國立臺灣科技大學
資訊管理研究所

Multimedia Systems

Deep Painterly Harmonization

學號：M10909315　姓名：黃柏崴

學號：M10909301　姓名：蔡忠諺

Agenda

1. Introduce

1-1. Motivation

Why do we want to make this project ? Because we often see a lot of interesting meme, suddenly thought of making related themes. And recently, we have studied in deep learning related courses, so I would like to implement the Neural network through the project. We think this technology can be used as a way for corporate marketing, media and social media. Finally, we hope to compare the difference between the paper and our implementation.

1-2. Paper

Topic：Deep Painterly Harmonization
Author：Fujun Luan, Sylvain Paris, Eli Shechtman, Kavita Bala
First published：20 July 2018

The paper uses a dedicated algorithm to solve these problems, which can carefully determine the local statistical information to be transmitted. Ensure statistical consistency between space and scale, and prove that these two aspects are the key to producing high-quality results. In order to cope with the diversity of abstract levels and painting types, a technique of adjusting transfer parameters according to painting is introduced. We show that our algorithm produces results that are significantly better than photo synthesis or global stylization techniques, and it enables creative painting editing that would otherwise be difficult to achieve.

1-3. Architecture

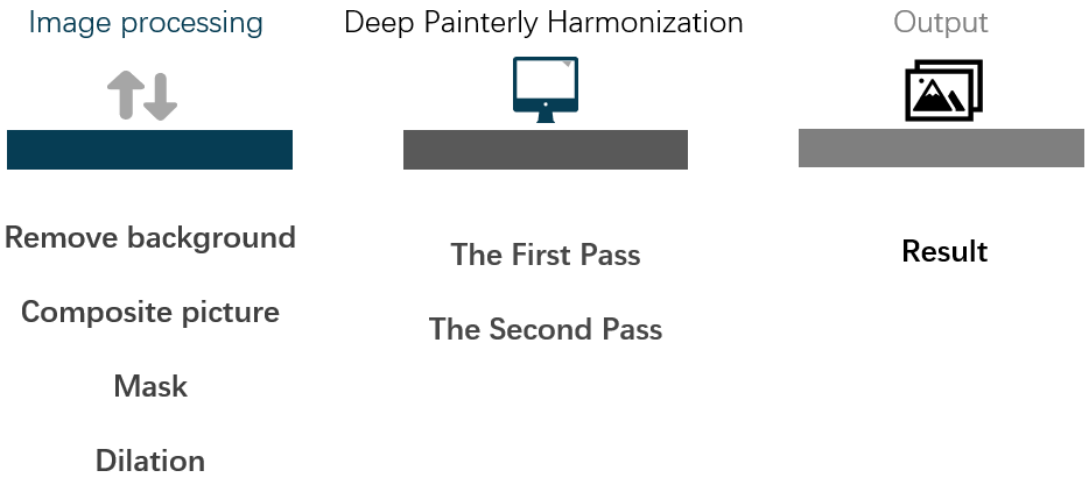

Figure.1

There are three step for us to do ,first we do the image processing, we remove the background to get item, composite the picture and we use mask and dilation. Second, we do the deep painterly harmonization, there are two pass for us to do ,after the pass, we will get the result we want.

2. Related Work

2-1.Tool

Programming language：

1. Python

Environment ：

1. Google Colab

Package：

1. Open CV
2. Numpy
3. Pytorch
4. Pillow
5. Matplotlib

API：

1. RemoveBg

2-2. Image processing

2-2-1. Remove background

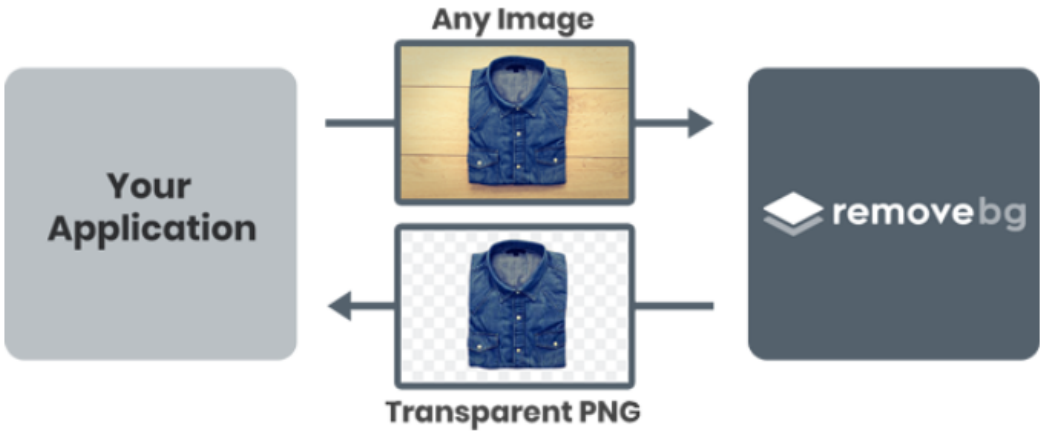

Figure.2 [3]

We once used OpenCV to complete the background removal operation, but the effect was not very satisfactory. There are still many unremoved backgrounds on the edge of the

image. So,in order to complete the image processing, we decided to use the API, just use the program to join the registered account the key of the obtained free API will be input to the picture material to be processed. The back end of remove.bg will have a smart AI that will automatically separate the main body of the photo from the background. You can even input 1000 pictures at a time to remove the background. However, since it is a free service, there will be a limit on the number of times. Only 50 API calls per month can be provided, but it is quite enough for our interim report, and the processed pictures are used for subsequent masking of the pictures. For processing and synthesis, we can get the result of the style conversion we hope through this operation.

Figure.3

2-2-2.Mask

Using mask is to use selected images, graphics or objects to occlude the processed images (total or regional) to control the image processing area or process. In digital image processing, mask is a two-dimensional matrix array, and sometimes multi-valued images are also used. The main purpose is:

1. Extract the area of interest: multiply the pre-made area of interest mask with the image to be processed to obtain the image of the area of interest. The value of the image in the area of interest remains unchanged, while the value of the image outside the area is all 0.
2. Masking function: Use a mask to mask certain areas on the image so that it does not participate in the processing or calculation of the processing parameters, or only the masked area is processed or counted.
3. Structural feature extraction: Use similarity variables or image matching methods to detect and extract structural features similar to Mask in the image.
4. The production of special-shaped images. [4]

Processing steps:

Step 1: Create a mask image of the same size as the original image, and initialize all pixels to 0, so the entire image becomes an all-black image.

Step 2: Set all the pixel values of the reserved area in the mask image to 255, that is, the entire reserved area becomes white.
Then you can get the mask image.

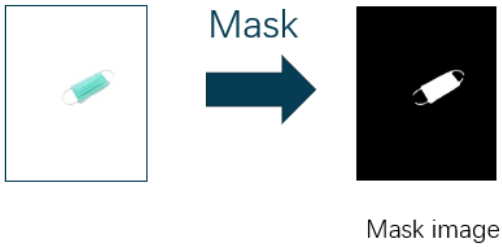

Figure.4

2-2-3.Deliation

The concept of image expansion is to expand the white area (or highlight) in the image. The calculated result image is larger than the white area of the original image. It can also be imagined to make the object fat , and the width of this circle is determined by the size of the convolution kernel. In fact, the convolution kernel slides and calculates along the shadow. If there is only one pixel value in the range of the convolution kernel mxn, then the new pixel value is 1, otherwise the new pixel value the pixel value of keeps the original pixel value, which means that all pixels scanned by the convolution kernel will be expanded or dilated (to 1), so the white area of the entire image will increase.

Uses of Dilation:
Purpose 1: Dilation image expansion is usually used in conjunction with image erosion. First, the erosion method is used to narrow the lines in the image and also remove the noise, and then the image is expanded back through Dilation.
Purpose 2: Used to connect two very close but separate objects.[6]

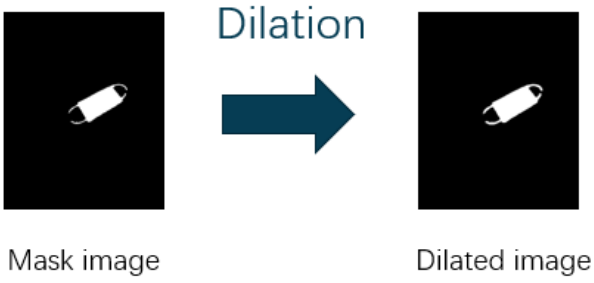

Figure.5

2-3. Algorithm

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Figure.6 [2]

VGG represents the Oxford Visual Geometry Group of Oxford University. The main work is to prove that increasing the depth of the network can affect the final performance of the network to a certain extent. VGG has two structures, VGG16 and VGG19, which are not essential. The difference is that the network depth is different.

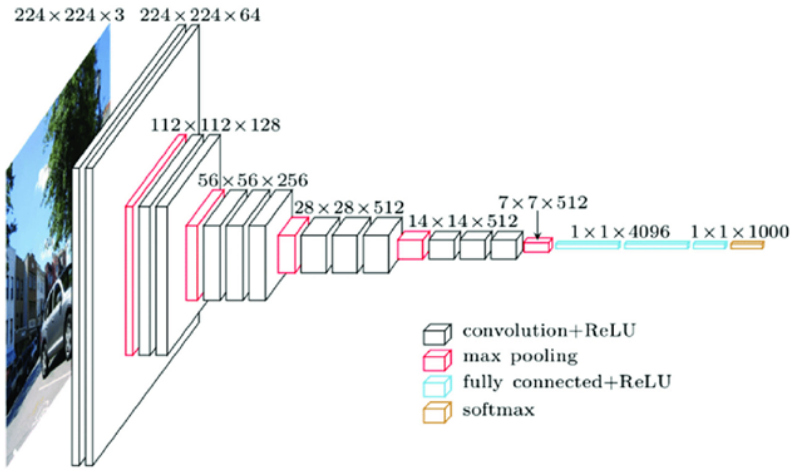

Figure.7 [1]

Features of VGG:

- Small convolution kernel ：

It's convolution kernel 3x3 (step size=1, padding=0).

- Small pooling kernel ：

And it's all pooling kernel is 2x2.

- The number of layers is deeper and the feature map is wider ：

Since the convolution kernel focuses on expanding the number of channels, and the pooling kernel focuses on reducing the width and height, the model architecture is deeper and wider, while the increase in calculation slows down, so that the network has a greater experience.

- Use Multi-Scale for data enhancement during training and prediction ：

During training, the same picture is scaled to different sizes and randomly cropped to a size of 224*224, which can increase the amount of data. When predicting, the same picture is scaled to different sizes for prediction, and finally the average is taken.

- In the network test phase, full connection is replaced with convolution：

In the network test stage, the 3 full connections in the training stage are replaced with 3 convolutions, and the training parameters are reused in the test, so that the tested full convolutional network can receive any width or height because there is no restriction on full connections input.

2-3-1.VGG16 And VGG19

The depth of the convolutional layer is 64 -> 128 -> 256 -> 512 ->512

- VGG16

-VGG16 contains 16 hidden layers (13 convolutional layers and 3 fully connected layers)

- VGG19

-VGG19 contains 19 hidden layers (16 convolutional layers and 3 fully connected layers)

2-3-2. VGG disadvantages

VGG has two big disadvantages:

1. The weight of the network architecture is quite large, which consumes disk space.
2. Training is very slow

Due to the large number of fully connected nodes and the deep network, VGG16 has 533MB+ and VGG19 has 574MB. This makes it time-consuming to deploy VGG. We still use VGG in many deep learning image classification problems.[5]

3.Deep Painterly Harmonization Theory

3-1. Style transfer

Style transfer is the technique of recomposing one image in the style of another. Two inputs, a content image and a style image are analyzed by a convolutional neural network which is then used to create an output image whose “content” mirrors the content image and whose style resembles that of the style image

3-2. Foreword

We propose to make two different phase . The first one will focus more on the general style, giving an intermediate result that where the object will still stand out a bit in the picture. The second phase will focus more on the details, and smoothening the edges that could have appeared during the first part

We'll call the content picture the painting with our object pasted on it and the style picture the original painting. The input is the content picture for phase 1, the result of this first stage for phase 2. In both cases, we'll compute the results of the convolutional layers for the content picture and the style picture at first, which will serve as our reference features. Then we compute the results of the same convolutional layers for our input, compare them and calculate a loss from that.

We'll compute the gradients of this loss and use them to get a better input, then reiterate the process

3-3. The first pass

In this pass, first step we define two Loss Function, one is a **content loss**, that measures the difference between our input and the content image, a **style loss**, that measures the difference between our input and the style image, sum them with certain weights to get our final loss, and will mask all the parts of the image that have nothing to do with it when we compute our loss.

We will use a slightly dilated mask, that encircles a bit more than just the object we're adding. It's the **mean-squared error** (**MSE**) between the masked features of our content image and the masked features of our input, use the result of the fourth convolutional layer only for this content loss. Using one of the first convolutional layers would force the final

output to match the initial object :

$$\frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2$$

Figure.8

Reference:

https://staruphackers.com/%E4%BB%80%E9%BA%BC%E6%98%AF%E5%9D%87%E6%96%B9%E8%AA%A4%E5%B7%AE-mean-square-error-mse%EF%BC%9F/

About style loss, We'll use Gram matrices like we do for regular style transfer, for each layer of results we have from our model, we'll look at each 3 by 3 part of the content features, and find the 3 by 3 patch in the style features that looks the most like it, and match them. To measure how much two patches look alike, we'll use the cosine similarity between them.

$$\text{similarity} = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2} \times \sqrt{\sum_{i=1}^{n} (B_i)^2}}$$

Figure.9

Reference: https://clay-atlas.com/blog/2020/03/26/cosine-similarity-text-count/

Once that mapping is done , we will transform the style features so that the centers of each 3 by 3 patch in the content features is aligned with its match in the style features. Then we will apply the resized mask on the input features and the style features, compute the Gram matrices of both of them then take the mean-squared error to give us the style loss.

Final loss **of this first stage is** then:

$$\mathcal{L} = 5\mathcal{L}_{content} + 100\mathcal{L}_{loss}$$

Figure.10

Reference: https://sgugger.github.io/deep-painterly-harmonization.html

Once we're done with the construction of our input, we make a mean between our output and the style picture to have our final output.

3-4 . The second pass

We propose to do a second pass to refine the first result. The first change is in the matching process. This time, the matching between the content and the style is done on a reference layer first, and the results will be transported to the others, but this mapping won't be different for each layer like in the first pass. Then, after doing the first mapping between the content and the style for this reference layer like in the first stage, they refine it by trying to insure that adjacent vectors in the style features remain adjacent through the mapping.

For each pixel p, we consider a certain set of candidates built by going in every direction on p', taking the value given by our first match, and applying to it the inverse of the translation that goes from p to p'. Then we find the candidate that minimizes the **L2 loss** between its style features and the ones of its neighbors.

The matching being done, use the convolutional layers number 1, 2, 3 and 4 for the style loss, and the fourth convolutional layer for the content loss. also consider two more losses. The first one, the Total Variation loss, just sums the difference between adjacent pixel values, which will insure the result is smoother:

$$\mathcal{L}_{tv} = \sum_{x,y} \left((\varnothing_{x,y} - \varnothing_{x-1,y})^2 + (\varnothing_{x,y} - \varnothing_{x,y-1})^2 \right)$$

Figure.11

Reference: https://sgugger.github.io/deep-painterly-harmonization.html

3-5. The histogram loss

Histogram matching is a technique that is often used to modify a certain photograph with the luminosity or shadows of another .

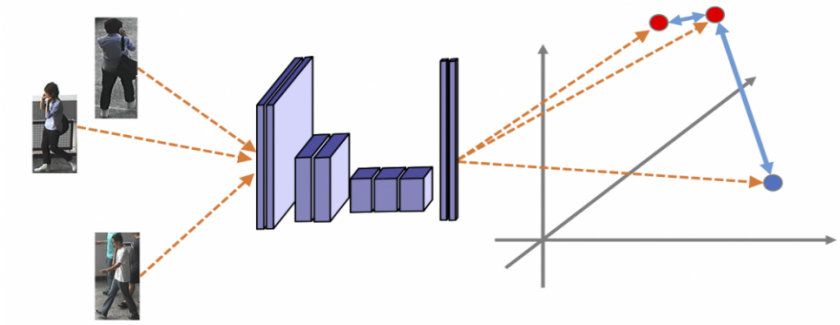

Figure.12

Reference: https://oldpan.me/archives/histogram-loss-description

We compute the histogram of each channel of the style features as a reference. Then, at each pass of our training, we calculate the remapping of our output features so that their histogram matches the style reference. We then define the histogram loss as being the the mean-squared error between the output features and their remapped version. The challenge here is to compute that remapping.

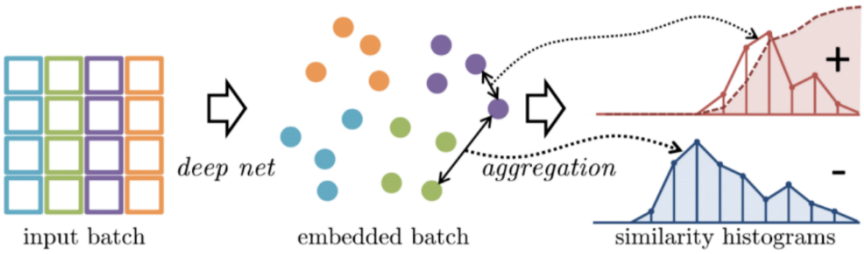

Figure.13

Reference: https://oldpan.me/archives/histogram-loss-description

We are change x so that it matches an histogram hist. We sort x first, while keeping the permutation we had to do. Then, when we treat the i-th value, we look at the first index idx such has hist.cumsum(idx) is greater than i (which means the i-th value of the data we are trying to match the histogram is in the bin with index idx). The value x[i] is basically:

$$\min + \text{idx} \times \frac{\max - \min}{n_{bins}}$$

Figure.14

Reference: https://sgugger.github.io/deep-painterly-harmonization.html

Because we have several values of x with the same index idx, we want them to be evenly distributed inside the range of the bin. So we compute the ratio :

$$\text{ratio} = \frac{i - \text{hist.cumsum}(\text{idx} - 1)}{\text{hist}[\text{idx}]}$$

Figure.15

Reference: https://sgugger.github.io/deep-painterly-harmonization.html

Finally calculate :

$$x[i] = \min + (\text{idx} + \text{ratio}) \times \frac{\max - \min}{n_{bins}}.$$

Figure.16

Reference: https://sgugger.github.io/deep-painterly-harmonization.html

3-6. Normalization

we apply a mask with n1 elements for the style features and a mask with n2 elements for the input features, I decided to multiply the style features by $\sqrt{\frac{n2}{n1}}$, to artificially *resize* them. Why? Well the gram matrix is computed by doing a sum, which will either have n1 or n2 elements, of products of two elements of our features. So inside that sum, when we compute the gram matrix of the style features, the square root will disappear and we will multiply the result by n2/n1, which is a way to *resize* this sum of n1 elements to a sum of n2 elements .

At the end, those four losses are summed with some weights to give the final loss of the second stage:

$$\mathcal{L} = \mathcal{L}_c + w_s \mathcal{L}_s + w_h \mathcal{L}_{hist} + w_{tv} \mathcal{L}_{tv}$$

Figure.17

Reference: https://sgugger.github.io/deep-painterly-harmonization.html

4. Deep Painterly Harmonization Process

4-1. Phase 1

We will focus on four pictures associated to it: the input (our objet superposed on the painting), the style image , the mask that allows us to know where we added the object and a dilated version of it.

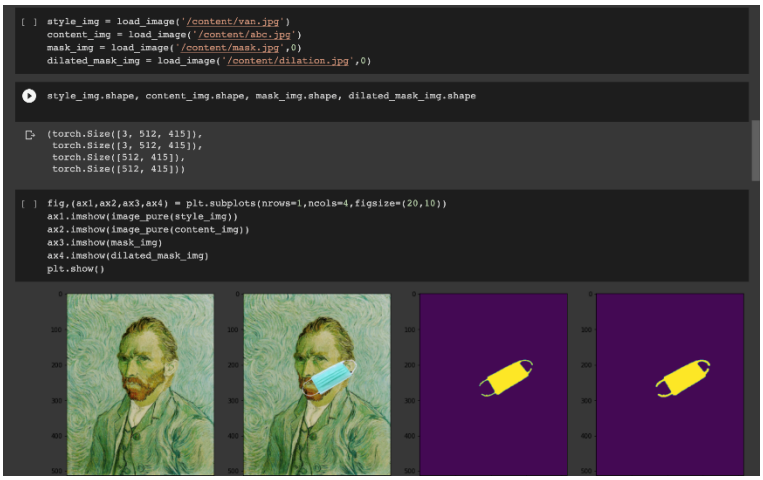

Figure.18

We using VGG16 and in the two steps of the algorithm, we will only use the results of the layer 36 max .

```
model = models.vgg16_bn(pretrained=True).features.to(device)
model.eval()
for param in model.parameters():
    param.requires_grad_(False)
```

```
def vgg_features(data,index=None):
    layers_index = {'2' : 'conv1_1', # 'layer' : 'layer_name'
                    '9' : 'conv2_1',
                    '16' : 'conv3_1',
                    '26' : 'conv4_1',
                    '29' : 'conv4_2',  ## content representation
                    '36' : 'conv5_1'}
    layers = index if index else layers_index
    features={}
    for layer_index, layer in model._modules.items():
        data = layer(data)
        if layer_index in layers:
            features[layers_index[layer_index]]=data
    return features
```

Figure.19

The first pass is based on a content loss and a style loss as is usual in style transfer, but first, we have to map every pixel of the content features to one in the style features. To determine the mapping, we look at every 3 by 3 window of the style features, that we flatten into a vector of size 9 * channels. We will then map a pixel in the content to the one in the style that is at the center of the patch nearest to the 3 by 3 patch around the pixel.

Basically we had a padding of zeros the start indexes correspond to all the top-left corners of the windows, and the grid gives the indexes of all the 3 by 3 by channel window.

```
[ ] def get_patch(img, kernel_size=3 ,stride=1 ,padding=1):
        """
        returns the array of values by applying the filter of
        kernel_size * kernel_size at each pixel of the image
        """
        patcher = nn.Unfold(kernel_size = kernel_size, padding=padding, stride=stride)
        patches = patcher(img)
        return patches.transpose(1,2).squeeze()

    def match_features(content_img, style_img):
        """
        Use the cosine similarity concept to find out how much
        two image patched look alike return the indexes
        """
        result = []
        for input_layerfeature, style_layerfeature in zip(content_img, style_img):

            input_layerpatch = get_patch(input_layerfeature)
            style_layerpatch = get_patch(style_layerfeature)

    #         input_dot_style = input_layerpatch @ style_layerpatch.T
    #         mod_input_layerpatch = torch.sqrt(torch.sum(input_layerpatch ** 2, 1)).unsqueeze(1)
    #         mod_style_layerpatch = torch.sqrt(torch.sum(style_layerpatch ** 2 ,1)).unsqueeze(0)
    #         down = (1e-15 + torch.mm(mod_input_layerpatch, mod_style_layerpatch))
    #         cosine_similarity = input_dot_style / down
    #         _, index = cosine_similarity.max(1)
            cosine = cosine_similarity(input_layerpatch.cpu(), style_layerpatch.cpu())
            index = np.argmax(cosine,1)
            result.append(torch.tensor(index, device=device))
        return result
```

Figure.20

Launch the training, And display the results, then we'll use the LBFGS optimizer.

```
%%time
optimizer = torch.optim.LBFGS([target_img], lr=1)
epoch = 1000
loop = 0
print(f'{"loop":>7}{"total_loss":>15}{"style_loss":>15}{"content_loss":>15}')
while loop<=epoch:
    def closure():
        optimizer.zero_grad()
        output = list(vgg_features(target_img, index=['16','26','36']).values())
        loss = total_loss(output)
        loss.backward()
        return loss
    optimizer.step(closure)
target_img = target_img.detach().squeeze()
```

```
   loop     total_loss     style_loss   content_loss
      1    69.58280182    69.58280182     0.00000000
    100     1.30727053     1.30095959     0.00631090
    200     0.78883028     0.78224307     0.00658723
    300     0.63925141     0.63250035     0.00675107
    400     0.56462258     0.55772656     0.00689602
    500     0.52353871     0.51656103     0.00697768
    600     0.49607390     0.48903811     0.00703578
    700     0.47650459     0.46943334     0.00707126
    800     0.46118429     0.45408255     0.00710175
    900     0.44950059     0.44236392     0.00713668
   1000     0.43983108     0.43268096     0.00715012
CPU times: user 52.5 s, sys: 24.2 s, total: 1min 16s
Wall time: 1min 16s
```

Figure.21

```
phase1_img = (target_img.cpu()*mask_img + content_img*(1-mask_img)).squeeze()
fig,(ax1,ax2) = plt.subplots(nrows=1,ncols=2,figsize=(20,20))
ax1.imshow(image_pure(target_img))
ax2.imshow(image_pure(phase1_img))
plt.show()
```

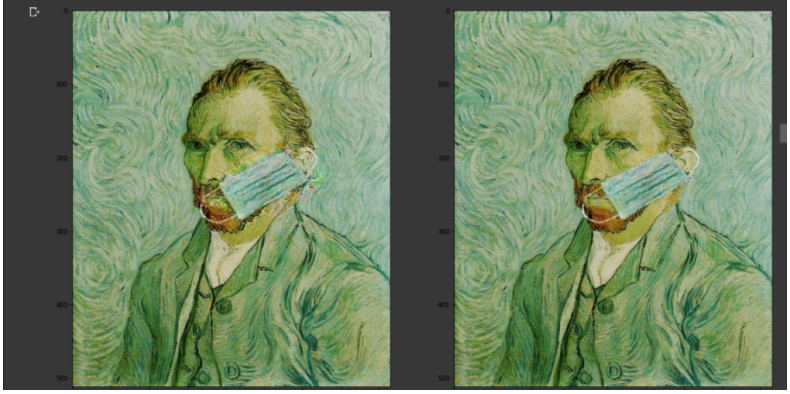

Figure.22

4-2. Phase 2

The features of the style image will be used in the style and histogram loss , as before, we let the mask *run* through the network.

```python
class MaskFeatures2(nn.Module):

    def __init__(self):
        super().__init__()
        self.convol_mask = nn.AvgPool2d(kernel_size=3, padding=1, stride=1)

    def resize2half(self,data):
        return F.interpolate(data, size=(data.shape[2]//2, data.shape[3]//2))

    def forward(self,data):
        features=[]
        data = data[None,None,:,:]
        data = self.convol_mask(data)
        features.append(data.squeeze().to(device))
        data = self.resize2half(self.convol_mask(data))
        data = self.convol_mask(data)
        features.append(data.squeeze().to(device))
        data = self.resize2half(self.convol_mask(data))
        data = self.convol_mask(data)
        features.append(data.squeeze().to(device))
        data = self.convol_mask(data)
        data = self.resize2half(self.convol_mask(data))
        data = self.convol_mask(data)
        features.append(data.squeeze().to(device))
        return features
```

Figure.23

Then we adapt the mapping to the other layers and remove the duplicates in the style features that are selected. This function still remaps the style but also returns a mask that will cover the repetitions with a 0.

```python
def upsample(mapped, orig_size, new_size):
    orig_height, orig_width = orig_size
    new_height, new_width = new_size
    TOTAL_NewMap_PIXEL = new_height * new_width
    new_map = torch.zeros(TOTAL_NewMap_PIXEL, dtype=torch.int32)#.to(device)
    scale_factor_height, scale_factor_width = new_height/orig_height, new_width/orig_width

    for map_pixel in range(TOTAL_NewMap_PIXEL):
        new_map_x_index , new_map_y_index = divmod(map_pixel, new_width)
        orig_x_index = min((0.5+new_map_x_index)//scale_factor_height, orig_height-1)
        orig_y_index = min((0.5+new_map_y_index)//scale_factor_width, orig_width-1)
        mapped_pixelIndex = mapped[int(orig_x_index*orig_width + orig_y_index)]

        mapped_x_index = int(new_map_x_index + (mapped_pixelIndex//orig_width - orig_x_index)*scale_factor_height + 0.5)
        mapped_y_index = int(new_map_y_index + (mapped_pixelIndex%orig_width - orig_y_index)*scale_factor_width + 0.5)

        mapped_x_index = min(mapped_x_index,new_height-1)
        mapped_y_index = min(mapped_y_index,new_width-1)

        new_map[map_pixel] = mapped_x_index*new_width + mapped_y_index
    return new_map
```

Figure.24

```python
def mapped_style2(style_features, map_features, mask_features):
    res=[]
    mask_=[]
    for style_ftr, map_ftr, mask_ftr in zip(style_features, map_features, mask_features):
        style_ftr = style_ftr.reshape(style_ftr.size(1),-1)
        mask_ftr = mask_ftr.reshape(-1)
        map_ftr = map_ftr.cpu().numpy()

        style_ftr = style_ftr[:,map_ftr]
        map_count = Counter(map_ftr)
        for index, pixel in enumerate(map_ftr):
            if mask_ftr[index] >=0.1 and map_count[pixel]>1:
                mask = (map_ftr==pixel)
                mask[index] = False
                mask_ftr[mask]=0
        res.append(style_ftr)
        mask_.append(mask_ftr)

    return res,mask_
```

Figure.25

We compute the histogram of the style features covered by its mask.

```
def histogram_mask(style_features, mask_features):
    masked = style_features * mask_features
    return torch.cat([torch.histc(masked[i][mask_features>=0.1], n_bins).unsqueeze(0)
    for i in range(masked.size(0))]).to(device)
```

Figure.26

We compute the histogram loss :

```
def histogram_loss(imgfeatures):
    hist_loss = 0
    mask_features = [dilated_mask_imgfeatures[0], dilated_mask_imgfeatures[3]]
    for img_layer, mask_layer, style_hist_layer in zip(imgfeatures, mask_features, style_hist_mask):
        img_layer = img_layer.view(img_layer.size(1),-1)
        mask_layer = mask_layer.view(1,-1)
        img_layer = img_layer * mask_layer
        temp_mask = mask_layer>=0.1
        temp_mask = temp_mask[0,:]
        img_layer = torch.cat([layer[temp_mask].unsqueeze(0) for layer in img_layer])
        hist_loss = F.mse_loss(img_layer, map_histogram(img_layer, style_hist_layer))
    return hist_loss/2
```

Figure.27

Finally the TV loss, and get the parameter used in computing the weight of the TV loss.

```
[ ] def total_variation_loss(imgfeatures):
        return ((imgfeatures[:,:-1,:] - imgfeatures[:,1:,:])**2).sum() + ((imgfeatures[:,:,:-1] - imgfeatures[:,:,1:])**2).s

def median_total_variation_loss(img):
    channel,height, width = img.shape
    tensor1 = torch.cat([torch.zeros((channel,width))[:,None,:], img], axis=1)
    tensor2 = torch.cat([torch.zeros((channel,height))[:,:,None], img], axis=2)

    return torch.median((tensor1[:,:-1,:] - tensor1[:,1:,:]) ** 2 + (tensor2[:,:,:-1] - tensor2[:,:,1:]) ** 2)
```

Figure.28

We can get our training to start , total run 1000 times, Every one hundred print total loss、 style Loss 、content loss、 histogram Loss and variational loss .

We'll use the LBFGS optimizer.

```
%%time
optimizer = torch.optim.LBFGS([final_target_img],lr=0.1)
epoch = 1000
loop = 1
print(f'{"loop":>7}{"total_loss":>15}{"style_loss_val":>15}{"content_loss":>15}{"histogram_loss_val":>20}{"total_variati
while loop<=epoch:
    def closure():
        optimizer.zero_grad()
        output = vgg_features(final_target_img, index=['2','9','16','26'])
        output = list(output.values())
        loss = final_loss(output)
        loss.backward()
        return loss
    optimizer.step(closure)
```

```
   loop     total_loss style_loss_val   content_loss  histogram_loss_val total_variation_loss_val
    100     0.08948172     5.52468538     0.00612011          0.02811476          2626755.50000000
    200     0.05225046     1.88618588     0.00546552          0.02792308          2629818.50000000
    300     0.04484642     1.23902655     0.00474547          0.02771068          2633572.50000000
    400     0.04196718     1.01283550     0.00427775          0.02756107          2637483.75000000
    500     0.04054983     0.90592569     0.00402232          0.02746825          2641346.50000000
    600     0.03958822     0.83860219     0.00381645          0.02738575          2645254.50000000
    700     0.03891996     0.79120725     0.00367905          0.02732884          2648927.25000000
    800     0.03846679     0.75935245     0.00358971          0.02728356          2652179.50000000
    900     0.03809878     0.73586369     0.00349348          0.02724667          2655319.00000000
   1000     0.03779149     0.71514720     0.00342698          0.02721304          2658503.00000000
CPU times: user 3min 36s, sys: 54.6 s, total: 4min 30s
Wall time: 4min 32s
```

Figure.29

5.Result

So, we will output 2 pictures in phase1, phase 2 and compare with original picture :

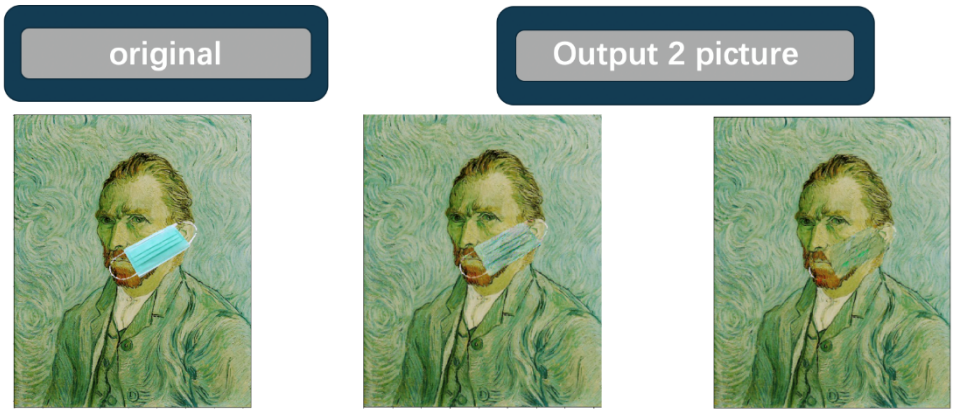

Figure.30

6. Conclusion

The difference between us and the paper is that we make the pictures we need and use the Pytorch to complete the training. And we use different algorithm to get the result. Then paper only use VGG19 to do training, and we used VGG16 to compare the differences, the result is that VGG16 results are already very good, and the use of time is a little less than VGG19, because VGG19 is much deeper, but VGG19 results have overfitting situation. So we thought it might be caused by different ways of working with pictures, but the conclusion was that it was enough to just use VGG16.

	VGG_16	VGG_19
Output		
Time	4min 21s	4min 38s

Figure.31

Our limit is that we can only manually change coordinates when we make composite pictures. And hardware is our biggest limitation, so the picture we can use size can only be below 600 pixels. If it exceeds 600 pixels, our code will not run because our graphics card Ram is not enough.

7. Reference

[1] Nash, Will & Drummond, Tom & Birbilis, Nick. (2018). A review of deep learning in the study of materials degradation. npj Materials Degradation. 2. 10.1038/s41529-018-0058-x.

[2] Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

[3] remove.bg. *Background Removal API*. https://www.remove.bg/api

[4] Madcola （2017 年 5 月 23 日）。 OpenCV 探索之路（十三）：詳解掩膜 mask 。取自網址 ： https://www.cnblogs.com/skyfsm/p/6894685.html

[5] KOUSTUBH.*ResNet, AlexNet, VGGNet, Inception: Understanding various architectures of Convolutional Networks* .
https://cv-tricks.com/cnn/understand-resnet-alexnet-vgg-inception/

[6] ShengYu （2020 年 6 月 21 日）。 [Python+OpenCV] 影像侵蝕 erode 與影像膨脹 dilate 。取自網址 ： https://shengyu7697.github.io/blog/2020/06/21/Python-OpenCV-erode-dilate/

[7] Ayush 。 Deep Painterly Harmonization 。 取自網路：
https://www.kaggle.com/codeayu/deep-painterly-harmonization

[8] Sylvain Gugger (2018 年 5 月 3 號)。 Deep Painterly Harmonization。 取自網址： https://sgugger.github.io/deep-painterly-harmonization.html

[9] Sylvain Gugger (2018 年 5 月 7 號)。 Deep Painterly Harmonization。 取自網址：
https://github.com/sgugger/DeepLearning/blob/master/DeepPainterlyHarmonization.ipynb