An implementation and further research of Artistic Style Transfer with Keras

GR5242 Advance Machine Learning Final Project

Github: https://github.com/shana9pm/ArtStyleTransfer

Group Leader: Haiqi Li(hl3115)

Group Members: Yifan Wu(yw3101)
Yutong Zhang(yz3245)
Ziyan Xu(zx2232)

Contents

1 Introduction

1.1 Project background

A mobile application named “Prisma” was released in 2016 and soon became popular over the Internet. This mobile app attracted many users, especially the young generations, for it can apply artistic effects on any photo uploaded by the users, such as make a picture of a deer look like an oil painting.

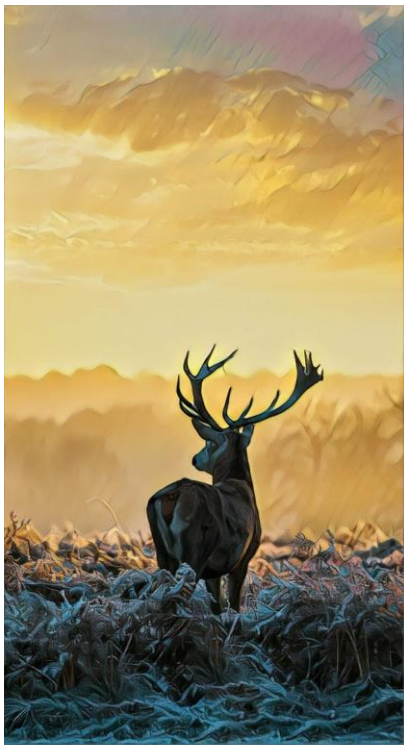


Figure 1-1: Picture of a deer rendered by Prisma

This picture-editing application applies convolutional neural network methods to extract the content and style of two pictures, and then apply the artistic style of one picture to the other, so we can get a new picture with the content of one picture and the style of another picture combined.

The artistic style transfer algorithm first was proposed in [2] and then released as [3]. The purpose of our project is to implement this algorithm described in [3] and construct a Keras based artistic style transfer system in Python with Tensorflow as Backend.

One of the most significant differences between our project and other traditional machine learning studies is that here we don’t have an objective “accuracy” measure on the output results, because this combination & recreation process is very subjective, and the variation between the content and style is also flexible. Actually, this is a not well-posed problem.

Therefore, the difficulties of this project will include:
1. Study and analyze very large pre-trained deep network data;
2. Preprocess pictures to fit the network model;
3. Define a very custom loss function to combine the content and style of input pictures.

But just an implementation of ideas is not interesting enough. We also developed our own research about this. We talked about the following things (all in Section 4):

1. Different initialization method: target picture from content picture or random initialization.
2. Different content picture and different feature transformation result.
3. Different loss function: Default square loss VS absolute loss.
4. Different feature loss weight combination.

1.2 Development environment and technical specifications

We build our virtual instance on GCP (Google Cloud Platform) with GPU acceleration. The following are part to technical specifications.

Hardware	Specifications
CPU	2.3 GHz Intel Xeon E5 v3,8 cores
memory	30 GB Memory
GPU	1 x NVIDIA Tesla K80

Table 1-2: Technical specifications

2 Mathematical Foundations

2.1 Convolutional Neural Network and rectified linear units

Nowadays one of the most popular areas in deep learning is visual perception, and it includes facial recognition, visual cognition, etc. Because of its high perceptual quality, deep neural network models, especially Convolutional Neural Networks (CNN), are the most commonly used among all kinds of vision models. [2,4,5]

Fundamental principles of CNN are already talked about in our class and many research papers. Therefore, here we will skip this discussion.

One more thing need to mention is that there have been many works talking about activation function selection, and most commonly seen methods such as sigmoid function, softmax function, and rectified linear units (ReLU). Introduced in [6,7], ReLU helps accelerating the training process and preserves information while going through network layers. This is an adorable property especially when we are training very large neural network, and it can help improve model quality in areas where information preservation is important, such as visual perception.

2.2 VGG network

The most time-consuming stage of constructing a Convolutional Neural Network is the training/fitting part. One needs huge amount of training/validation data, and high-performance computing units or even parallel computation technics to fit a CNN model. This sounds impossible to achieve within few hours with an ordinary personal computer.

Thus like the original work, we choose to use a pre-trained CNN model by Visual Geometry Group (VGG). In [8] the authors researched about the effects of CNN depth on large-scale image recognition, and they trained some networks with different number of layers from 16 to 19. We will use the feature space specified by the VGG-network on [9].

2.3 Baseline Loss function (include total loss, content loss and style loss)

2.3.1 Total loss

The total loss includes two parts: content loss and style loss. Among these three, content and style loss are also called pixel loss because the Euclidean distances are calculated pixel-wise.
The loss function has form:

$$L_{total} = \alpha L_{content} + \beta L_{style} \tag{2-1}$$

Different weights of each loss component will lead to different output. And larger α/β ratio helps resemble content, while smaller α/β ratio emphasises more on the style.
In this work we choose $\alpha/\beta = 10^{-4}$. Also we will discuss about different ratio in Section 4.

2.3.2 Content loss

The content loss measures the ℓ_2 loss between input content image and output picture:

$$L_{content}(x, p, l) = \frac{1}{2}\sum_{i,j}(x_{ij}^{l} - p_{ij}^{l})^2 \tag{2-2}$$

where p_{ij} is the original image's feature representation in l^{th} layer, x_{ij} is the output picture, and $|l|$ is the size of image representation at the layer: the content loss function is the normalised Euclidean distance between the original content representation and the output image.

2.3.3 Style loss

The style loss measures the ℓ_2 loss between input style image and output picture but it's more complicated than content loss function: a Gram matrix is constructed to represent the feature correlations through inner product between feature map

$$G^{l}(x) = F^{l} \cdot F^{l\prime}$$
$$E^{l}(x, a) = \frac{1}{4N_l^2 M_l^2}\sum_{i,j}(G_{ij}^{l}(x) - a_{ij}^{l})^2 \tag{2-3}$$
$$L_{style}(x, a; w) = \sum_{l} w_l E^{l}$$

where F is the feature map matrix, a_{ij}^{l} represents the style representation in l^{th} layer, E^{l} represents the style loss in l^{th} layer and w_l is the loss weight for each layer.
Similar to content loss function, the style loss on each layer is also normalised Euclidean distance between the original style representation and the output image, and total style loss is a weighted average of losses from each layer. Here we set $w_l = \frac{1}{L}$, where L is the number of layers used for style feature extraction.

2.3.4 Absolute loss

Also the original paper use the squared loss, we would like to go further about absolute loss here. Apparently, if we change to absolute loss, we need to not only change the square part, but also the balanced part of denominator. We change the formula of 2-2 and 2-3 to the following representation:

$$L_{content}(x,p,l) = \frac{1}{2}\sum_{i,j}|x_{ij}^l - p_{ij}^l|^2$$
$$E^l(x,a) = \frac{1}{4N_lM_l}\sum_{i,j}(G_{ij}^l(x) - a_{ij}^l)^2 \quad (2\text{-}4)$$

3. Keras Implementation

3.1 System overview

The style transfer application is implemented with Python Keras, with many packages including NumPy, SciPy, and Pillows. A flowchart of the overview is provided in Fig. 3-1.

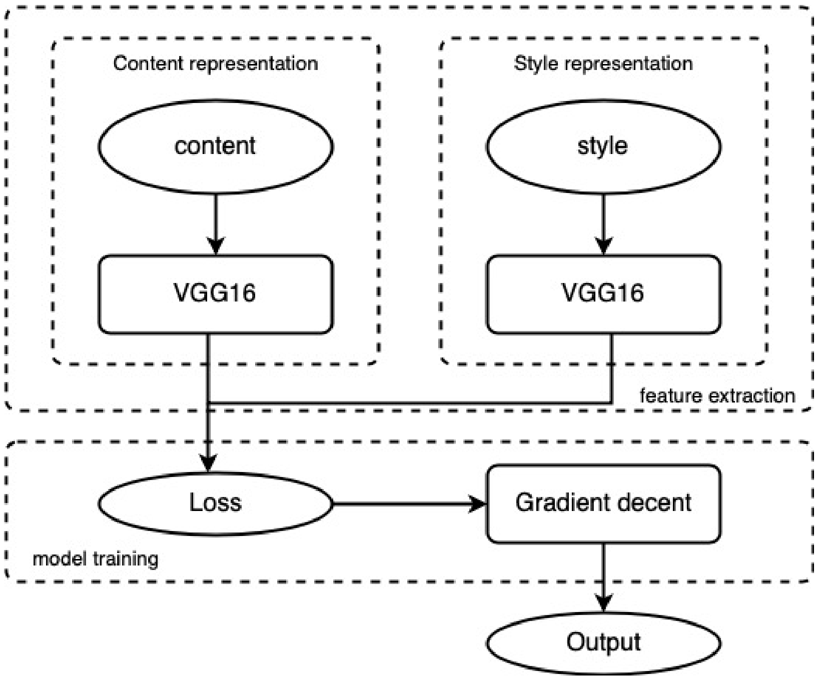


Figure 3–1: Keras implementation overview

In feature extraction part, the pre-trained VGG network is used for both content and style representation. We extract ‘block4_conv2’ from content model and first four block from style model features for feature extraction.

In the training stage, as we talked above, we have two initialization method. One is initialization from random. The other is from content initialization. We will discuss the two method in detail in Section 4. The use scipy L-BFGS as our optimization method with early stopping.

3.2 Default parameters

Some tuning parameters are hard-wired in this model. Most of the values follows the original work [3] with some minor modifications.
Some pre-set parameters are listed bellow in Table 3-2. Note that one of our research target is to try different hyperparameters to see want happens and summarize about it, so in Section 4 we will change

specific hyperparameters in each sub-section. Some hyper-parameters are not mentioned before because we will go in details in the following sections.

Parameter	Value
content weight(alpha,α)	1
style weight(beta,β)	10000
maximum iterations	600
optimization method	L-BFGS
loss type	square loss
feature layer weight	[0.25,0.25,0.25,0.25]
initialization method	Content picture
structure of feature extract	VGG16[9]
pre-load weight	Imagenet[4]
content picture	Butler_Library_night.jpg

Table 3-2: Default model parameters

3.3 File structure and code running instruction

Our project is structured as follow:

```
project/
   |---- input/
   |       |---- content/
   |       |        |---- content.jpg
   |       |---- style/
   |       |        |---- style.jpg
   |---- output/
   |       |---- output.jpg
   |
   |---- ArtStyleTransfer.py
   |----Settings.py
   |----utils.py
```

In the input director, we have two content pictures ‘Butler_Lib_s.jpg’ and ‘Butler_Library_night.jpg’. ‘Butler_Lib_s.jpg’ is afternoon Butler library and ‘Butler_Library_night.jpg’ is Butler library at night.

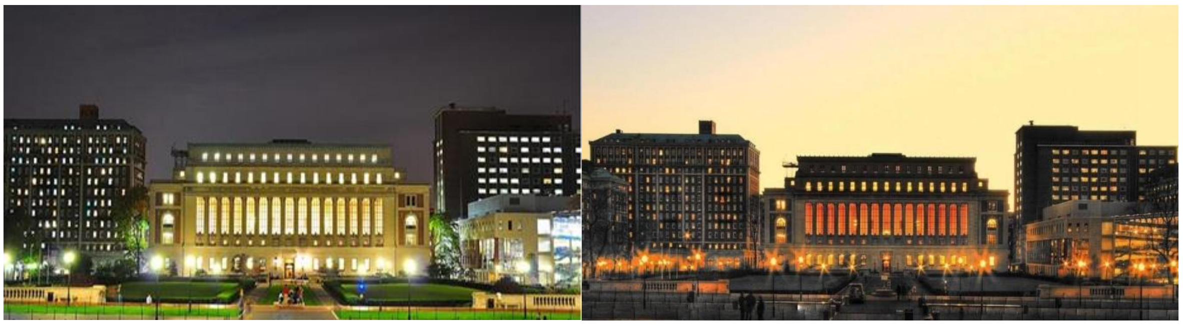


Figure 3–3: Butler_Library_night.jpg (left) and Butler_Lib_s.jpg(right)

To run the code, just cd into the directory and run the following code:

```
python ArtStyleTransfer.py --content content.jpg --style style.jpg --output output.jpg
```

--iter: num of iterations you want to specify. Default 400.

```
python ArtStyleTransfer.py --content content.jpg --style style.jpg --output output.jpg
--iter 600
```

--record: Default False. True for record the loss of each step and plot them in output dir.

```
python ArtStyleTransfer.py --content content.jpg --style style.jpg --output output.jpg
--record T
```

--flw:The weight of each feature layer in VGG structure. The exact weight is in Settings.py. The number of it is the index of layer weight list. Default 0 to be [.25,.25,.25,.25]

```
python ArtStyleTransfer.py --content content.jpg --style style.jpg --output output.jpg
--flw 3
```

--lt:loss type. Default to be sqaure loss. Another choice is absolute loss. Note the loss function is changed.

```
python ArtStyleTransfer.py --content content.jpg --style style.jpg --output output.jpg
--lt AE
```

--rstep: record per step. Default 50 means record target picture every 50 steps.

```
python ArtStyleTransfer.py --content content.jpg --style style.jpg --output output.jpg
--rstep 10
```

--alpha --beta: The parameter in paper of loss weight. Alpha is weight of content loss and beta is weight of style loss.

```
python ArtStyleTransfer.py --content content.jpg --style style.jpg --output output.jpg
--alpha 10.0
```

--fromc: The target picture initialization method. Default False to be random initialization. True to initialize from content picture.

```
python ArtStyleTransfer.py --content content.jpg --style style.jpg --output output.jpg
--fromc T
```

--cont: Continue training. Recommend not to use this.

If you really want to try this, go as the following:

1.upload the sList.dat from output to dir output

2.Specify iteration to be like

iteration=iteration you have already trained + iteration you want to go further

EX:if you have trained 400 iterations and want to continue to 600 iterations as total, then

```
python ArtStyleTransfer.py --content content.jpg --style style.jpg --output output.jpg -
-cont T --iter 600
```

It will train another 200 iterations.

4. Our research

4.1 Target initialization method

Two different initialization methods:

Different target initialization methods have different effects on the final output and which is quite vital. There are two different initialization methods:

1). Initialize from the content (the imported photo)

2). Initialize from random (a random photo)

If we initialization from the target photo, our output will more like the target and might has few differences, but if we initialization from random, the output might seems quite different at all.

4.1.1. Compare output from different initialization methods.

From Figure 4-1,it is easy to find that, when initialized from random, the network will provide a mess when the iteration number is low, this is because we initialized the target photo from a random picture. Then, with the growth of the iteration time, we can see the output is more and more closed to the ideal output. After about 300 iterations, our output tends to be converged.

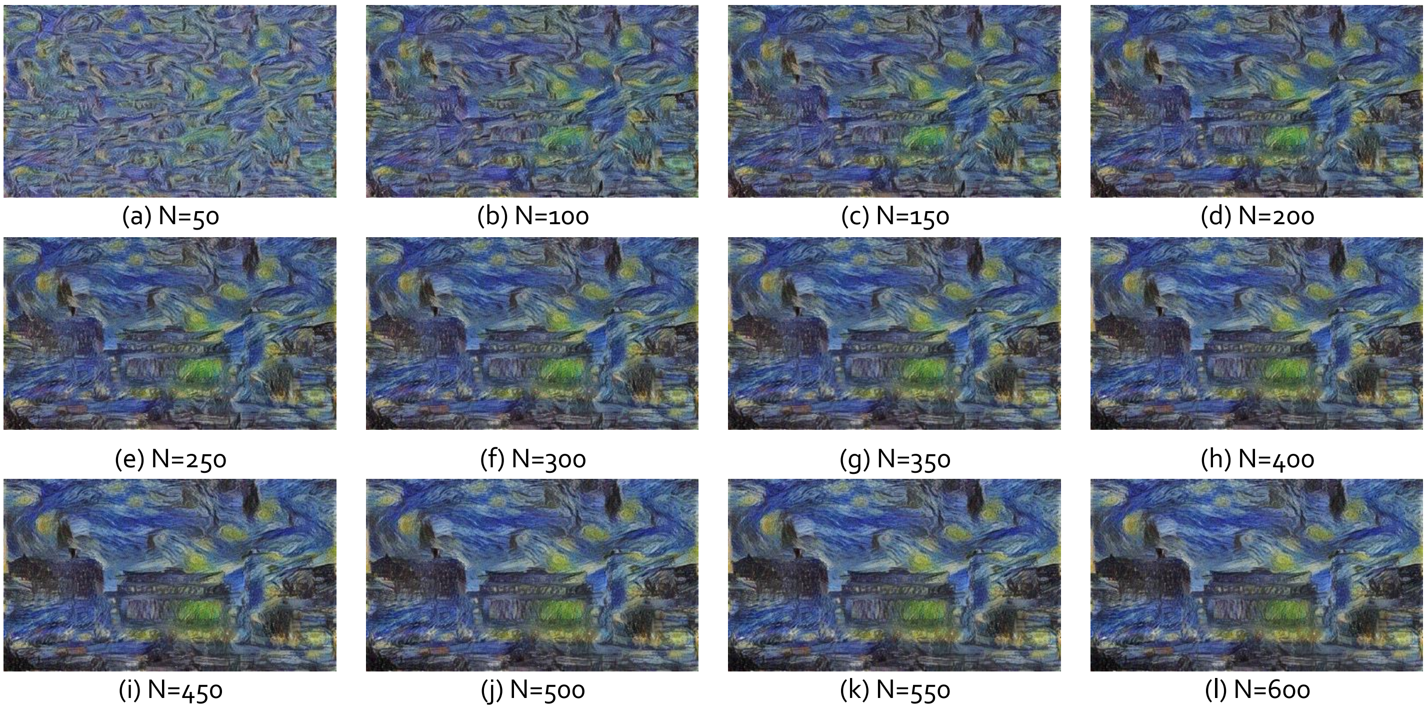


Figure 4-1 Output, initialization from random picture and use different times of iterations

However if we initialize from the content, which is the target picture, we can see from Figure 4-2, the photo we input seems have not changed much during few iterations (about 50 to 150). If we enlarge the iterations, the output tends to change a lot compared with the original input and closed to our target. After 400 iterations, the output tends to be converged. (Figure 4-2 is listed below)

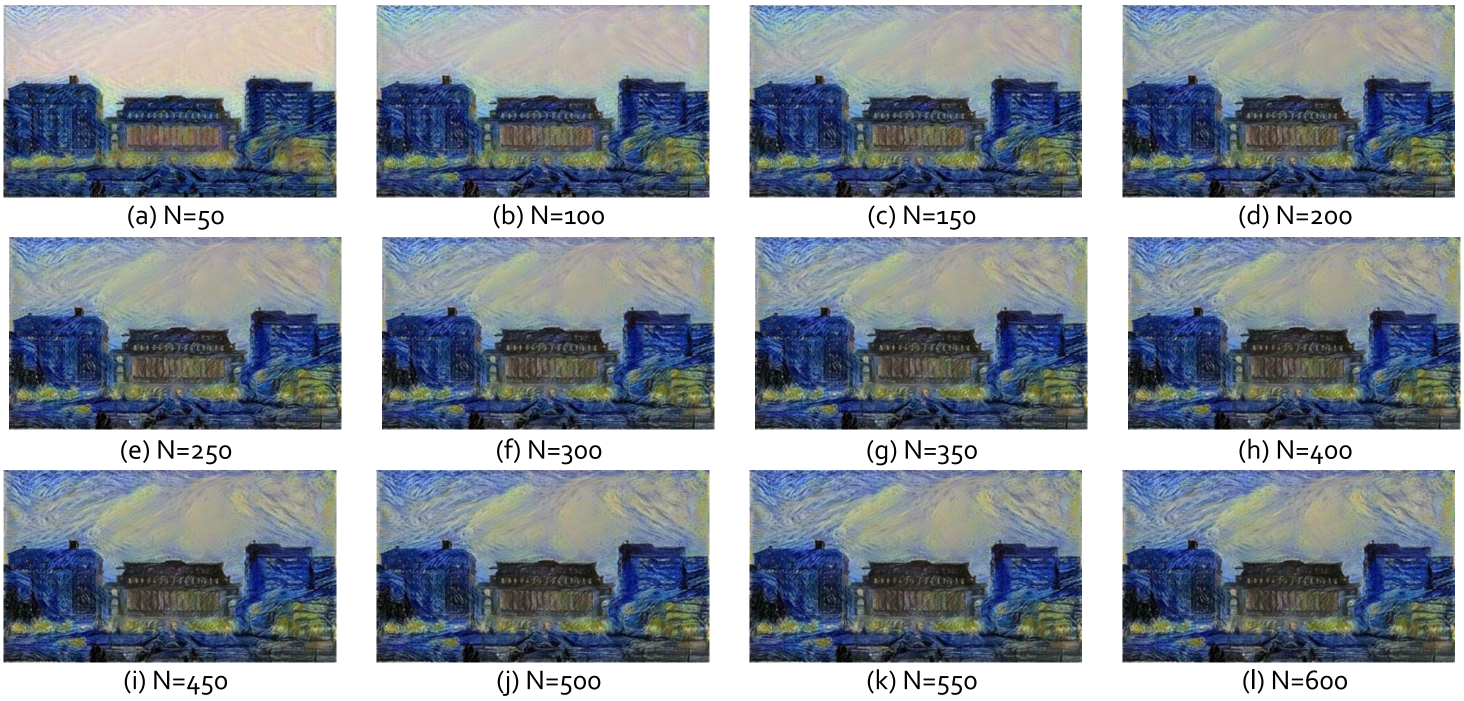
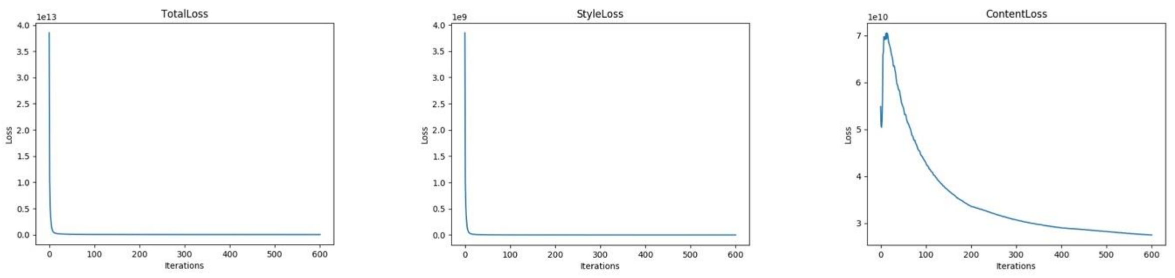


Figure 4-2 Output, initialization from content (input picture) and use different times of iterations.

To conclude, after comparing the output of two different initialization methods, we can see that, if we initialize from a random picture, our output will look much more like the pattern we choose in the model, (in this case, Van Gogh Painting). While if we initialize from the input picture, our output will look much more like the picture we put in the model, (in this case, Butler Library of Columbia University). Also it is quite clear that, a larger iteration times will cause the model to converge and to be much more valid.

4.1.2 Compare loss of two methods.

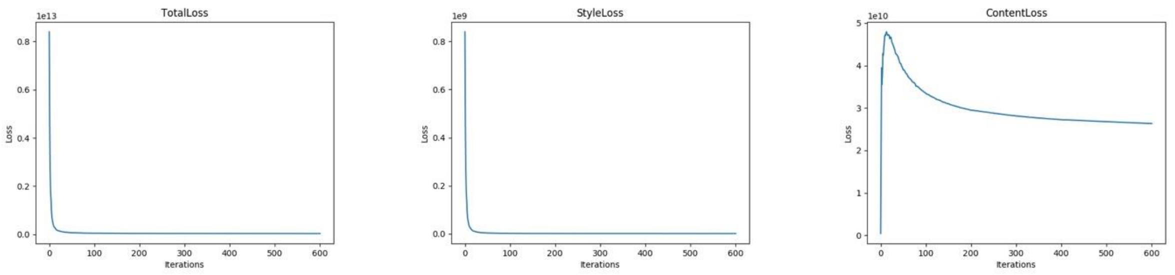
We compare the loss of two methods according to their style loss, content loss and total loss. Figure 4-3 and Figure 4.4 show the difference.



(a) Total Loss (b) Style Loss (c) Content Loss

Figure 4-3 Total Loss, Style Loss and Content Loss of random initialization.

From Figure 4-3, it is obvious that the Total Loss and Style Loss keeps dropping quickly and converges quickly. While the Content Loss starts at $5 \times e^{10}$ and has a jump in the lower iteration and tends to converge then. This means the random initialization will cause a mess in the first period of iterations. When the iteration times is larger, the content losses will converge then. This also fits the conclusion that the random initialization will look much more like to the pattern itself.



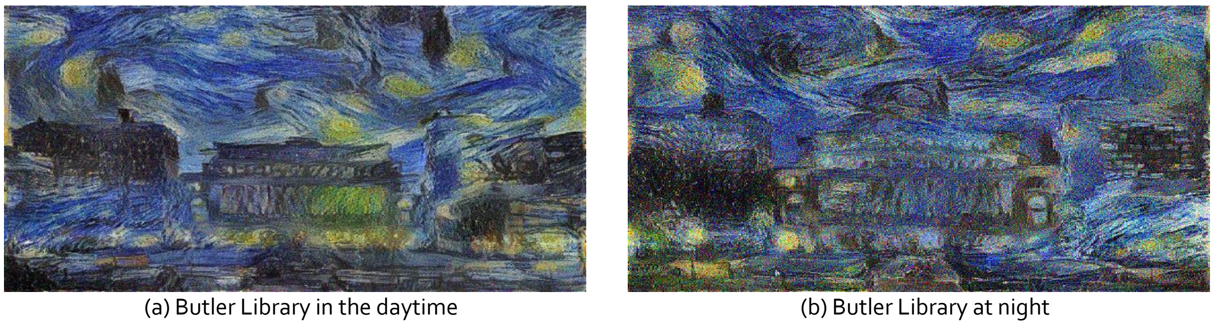
(a) Total Loss (b) Style Loss (c) Content Loss

Figure 4-4 Total Loss, Style Loss and Content Loss of content initialization.

From Figure 4-4, we can also see that the Total Loss and Style Loss keeps dropping quickly and converges quickly. However it is obvious that the content loss is much more different than that in Figure 4-3. The Content Loss starts at 0, not $5 \times e^{10}$. This means if we initialization from the photo imported, at first time, the content loss is extremely small. After the iteration, the loss will grow much faster and then tends to converge when the iteration numbers is larger. So we can see the content initialization method will make the output look much more like the content itself.

4.2 Similar content picture comparison

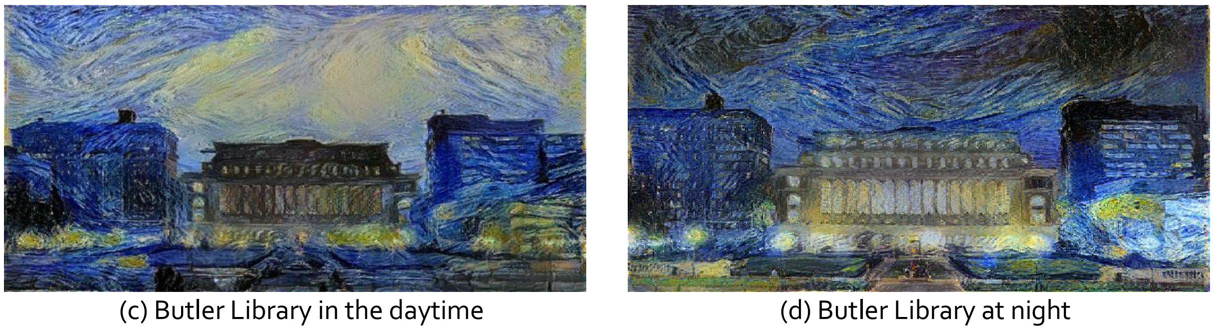
It is known by all that, if we put several similar photos (by similar, we mean photos of a place but taken in different time) in the model, our output will be different because the light, the environment, the background will definitely affect the effect of output. For example, if we use a pattern in light color, a photo taken in daytime and a photo taken at night, it is obvious that the daytime one will have little change and the night one will change a lot, vice versa. So we want to compare such a scenario. In this part, our example is use the photo of the Butler Library in Columbia University taken in daytime and night, and use the pattern (Van Gogh Painting) which is a pattern in dark background.



(a) Butler Library in the daytime
(b) Butler Library at night

Figure 4-5 Butler Library in the daytime vs at night, using initialization to random.

Focusing on the background , we can see that the photo taken in the daytime is brighter than that taken at night. Also we can see, in Figure 4-5 (b), the effect of the output is not good. This is because the boundary of the Butler Building at night is not clear compared to that in the daytime. To summarize, if we use initialization to random, the background effect is less obvious because the initialization process is conducted from a random photo. If we want to improve the effect, we may adjust the ratio of α, β.



(c) Butler Library in the daytime
(d) Butler Library at night

Figure 4-6 Butler Library in the daytime vs at night, using initialization to content

After comparing Figure 4-6, we can see the effect of the background color really affect the output. The photo taken in the daytime is brighter in the background (the sky in the figure),while the photo taken at night is brighter in the scenery (the Butler Library). The initialization to content show this effect much better than the other initialization method. What's more, we can clearly find that in Figure 4-6 (d), the feature of Van Gogh painting is shown obviously on the output. Focusing on the background, we can classify the background of the sky is , to some extent, quite similar to that in Van Gogh painting. That is to say, more features are returned in the "dark sky" background in the photo of night than that in the daytime. The problem now is to do research on the features of the patterns, which will need to changes the weights of the model and will be discussed in the chapters below.

Also we think that since 'Starry Night' is a picture at night, the feature in that picture in more suitable at night. So generally, we can see that Butler library at night get more features from style picture and has a better generalization affect (at least we think). We will discuss more about features in Section 4.4

4.3 Applying Absolute Loss

With initialization from the content picture and saving intermediate results every 50 steps, the output image at each checkpoint is listed in Figure 4-7.

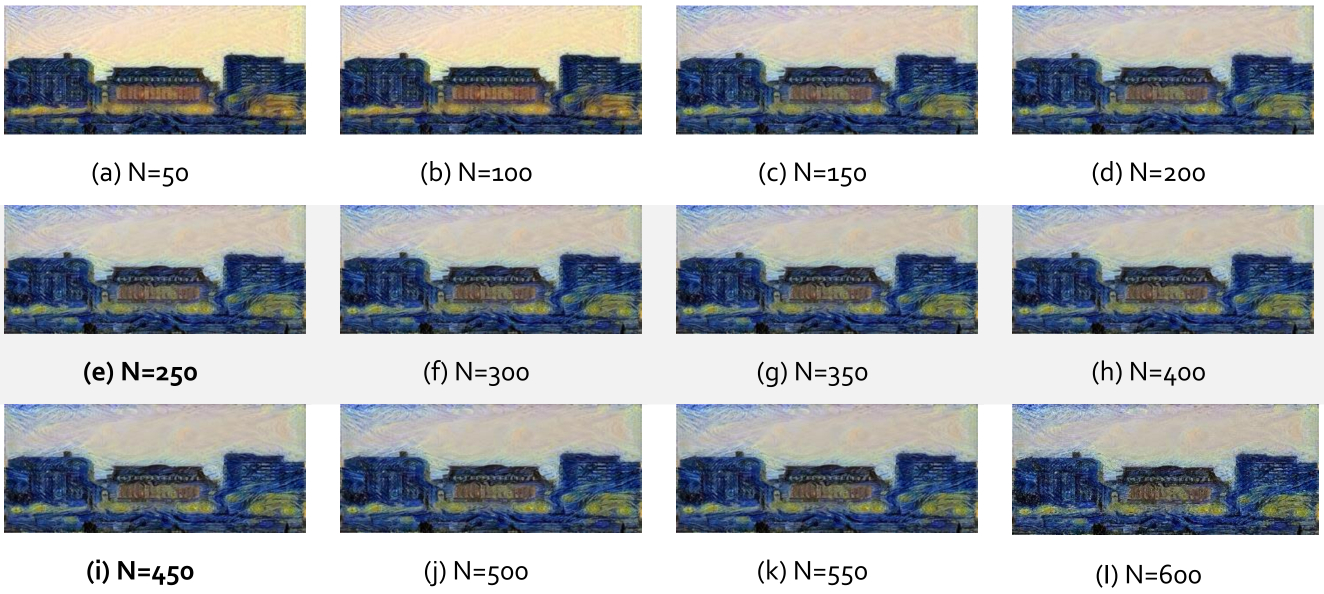


Figure 4–7: Tracing output from absolute loss

Compare with square loss function, we think that there is no obvious difference between two loss function. We think the result is reasonable because we modify the whole function (as shown in absolute loss), the loss calculation method is almost the same, only different in formula expression. The basic ideas of two is same so the output is alike.

4.4 Loss ratio of content and style

As discussed in Sec.2.3, α and β are the weighting factors for content loss and style loss, and the ratio α/β is the relative weight between content and style reconstruction. The default ratio is 10^{-4}, and now it is changed to 10^{-3}, the ratio indicates the emphasis on matching the content photograph and the style picture. Different ratios will lead to different trade-off between content loss and styleloss, and the user should be able to adjust the parameter to achieve a more suitable output result based on different needs.

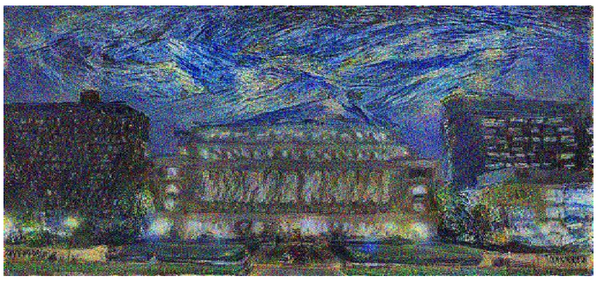
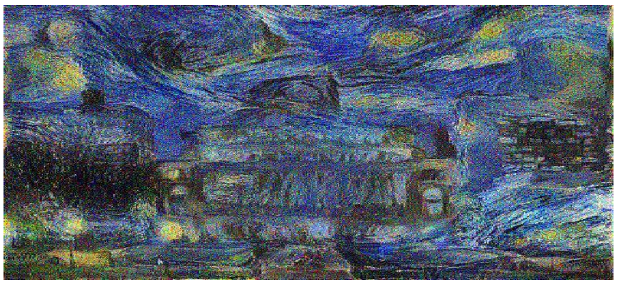



Figure 4-8 Comparison of Loss ratio of $\alpha/\beta = 10^{-3}$(left) and $\alpha/\beta = 10^{-4}$(right)

With $\alpha/\beta = 10^{-4}$, we can see that the outline of the library in the output photo is not clear, the reason could be the blurred the outline of the library at night as we can see in the Raw input of Butler Library photo at night. In this case, we adjust the ratio to 10^{-4} in order to capture more content representation and clearer shape of the library.

With initialization to random, by changing the ratio from 10^{-4} to 10^{-3}, ratio exerts relatively higher emphasis on content feature extraction, therefore the output would resemble the raw input more. We can tell that the unclear Butler Library is now clear, which indicates that the default ratio does not make enough emphasis on matching the night Butler Library. From the target pictures, it is obvious that the Butler Library of $\alpha/\beta = 10^{-3}$ is much cleared than $\alpha/\beta = 10^{-4}$.

4.5 Different feature layer weight combination

The results presented in the main text were generated on the basis of the VGG-Network,[1], a Convolutional Neural Network that rivals human performance on a common visual object recognition benchmark task.[2] we utilize a pre-trained neural network model by the Visual

Geometry Group (VGG), and we used the feature space provided by the 16 convolutional layers of the VGG-Network.

Convolutional Neural Networks are the most powerful class of Deep Neural Networks in image processing tasks. Each layer can be understood as a collection of image filters, each of which extracts a certain feature from the input image. Thus, the output of a given layer gives differently filtered versions of the input image. Apart from the choice of layer, the weighting factors of the contribution of each layer to the total loss would affect the output.

As shown by the two functions above, E_l is the contribution of that layer to the total loss. w_l is the weight on the contribution of that layer to the total loss. These weighting factors can be adjusted for different art style.

As shown in Fig. 4-10, we matched the style representations on layers 'block1_conv1', 'block2_conv1', 'block3_conv1', 'block4_conv1', with baseline setting w_l = 0.25 in those layers.

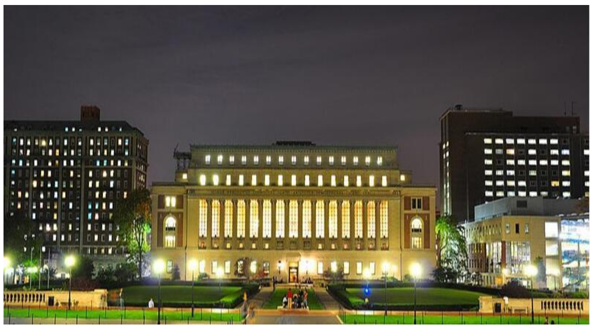
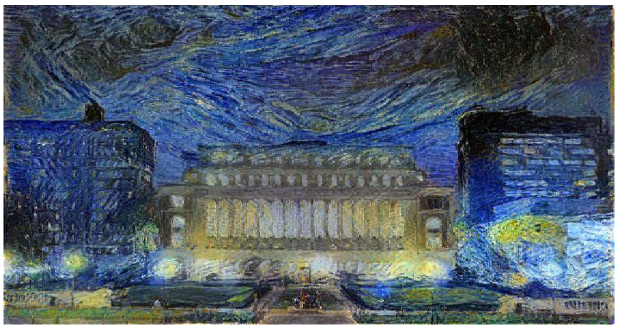
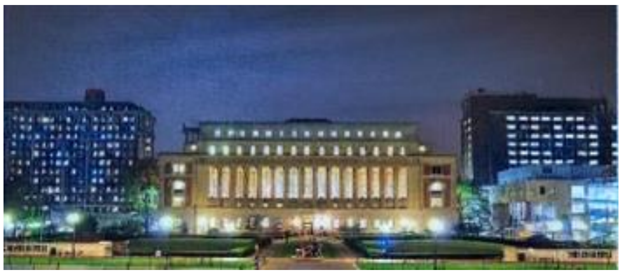
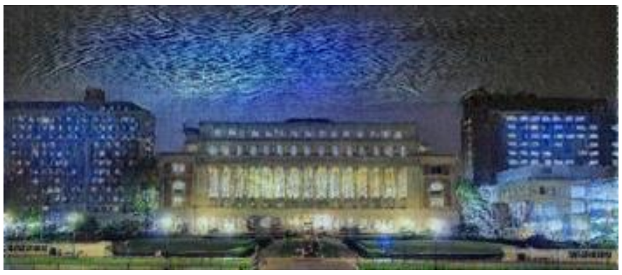



Figure 4-9 Content picture of Night Bulter Library and the Baseline target picture

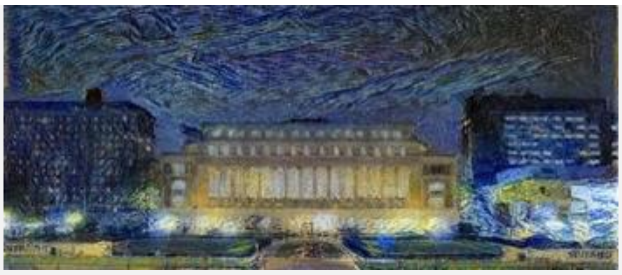
Fig 4-9 shows results for different relative weightings of the content and style reconstruction loss (along the columns) and for matching the style representations on layer 'block1_conv1', 'block2_conv1', 'block3_conv1' and ‘block4_conv1’ with weights $[w_1, w_2, w_3, w_4]$ correspondingly.

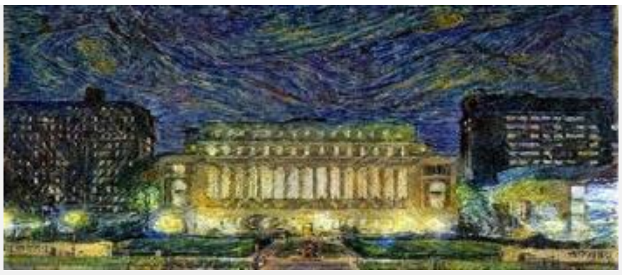


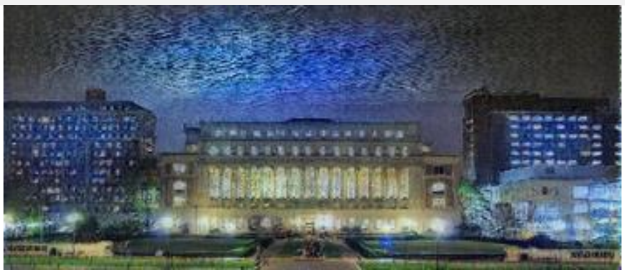
(A) [1, 0, 0, 0]

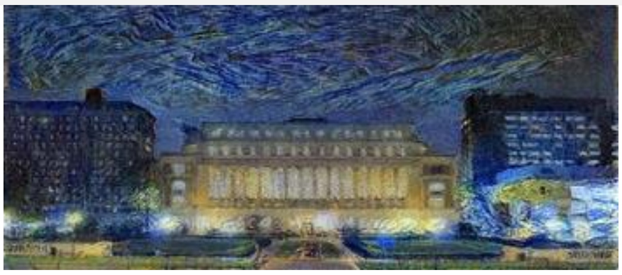


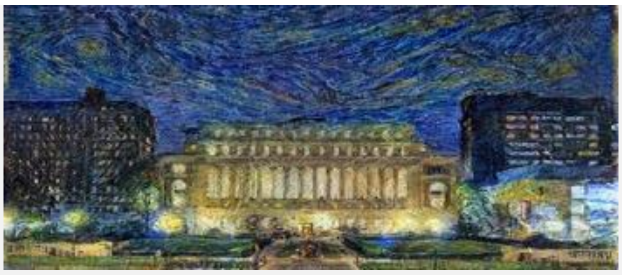
(B) [0, 1, 0, 0]

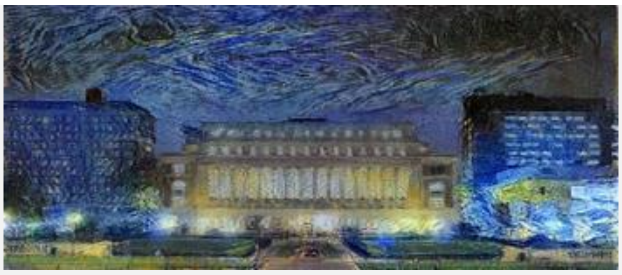

(C) [0, 0, 1, 0]

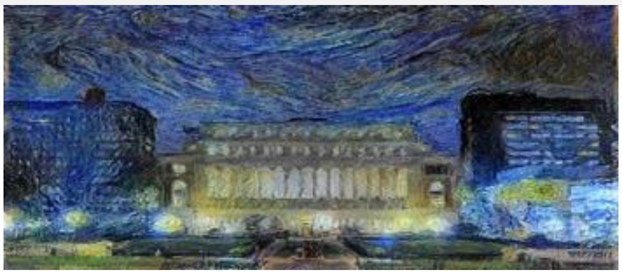

(D) [0, 0, 0, 1]

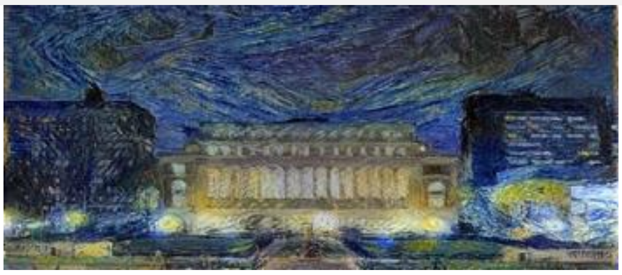

(E) [.5,.5,0, 0]

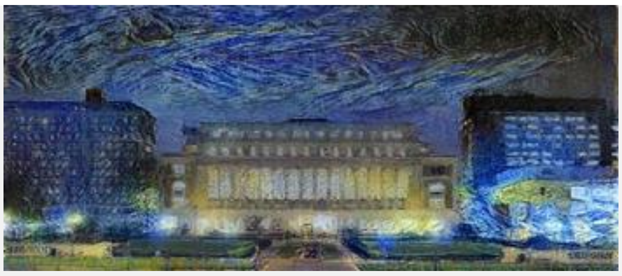

(F) [.5,0,.5, 0]

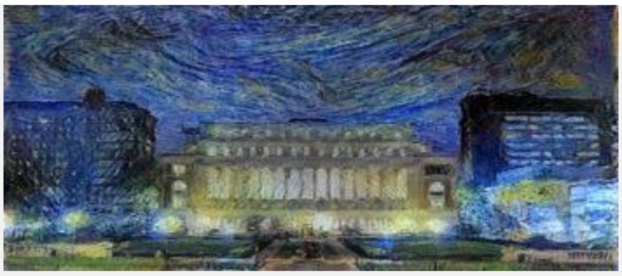

(G) [.5, 0,0,.5]

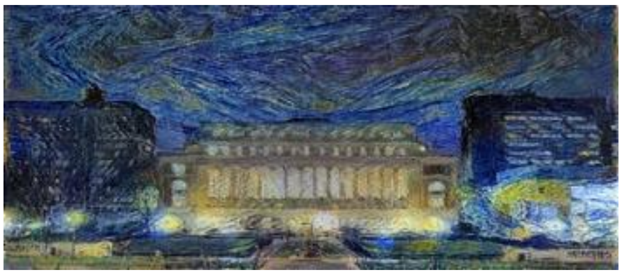

(H) [0,.5,.5,0]


(I) [0,.5,0,.5]


(J) [0,0,.5,.5]


(K) [1/3,1/3,1/3,0]


(L) [1/3,1/3,0,1/3]


(M) [1/3,0,1/3,1/3]

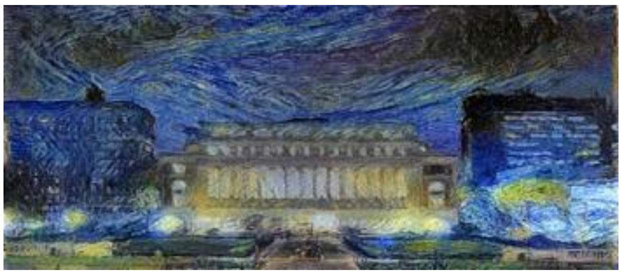

(N) [0,1/3,1/3,1/3]

Figure 4-10 Detailed results for the style of the painting

By comparing (A), (B), (C), and (D) in Fig. 4-10, we can find that layer 'block1_conv1' significantly change the picture color from darkish to bluish and make the lights a little dim, but it did not make much change on the building shape and the sky. We can still clearly see the library and the other buildings in the picture. From picture (B), 'block2_conv1' added some fine rippled patterns on sky, made the outline of the library and the building besides becomes more blurry, and made the lights much more dim. From picture (C), 'block3_conv1' added larger pattern on the sky and make the building shape even more unclear. From picture (D), 'block4_conv1' changed the picture by adding the most obvious rippled pattern on sky, blurring the building outline, and make the harsh lights dim, but overall, it kept the most original color and did not make the picture too bluish. In addition, from (A) to (D), input content image structures captured by the style representation increase.

So with the weight of layer become deeper, we can see that the feature of style transferred becomes more and more obvious. (A) only transferred color, (B) and (C) becomes more generalized and (D) with even more features if we compare to the baseline picture. We think the transformation in conv4, which is the deepest layer, has the best style transformation result. We conclude from the above analysis that the deeper the layer, the better style transformation generalization.

There is still other things interesting: if you go though lights and windows in the picture, you can find that windows and light in (A) and (D) is very clear. But in (B) and (C), especially (C), it is not. We think the conv2 and conv3 may have some important fuzzy-up affection of picture.

So let us go combination of weight. Basically, the combination of weigh is just like the merge of feature transformed by each layer. We can see the if weight fell on conv2 and conv3, the blurred affection is transformed which confirms our assumptions.

To summarize here, we have 3 conclusions:

1.The deeper the feature layer, the better style transformation performance.

2.Layer of conv2 and conv3 has a fuzzy-up affection

3.The combination of layer weight works just like putting different features of layers together.

5. References

[1] Wikipedia. Prisma (app) — wikipedia, the free encyclopedia, 2017.

[2] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. A Neural Algorithm of Artistic Style. arXiv.org,August 2015.

[3] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. Image Style Transfer Using Convolutional Neural Networks. In 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR, pages 2414–2423. IEEE, 2016.

[4] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. International Journal of Computer Vision, 115(3):211–252, December 2015.

[5] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. Deepface: Closing the gap to human-level performance in face verification. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 1701–1708, 2014.

[6] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In Proceedings of the 27th international conference on machine learning (ICML-10), pages 807–814, 2010.

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems, pages 1097–1105, 2012.

[8] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv.org, September 2014.

[9] University of Oxford Department of Engineering Science. Visual geometry group: Very deep convolutional networks for large-scale visual recognition, 2014.

6. Appendix

Github: https://github.com/shana9pm/ArtStyleTransfer

Code file description:

ArtStyleTransfer.py: including runtime option, neural network structure

Settings.py: director setting, picture settings, structure setting as feature weight settings

utils.py: terminal parsers, preprocess and postprocess of pictures and model building functions

ArtStyleTransfer.py

```
from utils import *
import tqdm
import time
from scipy.optimize import fmin_l_bfgs_b
from Settings import *
import keras.backend as K
import copy
import matplotlib.pyplot as plt
import pickle


def calculate_loss(outputImg):
    if outputImg.shape != (1, WIDTH, WIDTH, 3):
        outputImg = outputImg.reshape((1, WIDTH, HEIGHT, 3))
    loss_fcn = K.function([outModel.input], [get_total_loss(outModel.input)])
    return loss_fcn([outputImg])[0].astype('float64')


def get_total_loss(outputPlaceholder):
    F = get_feature_reps(outputPlaceholder,layer_names=[contentLayerNames],
model=outModel)[0]
    Gs = get_feature_reps(outputPlaceholder,layer_names=styleLayerNames,
model=outModel)
    contentLoss = get_content_loss(F, P)
    styleLoss = get_style_loss(ws, Gs, As)
    totalLoss = alpha*contentLoss + beta*styleLoss
    return totalLoss


def get_content_loss(F, P):
    if lossType=='SE':
        cLoss = 0.5*K.sum(K.square(F - P))
    else:
        cLoss= 0.5*K.sum(K.abs(F - P))
    return cLoss


def get_style_loss(ws, Gs, As):
    sLoss = K.variable(0.)
    for w, G, A in zip(ws, Gs, As):
        M_l = K.int_shape(G)[1]
        N_l = K.int_shape(G)[0]
        G_gram = get_Gram_matrix(G)
        A_gram = get_Gram_matrix(A)
        if lossType=='SE':
            sLoss+= w*0.25*K.sum(K.square(G_gram - A_gram))/ (N_l**2 * M_l**2)
        else:
            sLoss += w * 0.25 * K.sum(K.abs(G_gram - A_gram)) / (N_l  * M_l )
    return sLoss


def get_Gram_matrix(F):
    G = K.dot(F, K.transpose(F))
    return G


def get_grad(gImArr):
    """
    Calculate the gradient of the loss function with respect to the generated image
    """
    if gImArr.shape != (1, WIDTH,HEIGHT, 3):
        gImArr = gImArr.reshape((1, WIDTH,HEIGHT, 3))
    grad_fcn = K.function([outModel.input],
K.gradients(get_total_loss(outModel.input), [outModel.input]))
    grad = grad_fcn([gImArr])[0].flatten().astype('float64')
    return grad


def get_feature_reps(x,layer_names, model):


    featMatrices = []
    for ln in layer_names:
        selectedLayer = model.get_layer(ln)
        featRaw = selectedLayer.output
        featRawShape = K.shape(featRaw).eval(session=K.get_session())
        N_l = featRawShape[-1]
        M_l = featRawShape[1]*featRawShape[2]
        featMatrix = K.reshape(featRaw, (M_l, N_l))
        featMatrix = K.transpose(featMatrix)
        featMatrices.append(featMatrix)
    return featMatrices


def callbackF(Xi):
    """A call back function for scipy optimization to record Xi each step"""
    global iterator
    global count




    if record:
        deepCopy=copy.deepcopy(Xi)
        this_styleLoss = calculate_style_loss(deepCopy)
        this_contentLoss = calculate_content_loss(deepCopy)
        contentLossList.append(this_contentLoss)
        styleLossList.append(this_styleLoss)
        this_totalLoss=this_styleLoss*beta+this_contentLoss*alpha
        totalLossList.append(this_totalLoss)


    if iterator%rstep==0:
        deepCopy=copy.deepcopy(Xi)
        i = iterator // rstep
        xOut = postprocess_array(deepCopy)
        imgName = PATH_OUTPUT + '.'.join(name_list[:-1]) + '_{}.{}'.format(
            str(i) if i!=stop else 'final', name_list[-1])
        _ = save_original_size(xOut, imgName, contentOrignialImgSize)
    iterator+=1
    count.update(1)


"""The following functions are basically used for loss record"""


def get_style_loss_forward(outputPlaceholder):
    Gs = get_feature_reps(outputPlaceholder, layer_names=styleLayerNames,
model=outModel)
    styleLoss = get_style_loss(ws,Gs,As)
    return styleLoss


def get_content_loss_forward(outputPlaceholder):
    F = get_feature_reps(outputPlaceholder, layer_names=[contentLayerNames],
model=outModel)[0]
    contentLoss = get_content_loss(F,P)
    return contentLoss


def calculate_style_loss(Xi):
    if Xi.shape != (1, WIDTH, WIDTH, 3):
        Xi = Xi.reshape((1, WIDTH, HEIGHT, 3))
    loss_fcn = K.function([outModel.input], [get_style_loss_forward(outModel.input)])
    return loss_fcn([Xi])[0].astype('float64')


def calculate_content_loss(Xi):
    if Xi.shape != (1, WIDTH, WIDTH, 3):
        Xi = Xi.reshape((1, WIDTH, HEIGHT, 3))
    loss_fcn = K.function([outModel.input],
[get_content_loss_forward(outModel.input)])
    return loss_fcn([Xi])[0].astype('float64')


if __name__=='__main__':


    parser=build_parser()
    args=parser.parse_args()
    content_name=args.content
    style_name=args.style
    output_name=args.output
    iteration=int(args.iter)
    flw=int(args.flw)
    lossType=args.losstype
    record=False if args.record=='F' else True
    rstep=int(args.rstep)
    stop = iteration // rstep
    alpha=float(args.alpha)
    beta=float(args.beta)
    fromc=False if args.fromc=='F' else True
    continueTraining=False if args.continueTraining=='F' else True
    contentLossList=[]
    styleLossList=[]
    totalLossList=[]


    iterator = 1
    actualIteration=iteration


    #So continueTraining will override fromc
    if continueTraining:
        sList=pickle.load(open(PATH_CONTINUETRAINING+'sList.dat','rb'))
        totalLossList+=sList[0]
        contentLossList+=sList[1]
        styleLossList+=sList[2]
        iteration-=sList[3]
        output, outputPlaceholder = outImageUtils(WIDTH, HEIGHT)
        output=sList[4]
    # so iteration is the num we train this time actual iteration is what we
accumulate in training


    else:
        if fromc:
            output,
outputPlaceholder=outImageUtils2(PATH_INPUT_CONTENT+content_name,WIDTH,HEIGHT)
        else:
            output, outputPlaceholder = outImageUtils(WIDTH, HEIGHT)






    contentImgArr, contentOrignialImgSize =
inputImageUtils(PATH_INPUT_CONTENT+content_name, SIZE)
    styleImgArr, styleOrignialImgSize = inputImageUtils(PATH_INPUT_STYLE+style_name,
SIZE)
    contentModel, styleModel, outModel = BuildModel(contentImgArr, styleImgArr,
outputPlaceholder)


    P = get_feature_reps(x=contentImgArr,layer_names=[contentLayerNames],
model=contentModel)[0]
    As = get_feature_reps(x=styleImgArr,layer_names=styleLayerNames,
model=styleModel)
    ws = wlList[flw]


    try:
        outputImg = output.flatten()
    except:
        outputImg=output


    start=time.time
    count=tqdm.tqdm(total=iteration)
    name_list = output_name.split('.')
```

```
xopt, f_val, info= fmin_l_bfgs_b(calculate_loss, outputImg, fprime=get_grad,
                                  maxiter=iteration,disp=True,callback=callbackF)
if record:
    plt.plot(totalLossList)
    plt.xlabel('Iterations')
    plt.ylabel('Loss')
    plt.title('TotalLoss')
    plt.savefig(PATH_OUTPUT + 'TotalLoss.jpg')


    plt.figure()
    plt.plot(contentLossList)
    plt.xlabel('Iterations')
    plt.ylabel('Loss')
    plt.title('ContentLoss')
    plt.savefig(PATH_OUTPUT + 'ContentLoss.jpg')


    plt.figure()
    plt.plot(styleLossList)
    plt.xlabel('Iterations')
    plt.ylabel('Loss')
    plt.title('StyleLoss')
    plt.savefig(PATH_OUTPUT + 'StyleLoss.jpg')


sList=[totalLossList,contentLossList,styleLossList,actualIteration,xopt]
pickle.dump(sList,open(PATH_CONTINUETRAINING+'sList.dat','wb'))
```

Settings.py

```
PATH_INPUT_STYLE
= 'input/style/'
                   PATH_INPUT_CONTENT = 'input/content/'
                   PATH_OUTPUT = 'output/'
                   PATH_CONTINUETRAINING=PATH_OUTPUT
                   WIDTH=512
                   HEIGHT=512
                   SIZE=(512,512)




                   contentLayerNames = 'block4_conv2'
```

```
styleLayerNames = [
    'block1_conv1',
    'block2_conv1',
    'block3_conv1',
    'block4_conv1',
]


wlList=[
    [0.25, 0.25, 0.25, 0.25],#baseline
    [1, 0, 0, 0],
    [0, 1, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 1],
    [.5,.5,0, 0],
    [.5,0,.5, 0],
    [.5, 0,0,.5],
    [0,.5,.5,0],
    [0,.5,0,.5],
    [0,0,.5,.5],
    [1/3,1/3,1/3,0],
    [1/3,1/3,0,1/3],
    [1/3,0,1/3,1/3],
    [0,1/3,1/3,1/3],
]
```

utils.py

```
from argparse
import
ArgumentParser
from PIL import Image
from keras.preprocessing.image import load_img, img_to_array
import keras.backend as K
from keras.applications.vgg16 import preprocess_input
import numpy as np
from keras.applications import VGG16
from Settings import *


def build_parser():
    parser = ArgumentParser()
    parser.add_argument('--content', dest='content', required=True,
                        help='Content image, e.g. "input.jpg"')
    parser.add_argument('--style', dest='style', required=True,
                        help='Style image, e.g. "style.jpg"')
    parser.add_argument('--output', dest='output', required=True,
                        help='Output image, e.g. "output.jpg"')
    parser.add_argument('--iter',dest='iter', required=False,default=600,
                        help='Iteration with default and suggested 600 Better
to be multiple of 50')
    parser.add_argument('--record', dest='record', required=False,
default='F',
                        help='Record loss or not,T for record')
    parser.add_argument('--flw', dest='flw', required=False, default='0',
                        help='Feature weight selected ')
    parser.add_argument('--lt', dest='losstype', required=False,
default='SE',
                        help='Loss type selected ')
    parser.add_argument('--rstep', dest='rstep', required=False,
default='50',
                        help='Record picture per step')
    parser.add_argument('--alpha', dest='alpha', required=False,
default='1.0',
                        help='alpha')
    parser.add_argument('--beta', dest='beta', required=False,
default='10000.0',
                        help='alpha')
    parser.add_argument('--fromc', dest='fromc', required=False, default='F',
                        help='The output image is from content is
initialization')
    parser.add_argument('--cont', dest='continueTraining', required=False,
default='F',
                        help='Activation of continuos training mode')
    return parser







def inputImageUtils(imagePath,size):
    """
    Dealing input image
    Return Arrayed Image and original size
    """
    rawImage=Image.open(imagePath)
    rawImageSize=rawImage.size
    image=load_img(path=imagePath,target_size=size)
    ImageArray=img_to_array(image)
    ImageArray=K.variable(preprocess_input(np.expand_dims(ImageArray,
axis=0)), dtype='float32')
```

```
    return ImageArray,rawImageSize


def outImageUtils(width,height):
    """
    Initialize image and our target image
    Return Initialized Image and Placeholder for calculation
    """
    output=np.random.randint(256, size=(width, height, 3)).astype('float64')
    output = preprocess_input(np.expand_dims(output, axis=0))
    outputPlaceholder=K.placeholder(shape=(1, width,height, 3))
    return output,outputPlaceholder


def outImageUtils2(imagePath,width,height):
    """
    Initialize image from contentImg
    """
    img=Image.open(imagePath)
    img=img.resize((width,height))
    imgarr=np.array(img)
    output=preprocess_array(imgarr)
    outputPlaceholder=K.placeholder(shape=(1,width,height,3))
    return output,outputPlaceholder


def save_original_size(x,path, target_size):
    """
    Save output image as its original size
    """
    xIm = Image.fromarray(x)
    xIm = xIm.resize(target_size)
    xIm.save(path)
    return xIm


def BuildModel(contentImgArr,styleImgArr,outputPlaceholder):
    contentModel = VGG16(include_top=False, weights='imagenet',
input_tensor=contentImgArr)
    styleModel = VGG16(include_top=False, weights='imagenet',
input_tensor=styleImgArr)
    outModel = VGG16(include_top=False, weights='imagenet',
input_tensor=outputPlaceholder)
    return contentModel,styleModel,outModel


def postprocess_array(x):
    # Zero-center by mean pixel
    if x.shape != (WIDTH, HEIGHT, 3):
        x = x.reshape((WIDTH,HEIGHT, 3))
    x[..., 0] += 103.939
    x[..., 1] += 116.779
    x[..., 2] += 123.68
    # 'BGR'->'RGB'
    x = x[..., ::-1]
    x = np.clip(x, 0, 255)
    x = x.astype('uint8')
    return x


def preprocess_array(x):
    if x.shape != (WIDTH, HEIGHT, 3):
        x = x.reshape((WIDTH,HEIGHT, 3))
    # RGB -> BGR
    x = x.astype('float64')
    x = x[:, :, ::-1]


    x[:, :, 0] -= 103.939
    x[:, :, 1] -= 116.779
    x[:, :, 2] -= 123.68
    return x
```