Comparison of Different Neural Network Architectures for Neural Transfer Learning

STAT GR5242
Advanced Machine Learning
Final Project

Chui Kong (ck2964)
Runfeng Tian (rt2755)
Qihang Yang (qy2231)
Johnson Zhang (zz2677)

Department of Statistics, Columbia University

Group 36: Team Ganondorf

Contents

1 Introduction **2**

2 Project Objective **3**

3 Data Preparation **3**

4 Methodology **3**

5 Result **5**

5.1 Iteration Numbers 5

5.2 Random Seeds 6

5.3 Hyper-parameter Impacts 6

5.3.1 Style and Content Weight of Perceptual Loss Function (α and β) 6

5.3.2 Noise Ratio 7

5.3.3 Learning Rate for Perceptual Loss Optimizers 7

6 Comparisons Between Different Neural Networks Architectures **7**

7 Conclusion **9**

7.1 Main Conclusion 9

7.2 Limitations and Future Works 10

References **11**

Appendix - Key Output Images **12**

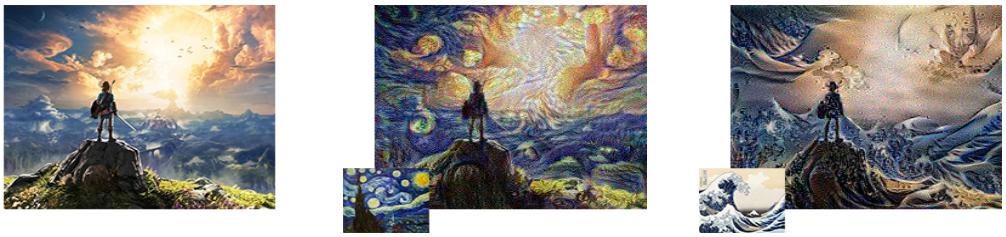
Appendix - Codes **16**

1 Introduction

If given an image, how can we convert another image into the given image's style while retaining the content of that image as much as possible. The traditional method is to perform analysis for a particular image style and build a statistical model for the style transfer. Such models have achieved good results, but it has a main flaw: a model can only implement the style transfer for one image style. Therefore, the application of such models of style transfer is very limited.[1]

With the development of neural networks and deep learning, machines in certain areas of visual perception, such as object detection and face recognition, have performed close or even beyond human beings. Meanwhile, with such deep learning architectures, we can not only perform traditional classification and regression tasks, but also implement image style transfer while retaining the content of the original image. Gatys[2] used image representations derived from CNNs optimised for object recognition to make high level image information explicit. They proposed a neural algorithm to separate and recombine the image content with style of the original image to produce a new combined image with high perceptual quality. Roman Novak[3] improved Gatys' Neural Algorithm by modifying the style representation including layers and weight settings and activation functions. Gatys[4] also improved the neural transfer algorithm for transferring style while preserving colors. In this work, they applied two different approaches (linear transformation and luminance channel transfer) to retain the content color while converting the content style into target style. Dmitry et al[5] proposed a feed forward-based method to move the computational burden to a learning stage. This makes the textures generation faster and the quality is comparable to the previous work.

In our work, we built a neural transfer system based on Gatys[2]'s work and compared the performance of this system under different hyper-parameter settings as well as different neural networks architecture. We will discuss the project objective details in the next section.



(a) Zelda (b) Zelda+Starry (c) Zelda+Kanagawa

Figure 1: Zelda and Outputs

(a) Snow (b) Snow+Starry (c) Snow+Kanagawa

Figure 2: Snow and Outputs

2 Project Objective

In this project, we built an end-to-end neural transfer system to experiment the desired results as required. By declaring the pre-trained neural network architecture, relevant parameters(e.g. α, β and noise ratio etc.) and the content image as well as the style image we would like to combine, the system would return the combined images with corresponding loss under different epochs. With this neural transfer system, we can test the results under different parameters settings as well as neural network architectures.

The hyper-parameters we studied in this projects are weights of loss & style functions(α and β), noise ratio for the content pictures(θ), learning rate for the gradient descent optimizer(η, we used Stochastic gradient descent as optimizer) We also observed the results under different iterations and random seeds. We compared the results between different neural networks architectures including `VGG-19`, `ResNet50`, `Xception` in our neural transfer system.

3 Data Preparation

Since we are using pre-trained neural network architectures to extract features and represent the image style as well as the contents, we do not need extra image dataset for neural network training.

However, we collected combinations of style images and content images for neural style transfer. The style images contain world famous paintings, mainly Impressionnisme works, we chose two different style for style pictures: The Starry Night (Vincent van Gogh, 1889, short for Starry)[Figure 3] and Great Wave off Kanagawa (Katsushika Hokusai, 1832, short for kanagawa)[Figure 4]. We chose them for their impressive texture and different color so that we can have a clearer comparison of the results. The two content images are a snow scene of Flushing photoed by one of our group members [Figure 5] and a CG picture of Zelda: Breath of the Wild[Figure 6]. One of the content images is a real photo and another one is taken from our group's favourite game. We would like to convert the painting styles into our selected content images.

Figure 3: The Starry Night, 1889

Figure 4: Great Wave off Kanagawa, 1832

4 Methodology

We mainly referred to the neural style transfer algorithm described in the paper[3]. In particular, we used Optimization-based method.

Firstly, we input a base map of random noise ϵ_{ijk} (i is the range from 1 to height of the random noise map, j is the range from 1 to width of the random noise map, k is the range from 1 to channel

Figure 5: Snow scene at Flushing

Figure 6: Zelda: Breath of Wild

of the random noise map. We set different random seeds to represent the influence of randomness). Then we calculated style loss as well as content loss with corresponding loss function and optimized the combined weighted loss function(which is called perceptual loss) iteratively with respect to the gradient of the base map until the weighted loss decreased to a certain stage (its style and texture are similar to the style image while the content is similar to the content image). Finally the ϵ_{ijk} is updated by the gradient descent process in terms of the perceptual loss. In our project, `VGG-19` is the first to be applied to the system, then we tried `ResNet50` and `Xception` to transfer two different painting styles into the content images for further comparison.

We would like to introduce the hyper-parameters such as noise ration, content loss function as well as style loss function in the neural style transfer system as follows:

Initialize Generate Image G**:** $G = \theta \times \epsilon + (1 - \theta) \times C$

- θ denote noise ratio
- ϵ is an initialized noise image. Each pixel $\epsilon_{ijk}(i = 1, \ldots, n_h, j = 1, \ldots, n_w, k = 1, \ldots, n_c)$ are initialized using uniform distirbution $U(-10, 10)$. ϵ's are the model parameters.
- C is the image array of content image.
- S is the image array of style image.

Content Cost Function $J_{content}(C, G)$**:**

$$J_{content}(C, G) = \frac{1}{4 \times n_H \times n_W \times n_C} \sum_{\text{all entries}} (a^{(C)} - a^{(G)})^2$$

- Here, n_H, n_W and n_C are the height, width and number of channels of the hidden layer you have chosen, and appear in a normalization term in the cost.

Style Cost Function $J_{stype}(S, G)$**:**

$$J^{[l]}_{style}(S, G) = \frac{1}{4 \times {n_C}^2 \times (n_H \times n_W)^2} \sum_{i=1}^{n_C} \sum_{j=1}^{n_C} (G^{(S)}_{(gram)i,j} - G^{(G)}_{(gram)i,j})^2$$

- $G^{(S)}_{gram}$: Gram matrix of the "style" image.
- $G^{(G)}_{gram}$: Gram matrix of the "generated" image.

Layer Weights:

By default, we'll give each layer equal weight, and the weights add up to 1. $(\sum_l^L \lambda^{[l]} = 1)$

Thus, we combine the style loss for different layers as follows:

$$J_{style}(S,G) = \sum_l \lambda^{[l]} J^{[l]}_{style}(S,G)$$

Finally, we defined the total cost function that minimizes both the style and the content cost as follow:

$$J(G) = \alpha J_{content}(C,G) + \beta J_{style}(S,G)$$

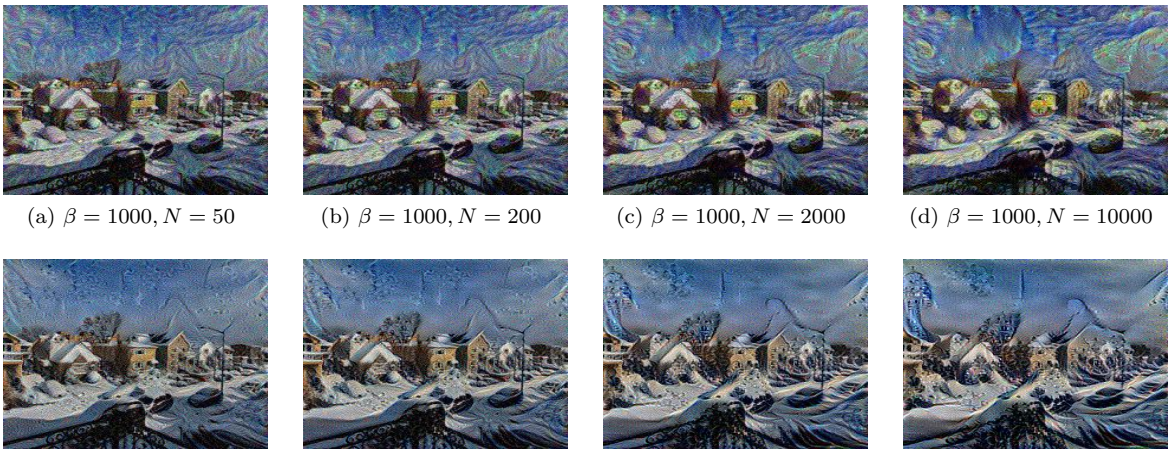
In our project, we mainly studied the following hyper-parameters: weights for the total cost function (α and β), noise ration (θ) for base map of random noise, learning rate(η) for the gradient descent optimizer, iteration number (N) and random seeds. We tuned the hyper-parameters for three different neural networks architectures (`VGG-19`, `ResNet50` and `Xception`). We would show the different results under different hyper-parameter settings for `VGG-19` while comparing the results between the other two architectures under best tuned parameters.

5 Result

By implementing the neural style transfer method in section 4 with `VGG-19`, we obtained 4 generated images corresponding to 4 possible combinations of content and style images: Zelda with Starry, Zelda with Kanagawa, Snow with Starry, and Snow with Kanagawa. Each combination has been trained under different content and style weights settings (fixed $\alpha = 10$, set $\beta = 100$, 500, 1000 and 10000, $\beta = 500$ with different seeds). In the appendix, for each α, β setting of one content style combination, we presented 4 images of iterating numbers: 50, 200, 2000, 10000. In the following subsections, the impact of iteration number as well as α and β setting of total cost function will be discussed.

5.1 Iteration Numbers

Due to space limitation, in Figure 7, we only show the generated image of 2 combinations: Snow with Kanagawa and Snow with Starry under the hyper-parameter setting: $\alpha = 10$ and $\beta = 1000$.



(a) $\beta = 1000, N = 50$ (b) $\beta = 1000, N = 200$ (c) $\beta = 1000, N = 2000$ (d) $\beta = 1000, N = 10000$

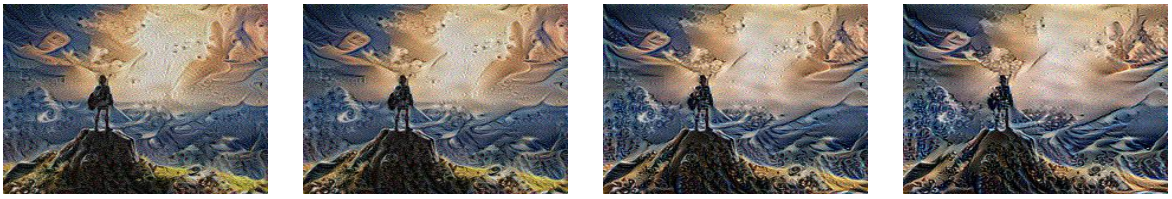
(e) $\beta = 1000, N = 50$ (f) $\beta = 1000, N = 200$ (g) $\beta = 1000, N = 2000$ (h) $\beta = 1000, N = 10000$

Figure 7: Impacts of iteration number for neural style transfer

We can find that in the first row of Figure 7, the house in the snow image is gradually distorted closer to starry-like textures and some yellow color is shown on it. Similarly, in the second row, we can see that as the number of training iterations increases, the snow is gradually turned into waves similar to what it is in the Great Wave off Kanagawa. In general, with more iterations, the image will get closer and closer to style in terms of color and texture.

5.2 Random Seeds

We selected two seeds of noise images on the 4 pairs of content and style images under the setting: $\alpha = 10$ & $\beta = 500$. Figure 8 shows the result of generated images over one pair of content and style images: Zelda with Kanagawa. The seeds of the first row and 2nd row are different. In fact, from the Figure 8, the generated images of 2 different seeds settings do not make a significant difference.



(a) $\beta = 500, N = 50$ (b) $\beta = 500, N = 200$ (c) $\beta = 500, N = 2000$ (d) $\beta = 500, N = 10000$

(e) $\beta = 500, N = 50$* (f) $\beta = 500, N = 200$* (g) $\beta = 500, N = 2000$* (h) $\beta = 500, N = 10000$*

Figure 8: Impacts of random seeds for neural style transfer

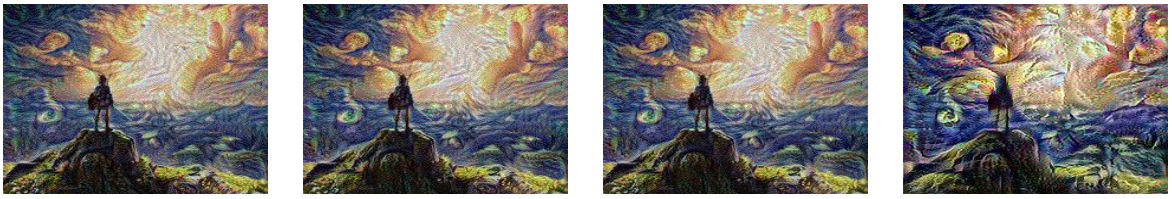
5.3 Hyper-parameter Impacts

5.3.1 Style and Content Weight of Perceptual Loss Function (α and β)

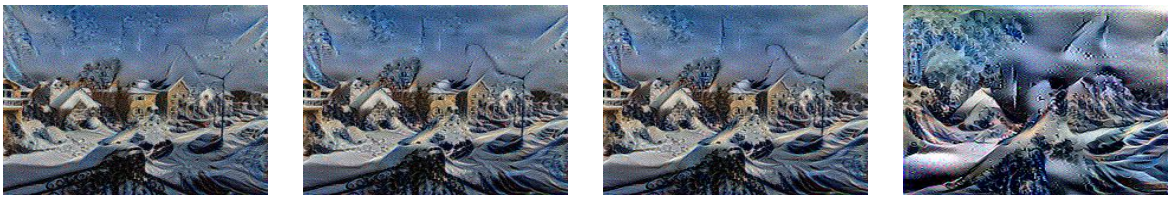
As has been mentioned before, we fixed the content weight and then tuned the style weight to learn how content and style weights influence the final visual impression of the generated image. In Figure 9, each column of images is generated from one pair of content and style images.

By comparing each column, one can clearly find that the bigger style weight is, the heavier painting styles are applied to the content images. If β is too low, the art style of the style image can hardly be embodied in the generated image (etc. first row of Figure 9). On the other hand, if β is too large, the original content image will be severely covered by the style image. For example, in the last column of Figure 9), the snow image is almost converted into a wave image, and one can hardly find it is generated by the content image, and the character in the zelda image is blurred due to the heavy style in Starry we apply to it.

Hence, to achieve a fantastic visual impression on the generated image, it is important to fine tune style & content weight in an appropriate range. For `VGG-19` net, we find that $\beta = 10$ and $\beta = 1000$, i.e. the ratio of content weight to style weight is 1e-2 seems to be a good choice.



(a) $\beta = 100, N = 2000$ (b) $\beta = 500, N = 2000$ (c) $\beta = 1000, N = 2000$ (d) $\beta = 10000, N = 2000$



(e) $\beta = 100, N = 2000$ (f) $\beta = 500, N = 2000$ (g) $\beta = 1000, N = 2000$ (h) $\beta = 10000, N = 2000$

Figure 9: Impacts of weights of total cost function for neural style transfer

5.3.2 Noise Ratio

We test different noise ratio values θ on different neural networks, and find that when θ is set around 0.2, style can be normally applied to the content image. When the ratio is too low, the style cannot be well captured. On the opposite, when the noise ratio is too large (over than 0.25), the generated image becomes extremely blurred or even turns into noise.

Such result can be explained by the optimization process: as the noise ϵ_{ijk} itself are the parameters we need to train in the neural transfer model, if the noise ratio is too high, it is very likely that ϵ_{ijk} stuck into a bad local optimal leading to the generation of a noise image instead. However, if the noise ratio is too low, even though ϵ_{ijk} converges into the right direction, it can hardly affect the final generated image since the final result still highly depends on the original content image.

5.3.3 Learning Rate for Perceptual Loss Optimizers

Learning rate η is another important hyperparameter for our system as it decides the total runtime of the training process to generate images to a large extent: if the learning rate η is too low, it may take a very long time to finish the training. However if it is too high, the gradient of loss will explode and leads to the training failure. Moreover, depending on the input images and pre-trained model architecture, the learning rate is different. For `VGG-19` model we generally set η to 0.001 and it takes approximately 1200 seconds to finish the N=10000 training process.

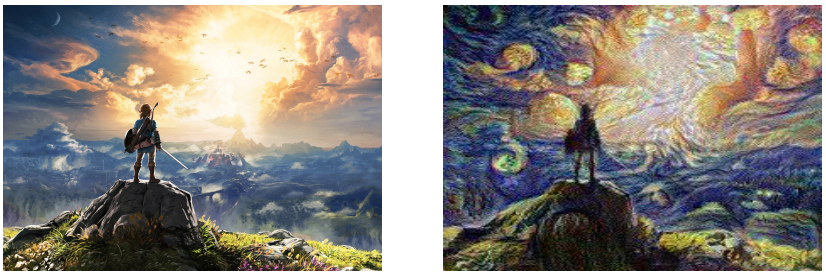
6 Comparisons Between Different Neural Networks Architectures

Besides `VGG-19`, we also applied other neural architectures for the neural style transfer. The architectures we used are `Xception` and `ResNet50`. Before `Xception`, we also tried Inception V3[6] to do the neural style transfer, but after a lot of adjustments to the parameters, we were still not able to reach satisfactory results. Our preliminary analysis on this is that for each CNN module in the inception network, it is only responsible for part of the feature training, which makes it hard for our

neural style transfer system to effectively capture the style representation of the whole original style image.

We used `Xception`[7] instead. `Xception` applied depth-wise separable convolution to reduce network complexity and speed up training process. In this case, we can obtain more style information from the selected layers. We also tried `ResNet50`[8] for our neural transfer system. `ResNet50` refers to the `VGG-19` network and adds residual models for shortcut connections. It performs better in the image classification than the pain CNN architecture like VGG and Inception. We selected different layers for content and style learning among different neural networks structures and set α=10, β = 50 million in `Xception` and α=10, β = 500 thousand in `ResNet50` for comparisons.

In the picture(Figure 10) transferred from Zelda to Starry using `Xception`. Compared with `ResNet50`, the vortex is more obvious and the texture is clearer. Compared with `VGG-19`, the output of `Xception` pays more attention to details, but the overall performance of `VGG-19` is more wild and obvious.



(a) Original content image: Zelda

(b) `VGG-19`: β=1000, N = 10000

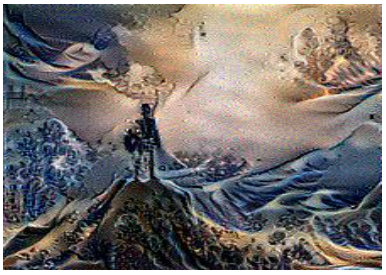
(c) `Xception`: β=50M, N = 10000

(d) `ResNet50`: β=500k, N = 10000

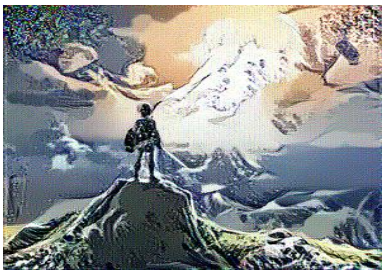
Figure 10: `VGG-19` VS `Xception` VS `ResNet50` on Zelda+Starry

The sunlight part of the image(Figure 11) transferred from Zelda to Kanagawa using `Xception` gets the texture of the waves instead of directly turning into a large area of white in the outputs `ResNet50`. It works better than the output of using `VGG-19` on the 10000 epoch.

(a) Original content image: Zelda



(b) `VGG-19`: β=1000, N = 10000



(c) `Xception`: β=50M, N = 10000

(d) `ResNet50`: β=500k, N = 10000

Figure 11: `VGG-19` VS `Xception` VS `ResNet50` on Zelda+Kanagawa

In our experiment, it can be seen from the outputs that compared to `Xception` and `ResNet50`, `VGG-19`'s output is heavier transferred, which shows that `VGG-19` is good at transferring the overall shape, and can be faster to see the significant transfer given the simpler layer selection process in simpler architecture.

Moreover, the `ResNet50` is less sensitive to beta(weight of style loss) than `VGG-19` and is more detailed than `VGG-19` in describing the texture and more sensitive to warm colors than `Xception`. But the overall shape of `ResNet50` is not satisfactory, and more training epochs are needed to see the significant transfer.

The architecture of `Xception` is less sensitive to beta compared to `VGG-19` and `ResNet50`, which means that it needs larger $\beta(10^5/10^2$ times of that in `VGG-19`/`ResNet50`), but the details are more refined even if it takes more epoch to get the significant style transfer(but less than that of `ResNet50`), especially in the transfer of Kanagawa Image to Zelda Image, it achieves good results in the sunlight part of the image.

7 Conclusion

7.1 Main Conclusion

In this project, we built a neural style transfer system and trained this system under different parameters setting and neural networks architectures. We compared the results based on the observation of color and texture on the output architectures.

For the comparison of neural networks architectures with fine tuned parameters, we found that `VGG-19` outperformed the other two compared architectures in overall style transfer. We can observe clear color and texture transfer from the style image to the content image by `VGG-19`. `Xception` has a sharper color retained while the texture is more blurred than `VGG-19`. `ResNet50`'s texture is rougher

than the other two neural architectures and it's hard for us to observe clear style transfer from the style image. Given the facts that the three models are well tuned, we can draw the conclusion that the neural style transfer systems perform better when the neural networks architectures can obtain more overall information of the style images. That's why `VGG-19` performs better than the other two neural networks architectures though the other two have more capabilities in other CV tasks like image classification. Also this explains why we need to choose the layers closer to the front.

For the hyper-parameter setting part, we used `VGG-19` as a baseline model. We found that higher iteration numbers can significantly influence the outcome while setting different random seeds has little effect to the final results. With respect to the hyper-parameters, we found that the setting α =10, β = 500, θ(noise ratio) = 0.2 works well in our system. We need to tune the learning rate based on the input images as well as neural architectures.

One of the challenges we face is to select the neural layers for content and style retaining. We need to choose layer combinations for style loss and content loss. Through reference to relevant literature and our practical experience,we found that it is better to choose the layers closer to the front and the number of layers do not exceed 5.

7.2 Limitations and Future Works

There are two factors that prevent us from making a comprehensive assessment for the effect of our neural style transfer like other computer vision tasks.

1. The final total loss cannot be treated as a standard measurement to measure the effect of neural style transfer. We evaluated the results based on subjective evaluation.

2. Due to the problems mentioned above, we can only judge the quality of transfer subjectively. Given that the insignificant results seen in the early stage do not mean that the training effect is not good, we have to train several thousands of epochs to see the results each time. In addition, each training takes 20 minutes, which causes the time cost of our parameter adjustment to be too high.

If we can make more refined screening on different architectures of different kinds of images' pairs, our style transfer may achieve better results.

- We can try more combinations of architecture and layer under different hyperparameters.

- Given that our content image and style image do not meet color retain problems, we can try more image pairs adding color retainments.

References

[1] Momodel. Style transfer-a neural network algorithm for artistic stylization. `https://juejin.cn/post/6844903889037180936#heading-13`.

[2] L. A. Gatys, A. S. Ecker, and M. Bethge. Image style transfer using convolutional neural networks. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2414–2423, 2016.

[3] R. Novak and Y. Nikulin. Improving the neural algorithm of artistic style. *arXiv preprint arXiv:1605.04603*, 2016.

[4] L. A. Gatys, M. Bethge, A. Hertzmann, and E. Shechtman. Preserving color in neural artistic style transfer. *arXiv preprint arXiv:1606.05897*, 2016.

[5] D. Ulyanov, V. Lebedev, A. Vedaldi, and V. Lempitsky. Texture networks: Feed-forward synthesis of textures and stylized images. *arXiv preprint arXiv:1603.03417*, 2016.

[6] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer. *CoRR, abs/1512.00567*, 2015.

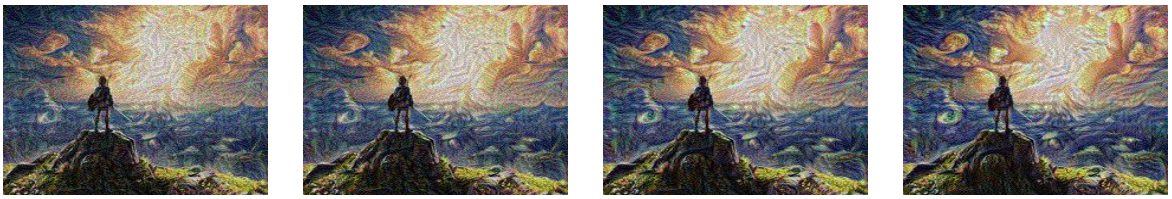
[7] F. Chollet. Xception: Deep learning with depthwise separable convolution. *arXiv preprint arXiv:1610.02357*, 2016.

[8] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

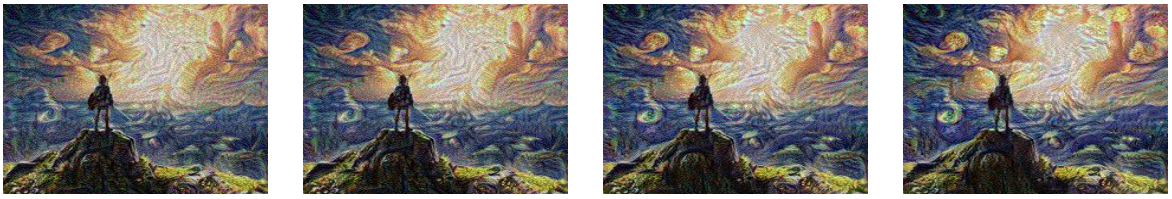
[9] Our Github.

Appendix - Key Output Images

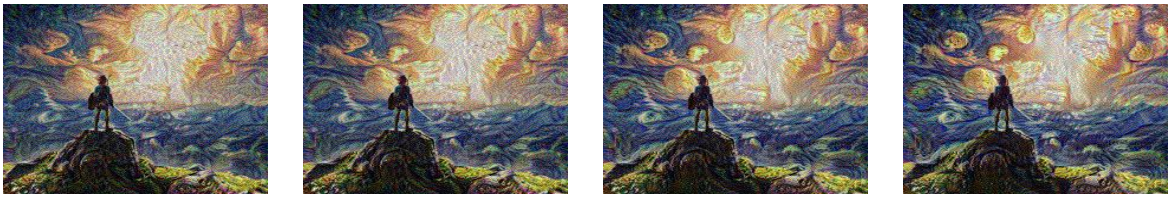
Our key output images shows below. In the images, * means that we use a new seed.



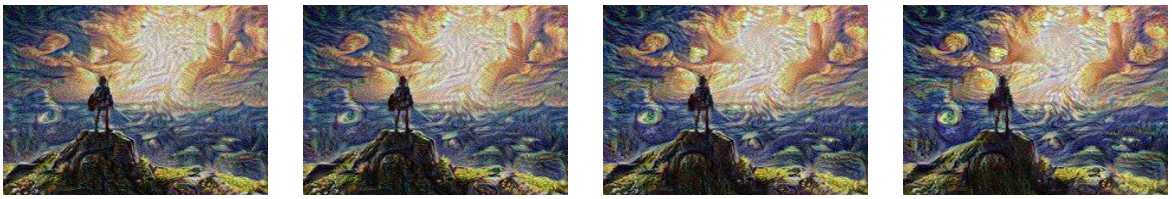
(a) $\beta = 100, N = 50$ (b) $\beta = 100, N = 200$ (c) $\beta = 100, N = 2000$ (d) $\beta = 100, N = 10000$



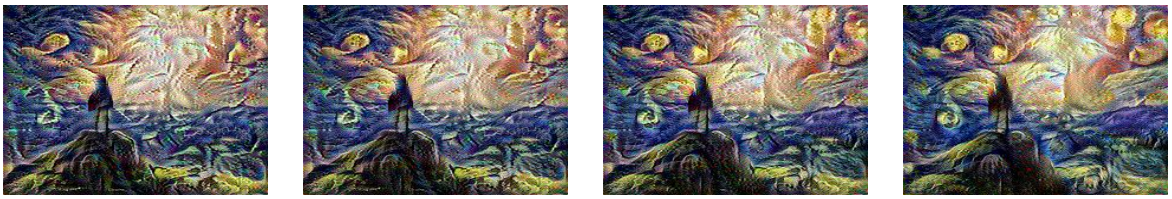
(e) $\beta = 500, N = 50$ (f) $\beta = 500, N = 200$ (g) $\beta = 500, N = 2000$ (h) $\beta = 500, N = 10000$



(i) $\beta = 500, N = 50$* (j) $\beta = 500, N = 200$* (k) $\beta = 500, N = 2000$* (l) $\beta = 500, N = 10000$*

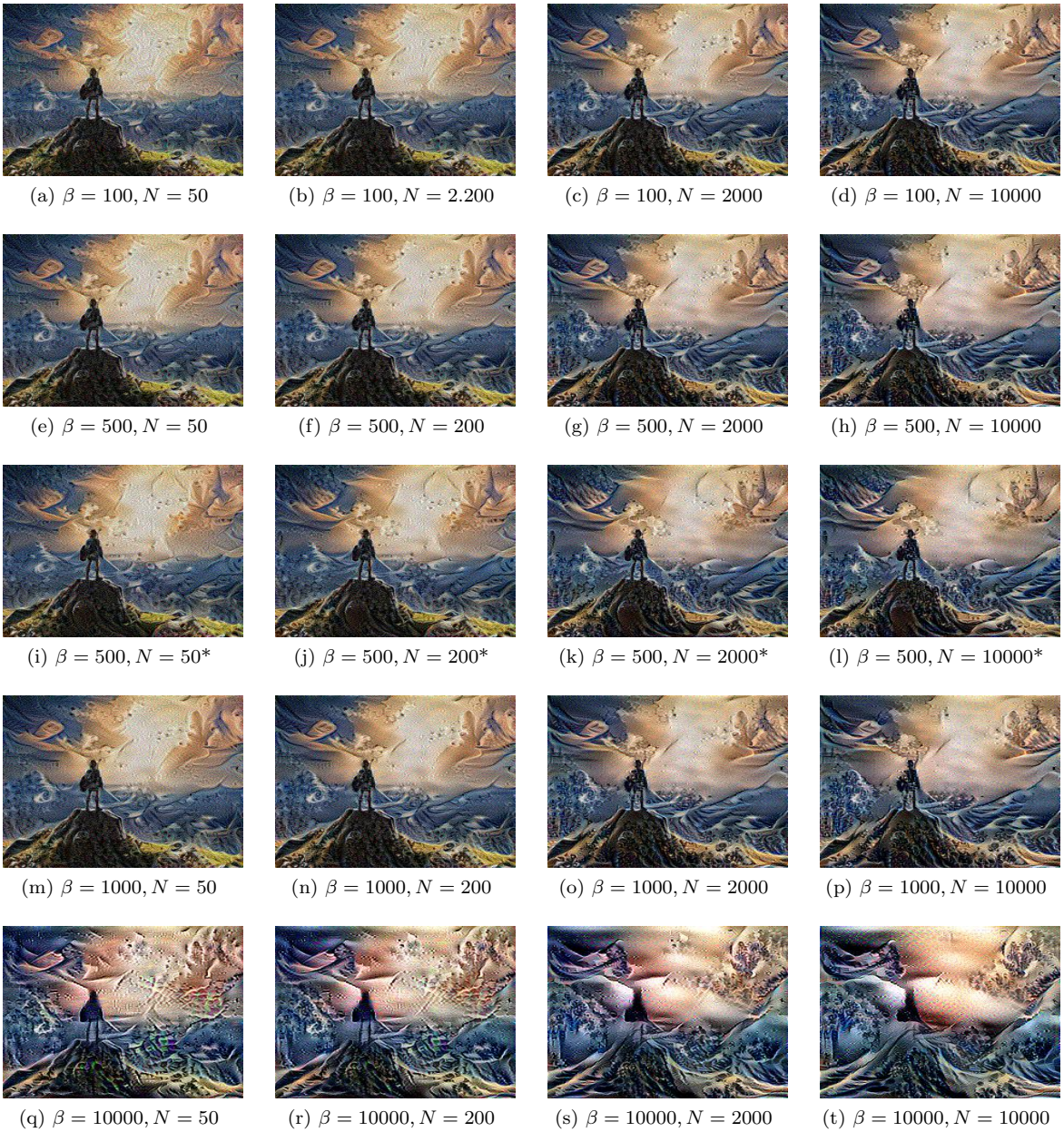


(m) $\beta = 1000, N = 50$ (n) $\beta = 1000, N = 200$ (o) $\beta = 1000, N = 2000$ (p) $\beta = 1000, N = 10000$



(q) $\beta = 10000, N = 50$ (r) $\beta = 10000, N = 200$ (s) $\beta = 10000, N = 2000$ (t) $\beta = 10000, N = 10000$

Figure 12: Zelda+Starry images



(a) $\beta = 100, N = 50$ (b) $\beta = 100, N = 2.200$ (c) $\beta = 100, N = 2000$ (d) $\beta = 100, N = 10000$

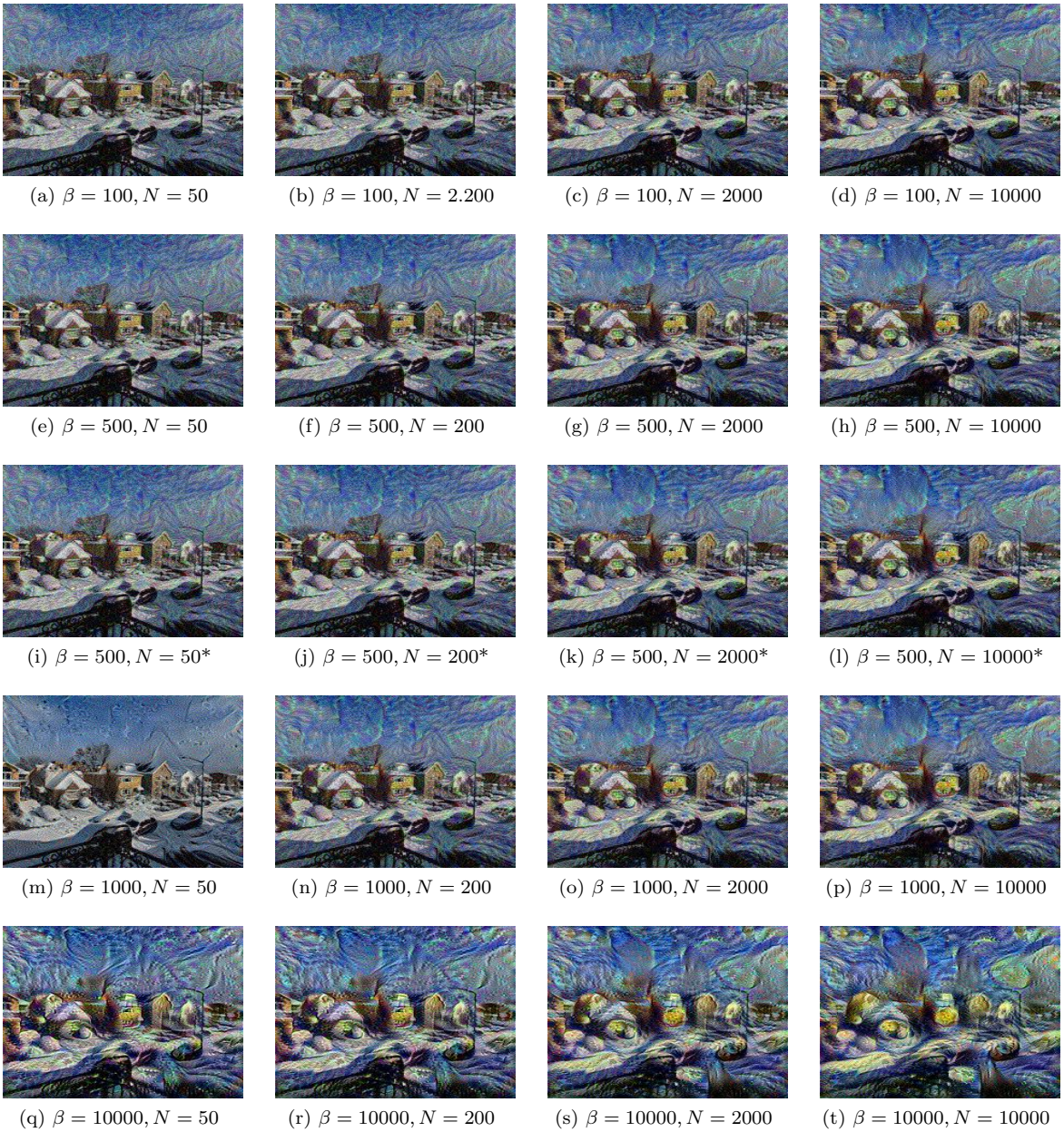
(e) $\beta = 500, N = 50$ (f) $\beta = 500, N = 200$ (g) $\beta = 500, N = 2000$ (h) $\beta = 500, N = 10000$

(i) $\beta = 500, N = 50$* (j) $\beta = 500, N = 200$* (k) $\beta = 500, N = 2000$* (l) $\beta = 500, N = 10000$*

(m) $\beta = 1000, N = 50$ (n) $\beta = 1000, N = 200$ (o) $\beta = 1000, N = 2000$ (p) $\beta = 1000, N = 10000$

(q) $\beta = 10000, N = 50$ (r) $\beta = 10000, N = 200$ (s) $\beta = 10000, N = 2000$ (t) $\beta = 10000, N = 10000$

Figure 13: Zelda+Kanagawa images



(a) $\beta = 100, N = 50$ (b) $\beta = 100, N = 2.200$ (c) $\beta = 100, N = 2000$ (d) $\beta = 100, N = 10000$

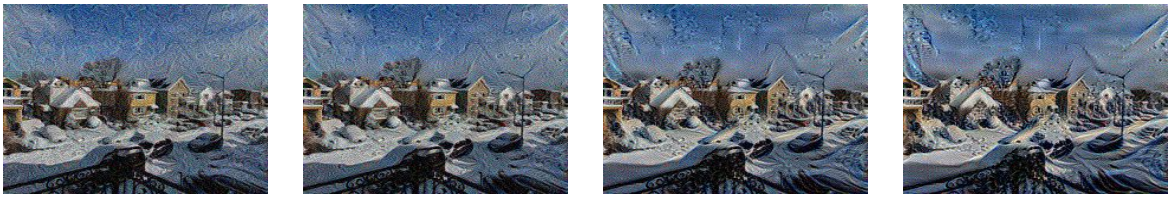
(e) $\beta = 500, N = 50$ (f) $\beta = 500, N = 200$ (g) $\beta = 500, N = 2000$ (h) $\beta = 500, N = 10000$

(i) $\beta = 500, N = 50$* (j) $\beta = 500, N = 200$* (k) $\beta = 500, N = 2000$* (l) $\beta = 500, N = 10000$*

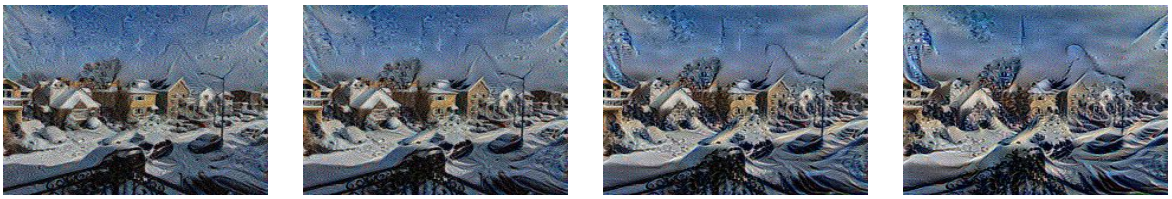
(m) $\beta = 1000, N = 50$ (n) $\beta = 1000, N = 200$ (o) $\beta = 1000, N = 2000$ (p) $\beta = 1000, N = 10000$

(q) $\beta = 10000, N = 50$ (r) $\beta = 10000, N = 200$ (s) $\beta = 10000, N = 2000$ (t) $\beta = 10000, N = 10000$

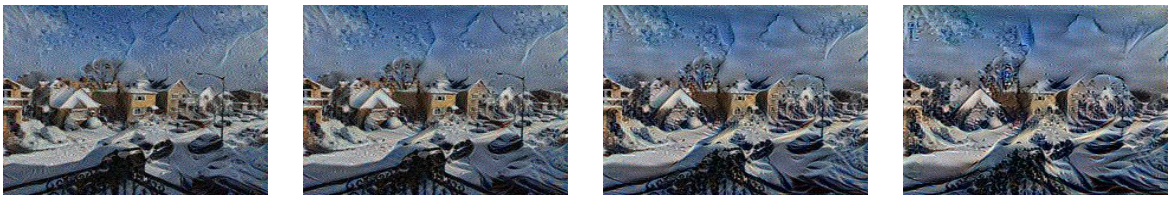
Figure 14: Snow+Starry images



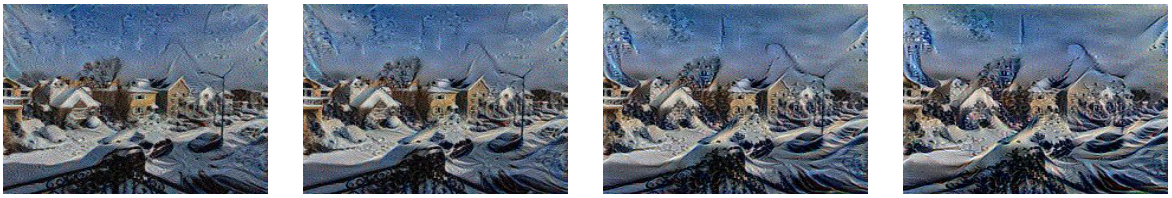
(a) $\beta = 100, N = 50$ (b) $\beta = 100, N = 200$ (c) $\beta = 100, N = 2000$ (d) $\beta = 100, N = 10000$



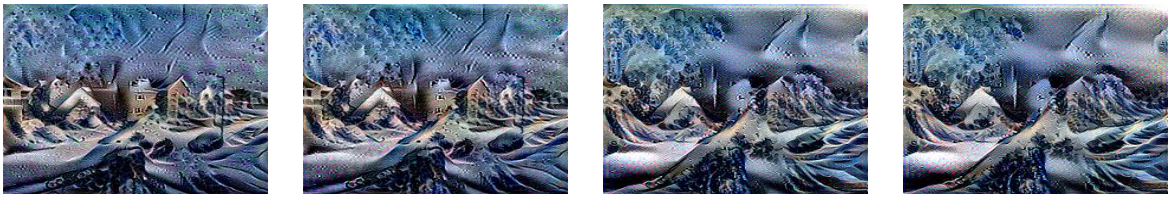
(e) $\beta = 500, N = 50$ (f) $\beta = 500, N = 200$ (g) $\beta = 500, N = 2000$ (h) $\beta = 500, N = 10000$



(i) $\beta = 500, N = 50$* (j) $\beta = 500, N = 200$* (k) $\beta = 500, N = 2000$* (l) $\beta = 500, N = 10000$*



(m) $\beta = 1000, N = 50$ (n) $\beta = 1000, N = 200$ (o) $\beta = 1000, N = 2000$ (p) $\beta = 1000, N = 10000$



(q) $\beta = 10000, N = 50$ (r) $\beta = 10000, N = 200$ (s) $\beta = 10000, N = 2000$ (t) $\beta = 10000, N = 10000$

Figure 15: Snow+Kanagawa images

Appendix - Codes

Our whole codes are published on our Github. [9]

The code of model class shows below.

```
1
2  # Define the neural_transfer model class
3
4  import numpy as np
5  import tensorflow as tf
6  import PIL
7  import matplotlib.pyplot as plt
8  import time
9
10
11 class neural_transfer():
12
13     def __init__(self,model):
14         self.img_model = model
15         self.input_shape = tuple(model.input.shape[1:])
16         self.img_encoder = None
17         self.W = None
18
19     # mathod to process image
20     def __get_image(self,image_path):
21         n_h, n_w, n_c = self.input_shape
22         image = PIL.Image.open(image_path)
23         image_array = np.asarray(image.resize((n_h,n_w)))/255.0
24         image_array = image_array.reshape((1,n_h,n_w,n_c))
25         return image_array
26
27     #save generated image after training.
28     def __save_image(self, path, image):
29         image = np.clip(image[0]*255, 0, 255).astype('uint8')
30         image = PIL.Image.fromarray(image, 'RGB')
31         image.save(path)
32         return
33
34
35     #plot the image after processing
36     def show_image(self, image_path):
37         image = self.__get_image(image_path)
38         plt.imshow(image[0])
39         return
40
41
42     #get the encoder model from pretrained model
43     def get_img_encoder(self,content_layer , style_layers):
44         self.img_model.trainable = False
45         n = len(style_layers)
46         self.style_layer_weights = [1/n]*n
47         output_layer_C = self.img_model.get_layer(content_layer).output  #
               use layer block3_conv3 for content representation
48
49         output_layers_S = []
50
51         for layer in style_layers:        #use layers in STYLE_LAYERS for
               style representations
52             output_layers_S.append(self.img_model.get_layer(layer).output)
53
54         new_input = self.img_model.input
```

```
        new_output = [output_layer_C] + output_layers_S
        ## let img_encoder denote the pretrained model
        self.img_encoder = tf.keras.Model(new_input, new_output)
        return


    # content cost
    def __compute_content_cost(self, a_C, a_G):


        m, n_H, n_W, n_C = a_G.get_shape().as_list()

        a_C_unrolled = tf.reshape(a_C, [m,-1,n_C])
        a_G_unrolled = tf.reshape(a_G, (m,-1,n_C))

        J_content = tf.reduce_sum(tf.square(tf.subtract(a_C_unrolled,
            a_G_unrolled)))/(4*n_H*n_W*n_C)

        return J_content


    #style cost
    def __gram_matrix(self, A):

        GA = tf.matmul(A, tf.transpose(A))
        return GA

    def __compute_layer_style_cost(self,a_S, a_G):
        m, n_H, n_W, n_C = a_G.get_shape().as_list()


        a_S = tf.reshape(tf.transpose(a_S, perm=[3,1,2,0]), [n_C, -1])
        a_G = tf.reshape(tf.transpose(a_G, perm=[3,1,2,0]), [n_C, -1])


        GS = self.__gram_matrix(a_S)
        GG = self.__gram_matrix(a_G)


        J_style_layer =  tf.reduce_sum(tf.square(tf.subtract(GS,GG)))/(4*n_C
            **2*(n_W*n_H)**2)


        return J_style_layer

    def __compute_style_cost(self,output_S,output_G, weights):

        style_cost = 0

        for i in range(1, len(output_G)):
            a_S = output_S[i]
            a_G = output_G[i]
            style_cost += weights[i-1]*self.__compute_layer_style_cost(a_S,
                a_G)

        return style_cost

    #total cost
    def __total_cost(self, J_content, J_style, alpha, beta):
        J = alpha*J_content+ beta*J_style

        return J
```

```
114
115
116     def __cost(self, C, S, style_layer_weights, alpha, beta, noise_ratio):
            # C:content image, S:style, model: pretrained model
117
118         G = self.__generate_noise_image(C, noise_ratio)   # generate graph
119
120         output_C = self.img_encoder(C)
121         output_G = self.img_encoder(G)
122         output_S = self.img_encoder(S)
123
124         # compute content cost
125         a_C, a_G = output_C[0], output_G[0]
126
127         J_content = self.__compute_content_cost(a_C, a_G)
128
129
130         # compute style cost
131         J_style = self.__compute_style_cost(output_S,output_G,
                style_layer_weights)
132
133         #total cost
134         J = self.__total_cost(J_content, J_style, alpha=alpha, beta=beta)
135
136         return J_content, J_style, J
137
138
139     def __generate_noise_image(self, content_image, noise_ratio):
140
141
142         # Generate a random noise_image
143
144         # Set the input_image to be a weighted average of the content_image
                and a noise_image
145         input_image = self.W * noise_ratio + content_image * (1 -
                noise_ratio)
146
147         return input_image
148
149
150     #initialize training weights
151     def __initialize_weights(self, seed):
152         Initializer = tf.random_uniform_initializer(-1,1,seed=seed)
153         n_w, n_h, n_c = self.input_shape
154         self.W = tf.Variable(Initializer((1, n_w, n_h, n_c)))
155         return
156
157     def set_opt(self, opt):
158         self.opt = opt
159         return
160
161     # method to reset weights
162     def reinialize_weights(self, seed=123):
163         Initializer = tf.random_uniform_initializer(-1,1, seed=seed)
164         n_w, n_h, n_c = self.input_shape
165         self.W = tf.Variable(Initializer((1, n_w, n_h, n_c)))
166         return
167
168
169     #one step training
170     def __one_step(self, C, S, style_layer_weights, alpha, beta, noise_ratio
            ):
```

```
171         with tf.GradientTape() as tape:
172             J_content, J_style, J = self.__cost(C, S, style_layer_weights=
                    style_layer_weights,
173                                                 alpha=alpha, beta=beta,
                                                        noise_ratio=noise_ratio)
174 
175         grads = tape.gradient(J, [self.W])
176         opt = self.opt
177         opt.apply_gradients(zip(grads, [self.W]))
178 
179         return J_content, J_style, J
180 
181     #main method for training
182     def model_nn(self, content_path, style_path, sav_dir ,start = 0 ,
            num_iterations = 20000, noise_ratio=0.2, alpha = 10, beta = 120,
            style_layer_weights = None, seed = 123):
183         start_time = time.time()
184         if not self.img_encoder:
185             print("Please run method get_img_encoder to get image encoder
                    first!")
186             return
187         if self.W is None:
188             self.__initialize_weights(seed=seed)
189 
190         if style_layer_weights is None:
191             style_layer_weights = self.style_layer_weights
192 
193         C = self.__get_image(content_path)
194         S = self.__get_image(style_path)
195 
196         print(style_layer_weights)
197         for i in range(num_iterations):
198 
199 
200             Jc, Js, Jt = self.__one_step(C = C, S = S, style_layer_weights =
                     style_layer_weights,
201                                          alpha = alpha, beta = beta,
                                                noise_ratio = noise_ratio)
202 
203             generated_image = self.__generate_noise_image(C,noise_ratio =
                    noise_ratio)
204 
205             if (start+i)<200:
206                 if i%10 == 0:
207                     print("Iteration " + str(start + i) + " :")
208                     print("total cost = " + str(Jt.numpy()))
209                     print("content cost = " + str(Jc.numpy()))
210                     print("style cost = " + str(Js.numpy()))
211 
212                     # save current generated image in the "/output"
                            directory
213                     self.__save_image(sav_dir+'/' +str(start + i) + ".jpg",
                            generated_image)
214 
215 
216 
217             elif i%200 == 0:
218                 print("Iteration " + str(start + i) + " :")
219                 print("total cost = " + str(Jt.numpy()))
220                 print("content cost = " + str(Jc.numpy()))
221                 print("style cost = " + str(Js.numpy()))
222 
```

```
223                 # save current generated image in the "/output" directory
224                 self.__save_image(sav_dir+'/'+str(start + i) + ".jpg",
                        generated_image)
225
226         # save last generated image
227         self.__save_image(sav_dir +'/'+str(start + i) +'.jpg',
                generated_image)
228         difference_time = time.time() - start_time
229         print('#################################')
230         print('total runtime for genrating graph:'+str(difference_time))
231         print('alpha = '+ str(alpha))
232         print('beta = '+ str(beta))
233         return
```