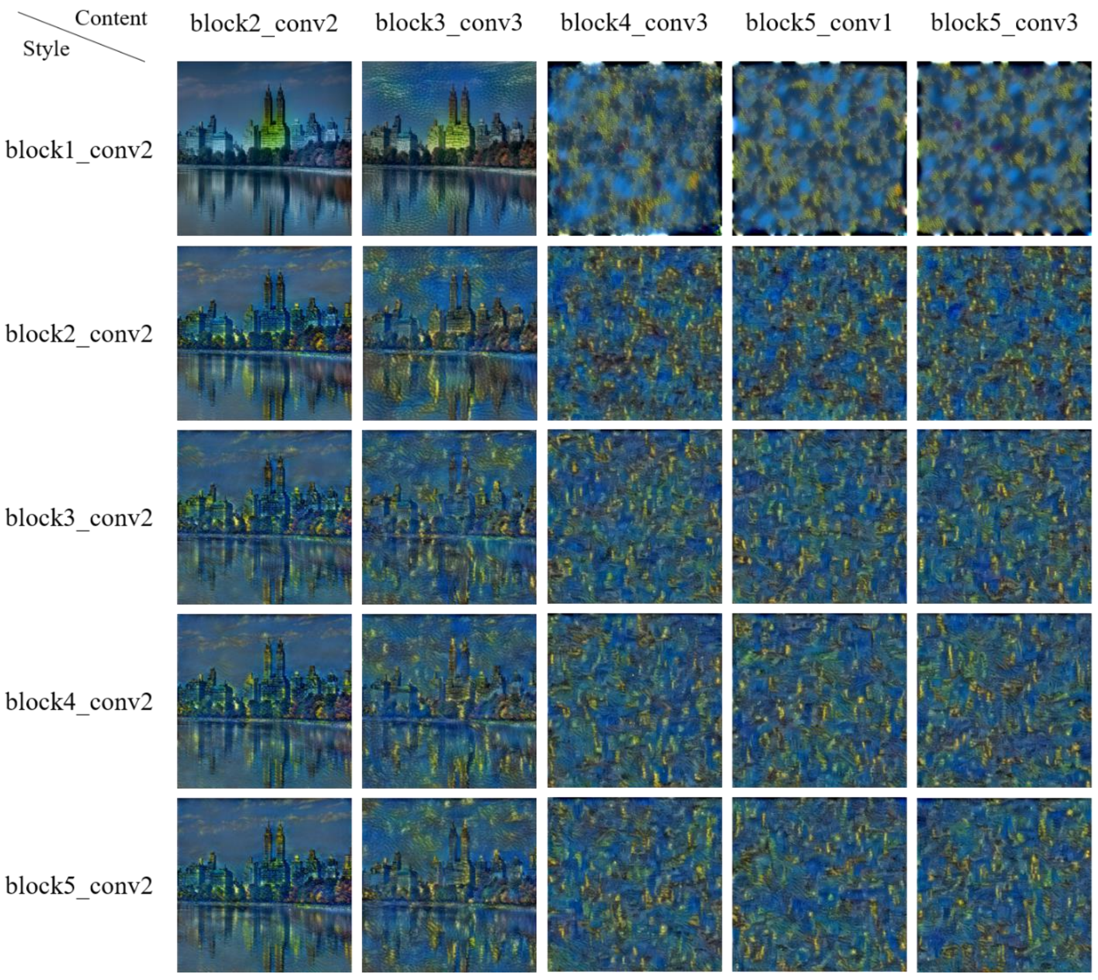


Fig 3-6: Representation Outputs

Weights. The next parameter we need to decide is the combination of content weight, style weight and variation weight. Before we start the tuning process, one thing to clarify is that we don't need to decide the actual value of these weights. Instead, we only need to decide the ratio between different weights. Thus, we fixed the value of style weight to 400 and tune the weights of content and variation loss. Fig 3-7 shows the result of different combination of weights. As the content weight increases, the combination image converges to content image. As the variation weight increases, the image becomes blurry. We chose two combinations: 0.01-400-1 and 0.005-400-1. The former one has higher content weight thus the style feature is less conspicuous.

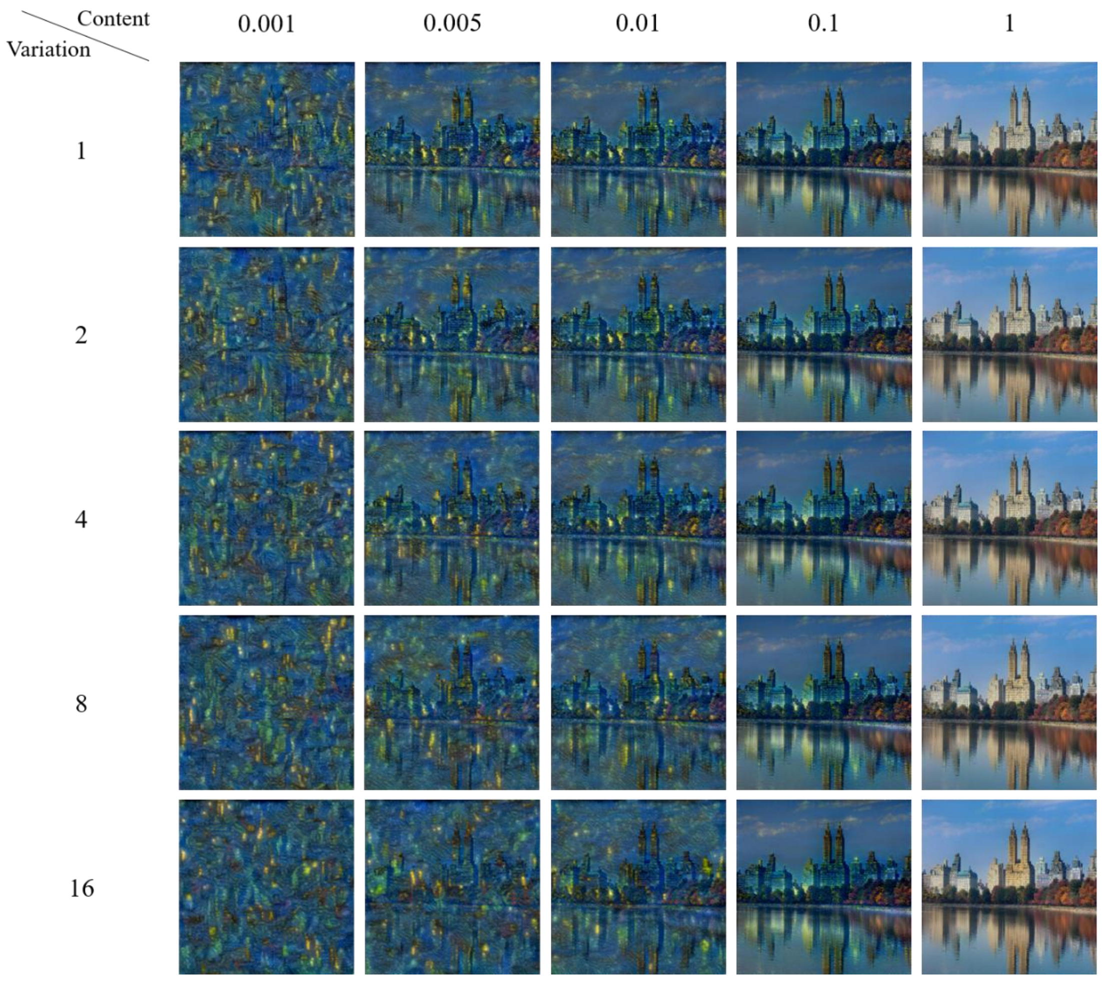


Fig 3-7: Weights Tuning Outputs

Optimizer. The original paper used gradient decent optimizer to minimize the total loss which is slow to converge and easy to be stuck in local minimum. Therefore, we changed the optimizer to be L-BFGS in our implementation. L-BFGS is a quasi-Newton algorithm that's significantly quicker to converge than standard gradient descent[9]. L-BFGS speeded up the convergence of loss.

To sum up, by comparing outputs of different combinations of parameters, the parameters that we used in our implementation is shown in Table 3-1.

Table 3-1: Pre-defined Parameters for 1 to 1 Neural Style Transfer

Parameter Type	Parameter Value
Content Weights	0.01
Style Weights	400
Variation Weights	1
Beta for Variation Loss	2.5
The number of training epochs	10
Optimizer	L-BFGS
Content Layer	block2_conv2
Style Layers	block1_conv2, block2_conv2, block3_conv2, block4_conv2 and block5_conv2

3.1.4 Results

Our starry central park is shown in Fig 3-8. It successfully transfer the style of starry night into the image of central park. We set the iterations to be 10 and the total training time is 179s.

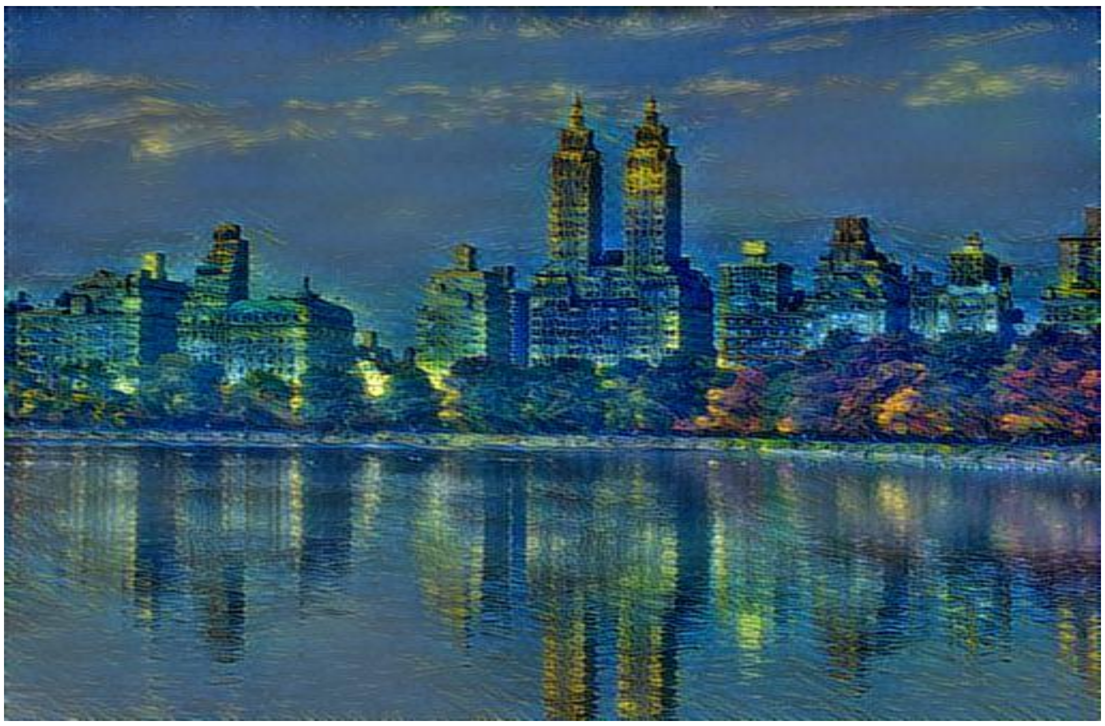


Fig 3-8: Starry Central Park

The process of training is shown in Fig 3-9. By looking at this, we could see how our model worked in each iteration.

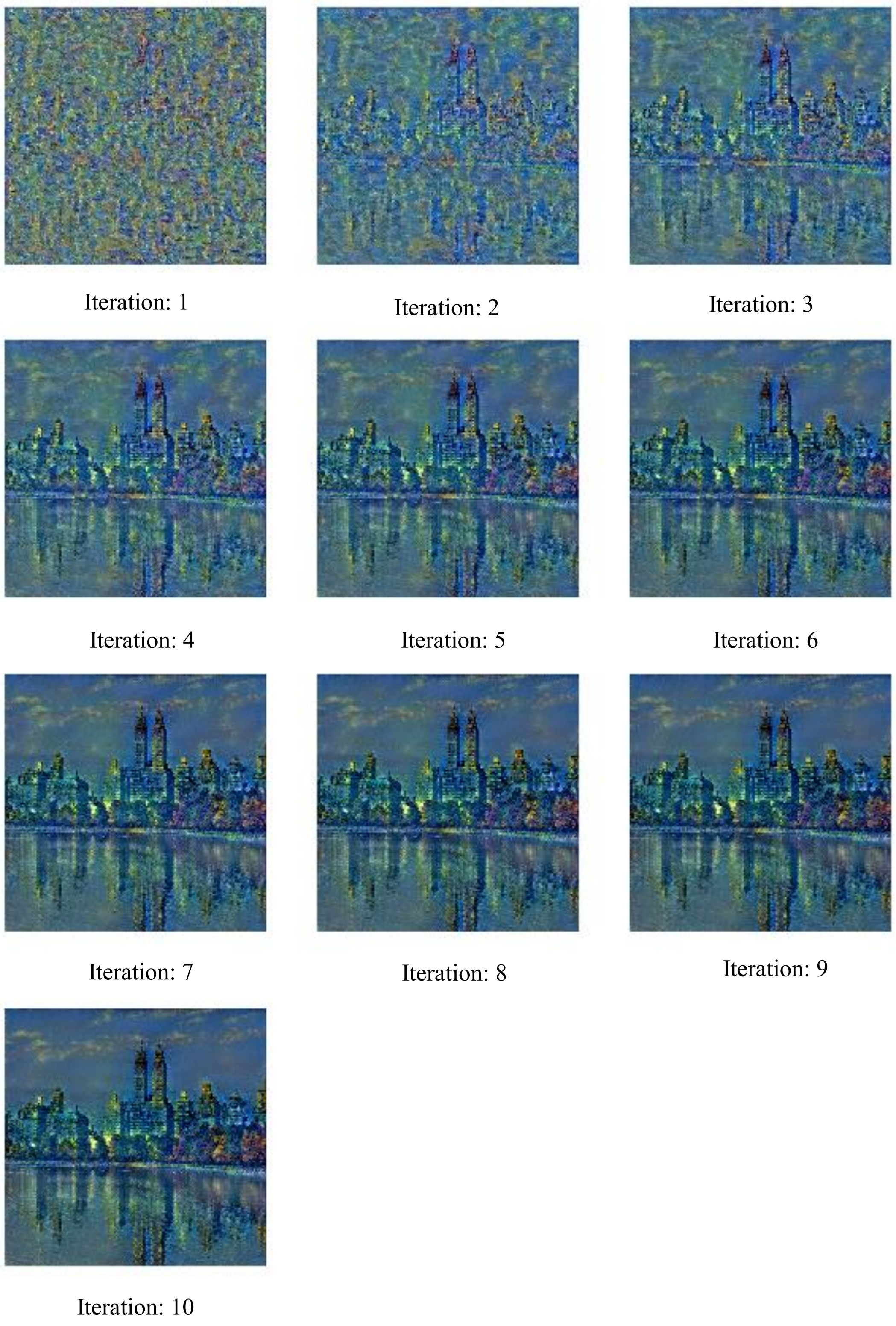


Fig 3-9: Training Process

The training loss over iteration is shown in Fig 3-10. We could see that loss converged after 7 iterations.

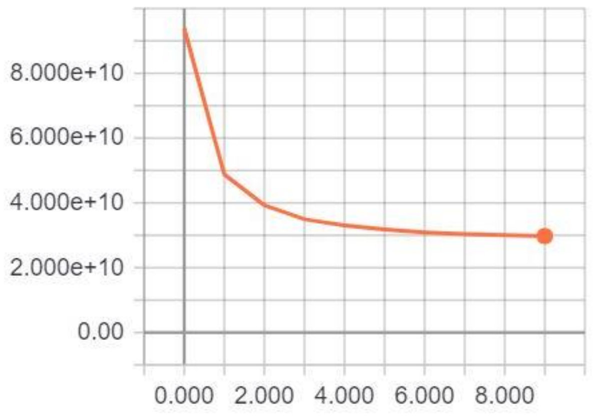


Fig 3-10: Training Loss over Iterations

3.2 Fast Neural Style Transfer

3.2.1 Network Architecture

Referring to Johnson's paper[2], we trained an image transformation network to transfer input images to a corresponding style quickly. We used VGG-16, a loss network pre-trained for image classification, to define perceptual loss functions. The loss network remains frozen during the training process.

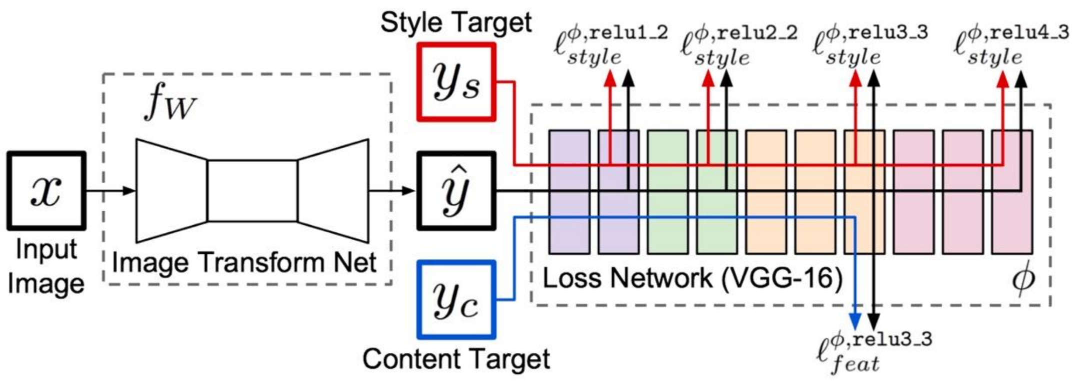
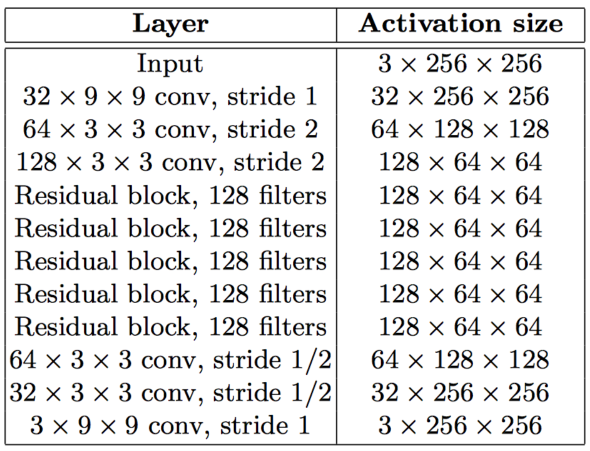

Style Target
y_s
$\ell_{style}^{\phi,relu1\_2}$
$\ell_{style}^{\phi,relu2\_2}$
$\ell_{style}^{\phi,relu3\_3}$
$\ell_{style}^{\phi,relu4\_3}$
f_W
x
Input Image
Image Transform Net
$\hat{y}$
y_c
Content Target
Loss Network (VGG-16)
ϕ
$\ell_{feat}^{\phi,relu3\_3}$

Fig 3-11: Fast Style Transfer Neural Network Architecture [2]

The image transformation network is a deep residual convolutional neural network. In Table 3-2, "C × H × W conv" denotes a convolutional layer with C filters size H × W which is immediately followed by spatial batch normalization and a ReLu nonlinearity with the exception of the output layer, which instead uses a scaled tanh to ensure that the output image has pixels in the range [0, 255].

Table 3-2 Image Transformation Network Architecture [2]



Layer	Activation size
Input	$3 \times 256 \times 256$
$32 \times 9 \times 9$ conv, stride 1	$32 \times 256 \times 256$
$64 \times 3 \times 3$ conv, stride 2	$64 \times 128 \times 128$
$128 \times 3 \times 3$ conv, stride 2	$128 \times 64 \times 64$
Residual block, 128 filters	$128 \times 64 \times 64$
Residual block, 128 filters	$128 \times 64 \times 64$
Residual block, 128 filters	$128 \times 64 \times 64$
Residual block, 128 filters	$128 \times 64 \times 64$
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	$3 \times 256 \times 256$

Based on the tuning process of 1 to 1 neural style transfer model, our pre-defined parameters for fast neural style transfer is shown in Table 3-3.

Table 3-3: Pre-defined Parameters for Fast Neural Style Transfer

Parameter Type	Parameter Value
Content Weights	100
Style Weights	0.3
Variation Weights	1e-4
Beta for Variation Loss	2.5
The number of training epochs	300
Optimizer	Adam
Learning Rate	1e-3
Content Layer	block2_conv2
Style Layers	block1_conv2, block2_conv2, block3_conv2, block4_conv2 and block5_conv2

There is one thing needed to be pointed out, we did not use L-BFGS optimizer for fast neural style transfer model because the original paper showed that Adam optimizer for this task would perform better than L-BFGS[2].

3.2.2 Dataset and Preprocessing

Dataset. For this implementation, we randomly choose 52 images from COCO dataset[1] as our content images dataset and 4 styles images from wiki-art. 4 of content images and all style images are shown in Fig 3-12 and 3-13.

1 http://cocodataset.org/#home

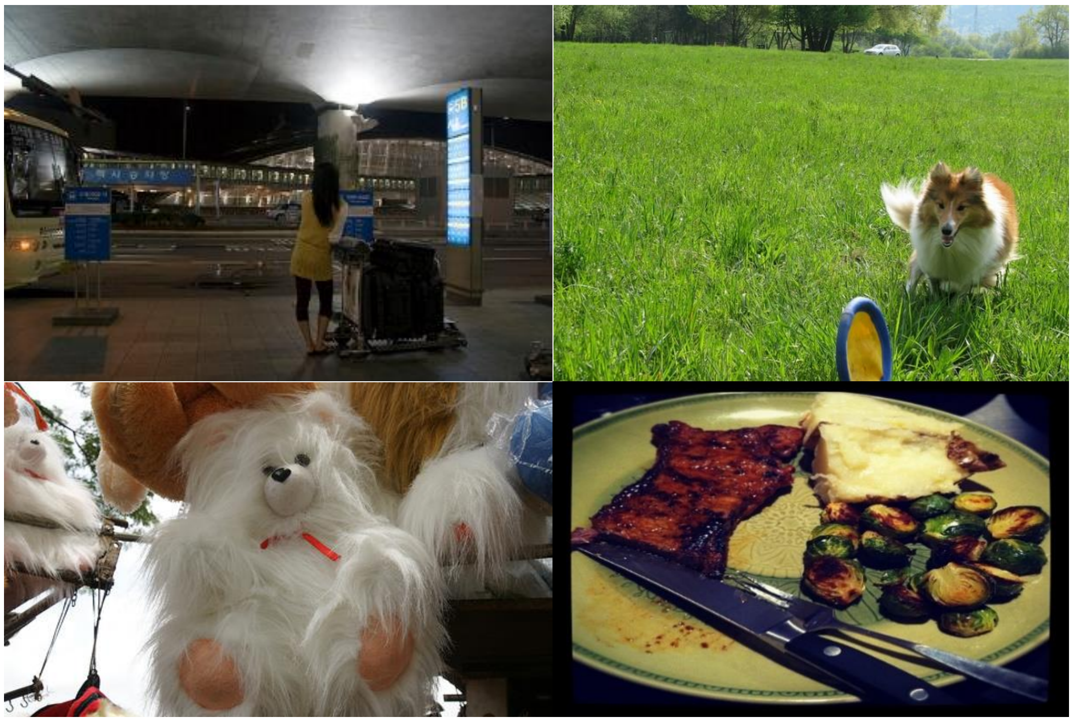


Fig 3-12: Content Images from COCO

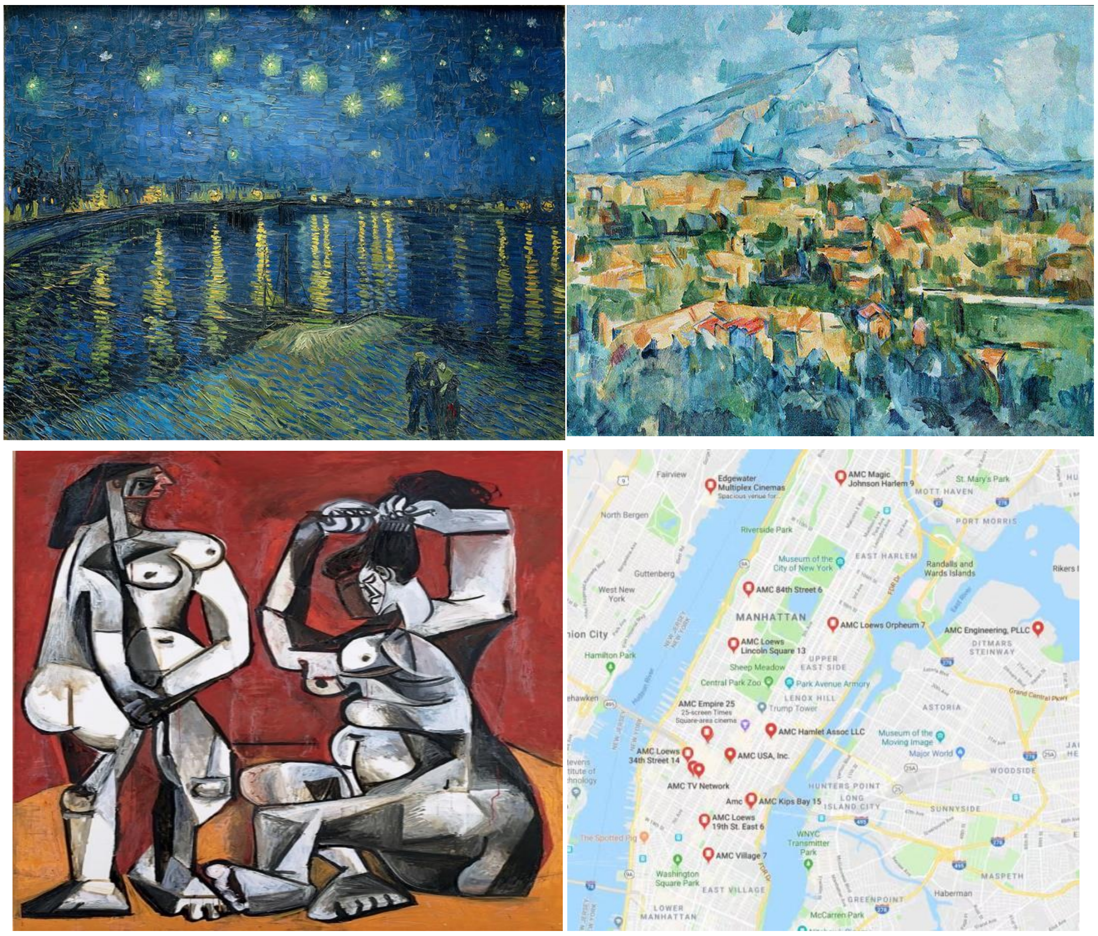


Fig 3-13: Starry Night (Left Top: Van Gogh, 1889), Victoire (Right Top: Mont Sainte, 1884), Women at Their Toilette (Left Bottom: Pablo Picasso, 1956) and AMC on Manhattan (Google Map, 2018)

Data Preprocessing. For this implementation, we resized content and style images into 512*512. Then, we subtracted the mean RGB value of all content images for the inputs of VGG16 but rescaled style images into [0, 1].

3.2.3 Results

Our results for central park is shown in Fig 3-14. We set iterations to be 15600 and the training time for one style is 1 hour. However, once the weights for transform net has been decided, we could transfer style for arbitrary images for fixed style within 0.5s. The final loss for these 4 styles are all around 34000000.

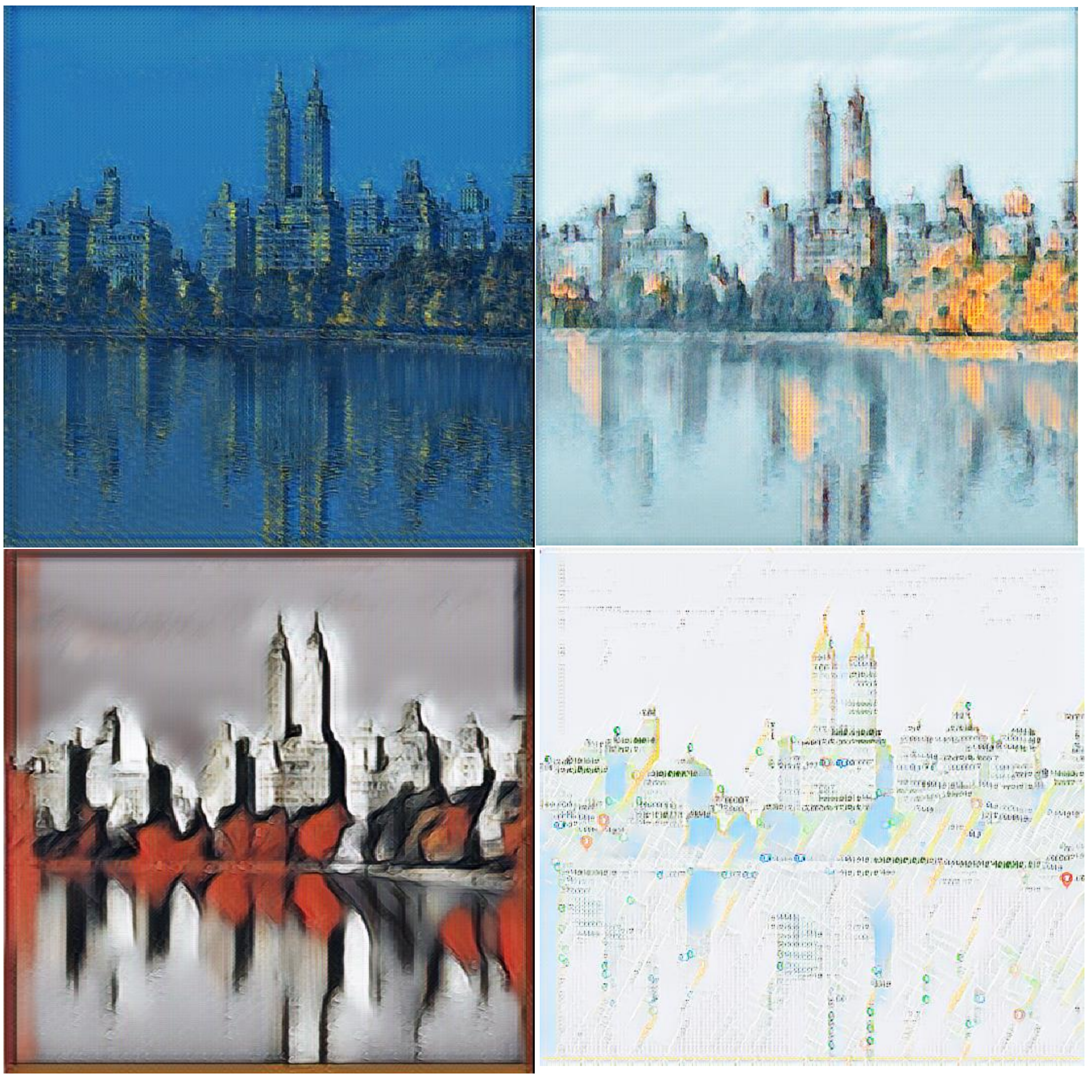


Fig 3-14: TransformNet Outputs

The weights for these 4 styles are stored in our Github repository. These outputs show that our fast neural network could successfully transfer style for a new image within 0.5s which provided the basis for our webcam application.

4. More Applications

In our 1 to 1 transfer model, we calculate the combination image using gradient method every time when we input a new pair of content and style image. It takes a few minutes to return a new combination image even when we run the algorithm on GPU. In our fast transfer net, we trained a transform net to do the style transfer job. Thus, the speed of fast transfer net is much faster than the 1 to 1 transfer model. With this much faster transfer speed, we are able to think of more application scenarios for our style transfer algorithm.

4.1 Style Transfer on Webcam

Our first extension is combining our fast transfer net with webcam. In this application, we use OpenCV library to deploy camera and capture videos. On webcam, each second's video is constructed by 30 consecutive pictures. In other words, 30 frames per second. We capture each picture in the video and transfer these pictures using our fast transfer net. Then we show these transferred images on the screen. This style transfer process goes along with camera shooting and create a sequence of style transferred pictures when the camera is turned on. Thanks to the fast speed of fast transfer net, our webcam application turns out to be a delightful real-time style transfer camera.

Fig 4-1 is a demo of our webcam application. We used Pablo Picasso's *Women at Their Toilette* as our style image to train our fast transfer net. The visual effect of our webcam is very pleasant.

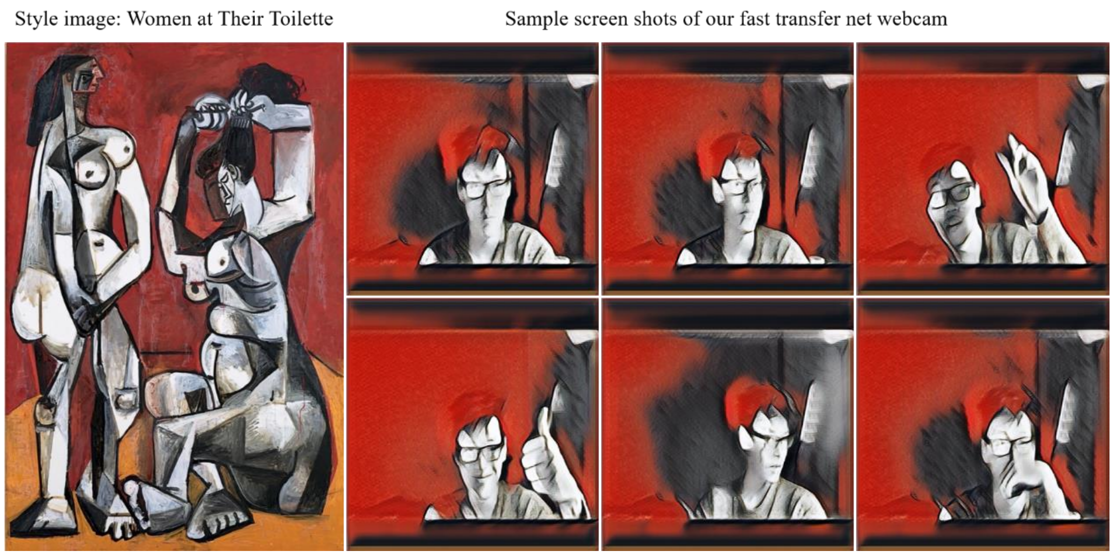
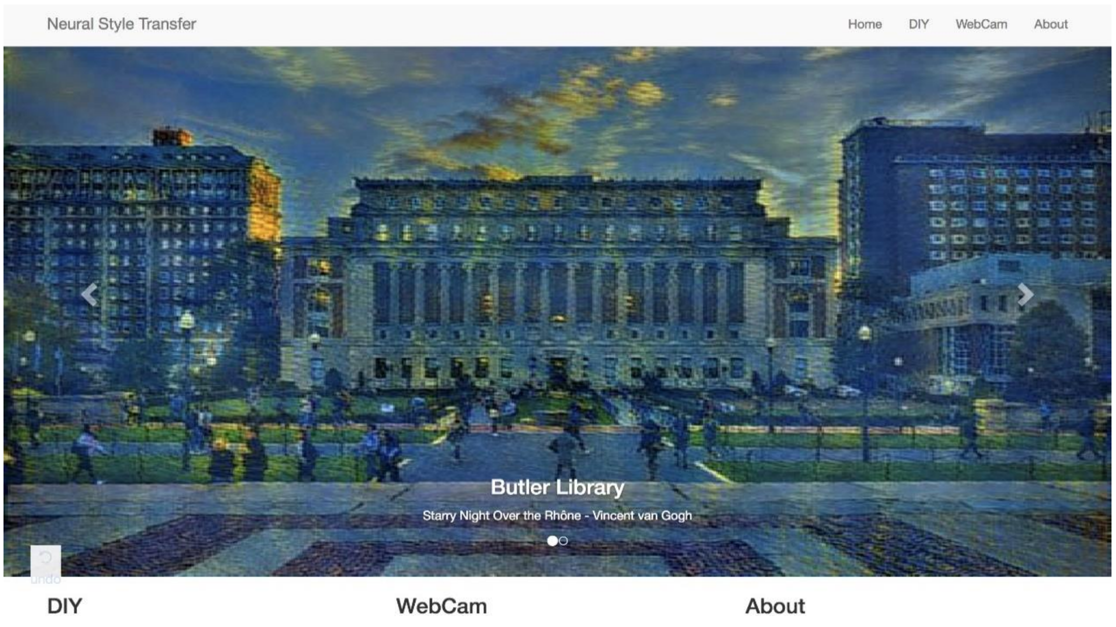

Style image: Women at Their Toilette
Sample screen shots of our fast transfer net webcam

Fig 4-1: Webcam Demo

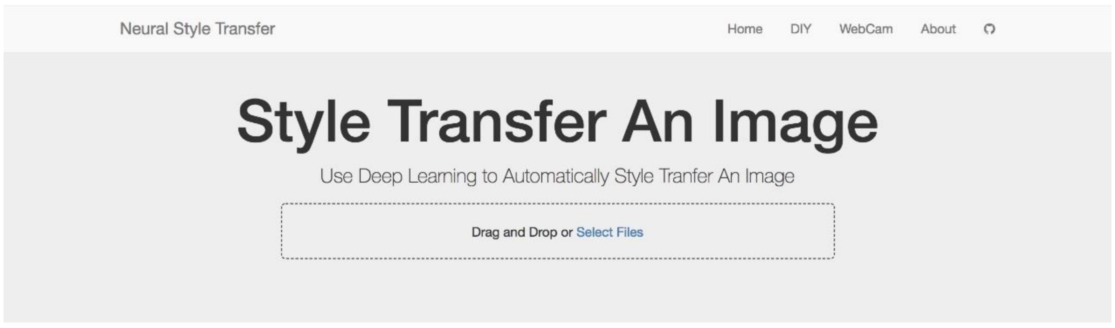
4.2 Style Transfer on Website

Except for our webcam app, we also used python dash app to design a website for our style transfer algorithm and applications. In this dash app, we put the image style transfer and webcam functions all together into a single website to allow users to play with our algorithms in any way they like. Unfortunately, unlike R shiny app, python dash app does not provide free server to ensure the functionality of the website. So we can only use screen shots to demonstrate our dash app's functions.

Step 1: Open our dash app website, we can see a homepage. The column at the top right corner shows two main functions: DIY style transfer and WebCam.



Step 2: Click on the DIY button on the top right corner, we can see the page of style transfer an image.



Neural Style Transfer

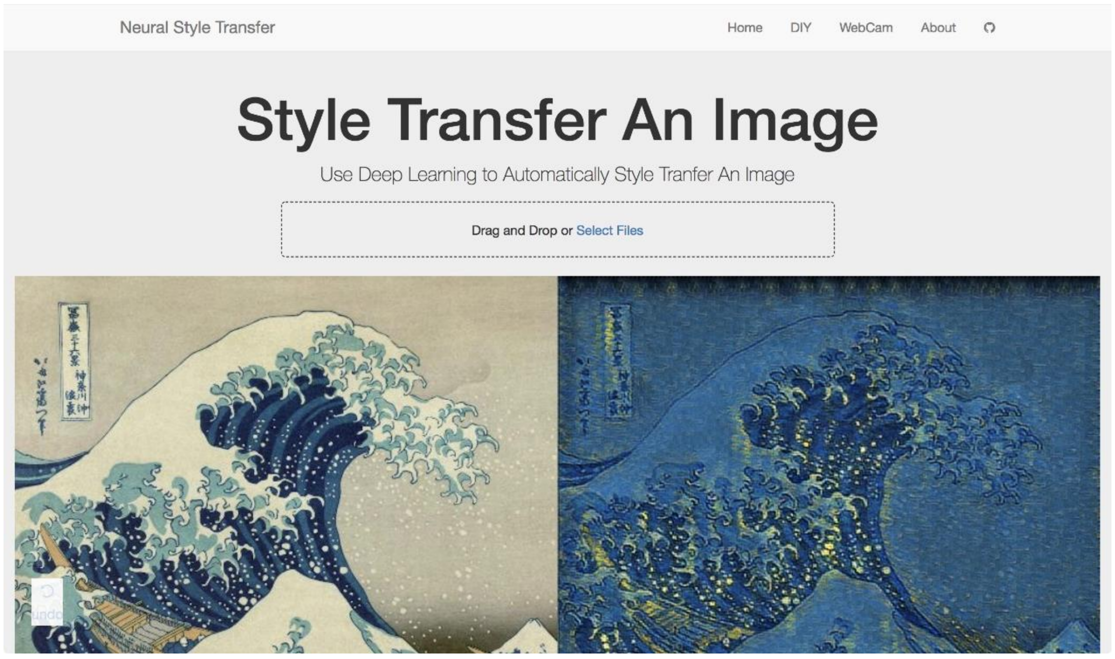
Home DIY WebCam About

Style Transfer An Image

Use Deep Learning to Automatically Style Tranfer An Image

Drag and Drop or Select Files

Step 3: Click the "Select Files" button and upload an arbitrary image, the style transferred image will appear on the right hand side of the screen!



Neural Style Transfer

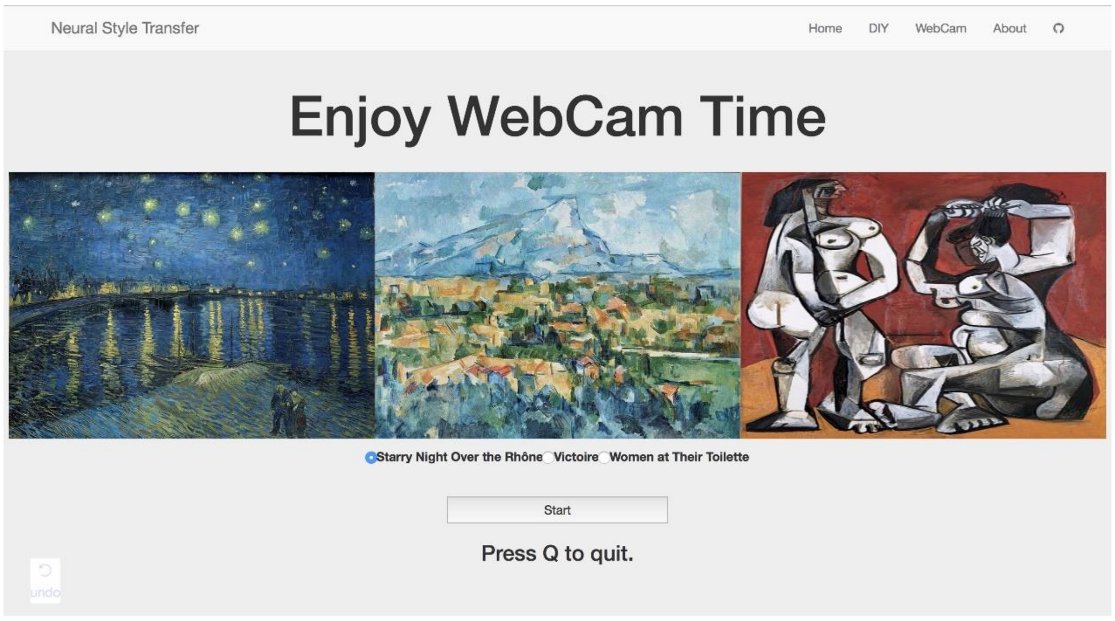
Home DIY WebCam About

Style Transfer An Image

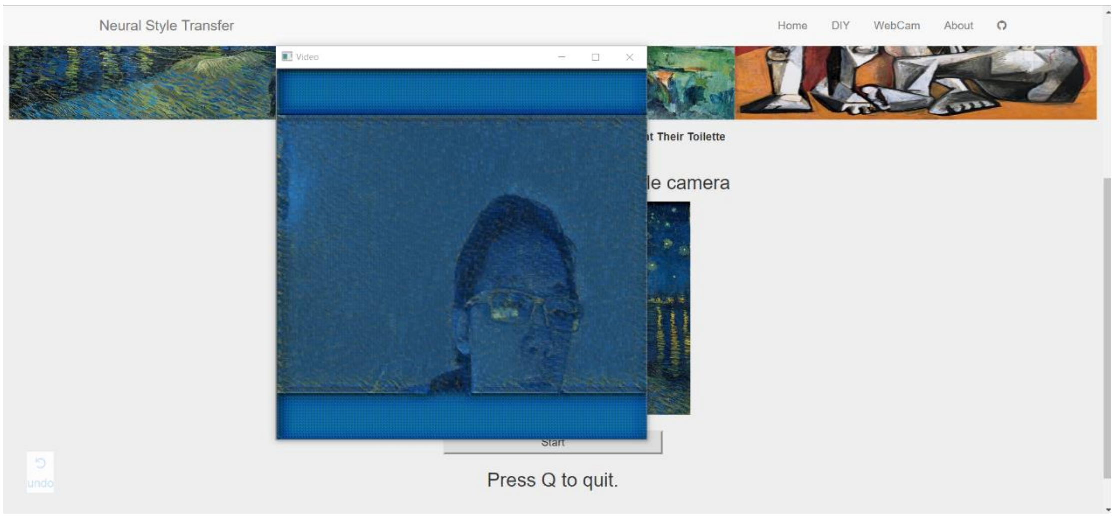
Use Deep Learning to Automatically Style Tranfer An Image

Drag and Drop or Select Files

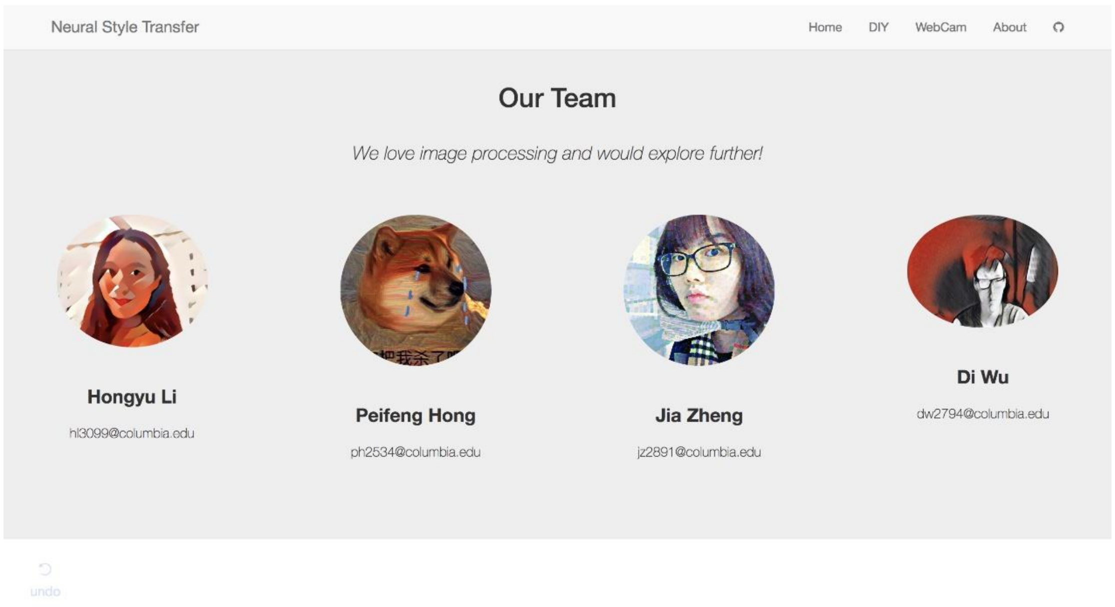
Step 4: The next function is WebCam. Click on the "WebCam" button on the top right corner, we enter the page of WebCam.



Step 5: We trained 3 different styles for you to choose from. Select a style and click on the "Start" button, a window will pop up and show you the real time style transferred camera.



Step 6: We put our style transferred photos on the about page!



Our Team

We love image processing and would explore further!

Hongyu Li
hl3099@columbia.edu

Peifeng Hong
ph2534@columbia.edu

Jia Zheng
jz2891@columbia.edu

Di Wu
dw2794@columbia.edu

5 Summary

In this project, we have implemented two style transfer algorithms: the original 1 to 1 neural style transfer algorithm (fixed style for a certain image) that was came up with by Gatys and fast neural style transfer algorithm (fixed style for arbitrary image) that was proposed by Johnson with TensorFlow eager execution. We achieved good performance on both of models. And we also applied the fast algorithm in real-time off a webcam and developed an online demo successfully.

In our works, we also found that:

- Data preprocessing plays an important role in transfer learning. Once we choose a pre-trained model, we should also apply the same data preprocessing method on our own dataset.
- Since our dataset is different from the two papers[1, 2], there are a lot of parameters needed to be tuned carefully. By designing a lot of experiments, we chose our own parameters for models and most of them are different from the original papers (see Table 3-1 and Table 3-3). For example, we used L-BFGS as our optimizer for 1 to 1 style transfer and 100(content)-0.3(style)-0.0001(variation) weights for fast neural style transfer.

In future work, we hope to implement arbitrary style transfer for arbitrary image. And it seems that we also could try to combine face detection with these style transfer algorithms to make transformation more pleasant for portraits.

References

[1] Leon A Gatys，Alexander S Ecker，Matthias Bethge. A neural algorithm of artistic style[J]. arXiv preprint arXiv:1508.06576, 2015.

[2] Justin Johnson，Alexandre Alahi，Li Fei-Fei. Perceptual losses for real-time style transfer and super-resolution[C]. Springer，2016:694-711.

[3] Falong Shen，Shuicheng Yan，Gang Zeng. Meta Networks for Neural Style Transfer[J]. arXiv preprint arXiv:1709.04111, 2017.

[4] Francois Chollet. Deep learning with python[M].Manning Publications Co., 2017.

[5] Waseem Rawat，Zenghui Wang. Deep convolutional neural networks for image classification: A comprehensive review[J]. Neural computation, 2017, 29(9): 2352-2449.

[6] Andrew Ng. Nuts and bolts of building AI applications using Deep Learning[C]. NIPS，2016.

[7] Guillaume Berger，Roland Memisevic. Incorporating long-range consistency in CNN-based texture generation[J]. arXiv preprint arXiv:1606.01286, 2016.

[8] Karen Simonyan，Andrew Zisserman. Very deep convolutional networks for large-scale image recognition[J]. arXiv preprint arXiv:1409.1556, 2014.

[9] Ciyou Zhu，Richard H Byrd，Peihuang Lu, etal. Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization[J]. ACM Transactions on Mathematical Software (TOMS), 1997, 23(4): 550-560.

[10] Harish Narayanan Blog: https://harishnarayanan.org/writing/artistic-style-transfer/.

[11] TF tutorial: https://medium.com/tensorflow/neural-style-transfer-creating-art-with-deep-learning-using-tf-keras-and-eager-execution-7d541ac31398.

[12] Neural Style Transfer Implementation with Tensorflow Graph Mode: https://github.com/Kautenja/a-neural-algorithm-of-artistic-style.

[13] Fast Style Transfer with Pytorch: https://github.com/jcjohnson/fast-neural-style.