A Deep-Learning Based Semi-Interactive Method for Re-colorization

Tengfei Zheng PB20000296

July 12, 2023

Contents

Abstract

Aiming at the problem of colorizing grayscale images, this paper concludes commonly used algorithms about style-transferring and traditional colorization methods, indicating different views of re-colorization: by a similar image or by given colored points, both of which gives rise to an interactive method.

Ways to solve style-transferring problem include pixel-wise color transform, frequency methods and edge-detecting methods. Given a grayscale image (target image) and an RGB image (source image), we can use these methods to transfer the color onto target image.

Otherwise, if the color of the target image in some parts is already known, we can introduce optimization to fill other parts of the image by the simple principle that points having short distance in space and brightness should also be close in color.

However, both views need prior information about the image, especially about the semantic information of the image. Convolution Neural Network (CNN) is expertise in identify such information, so using neural networks becomes a natural choice.

By introducing pretrained deep learning models, this paper discusses the limitation of conventional methods, while synthesizing both views to a full semi-interactive approach. To extend the method to automatic situations, this paper studies the structure of different colorization nets, and applies some integration and improvements on them, leading to more appealing results.

As a conclusion, this paper compares the features of different views and methods, analyzing how to apply them automatically by modifying network construction.

1 Introduction

1.1 Colorizing Grayscale Images

1.1.1 Background

Before color photographic technology became widespread, all photographs were in black and white. Coming to the problem of restoring old photos, how to fill color on them forms an important part. What is more, re-colorization technology can deal with distortion caused by different light conditions, restoring the real color of the image. As **figure 1** shows, the three columns are original images, grayscale images and re-colorized images by a certain algorithm.

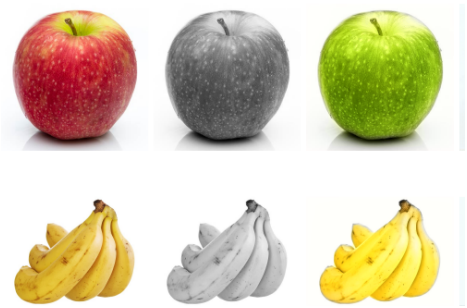

Figure 1: Sample re-colorize results [1]

Nevertheless, colorization is a challenging problem, for we have to find a fixed work procedure that can cover varying image conditions. Although scene semantics are usually helpful, for instance, grass is green and clouds are white, to recognize these high level semantics is always a difficult task.

Before taking a look at traditional or deep learning methods, we must first describe the problem in language of math and programming:

1.1.2 Image Representation

The grayscale image is often represented as a matrix, each pixel of which transformed into an integer value (called intensity) between 0 and 255, meaning the brightness at that point, see **figure 2**.

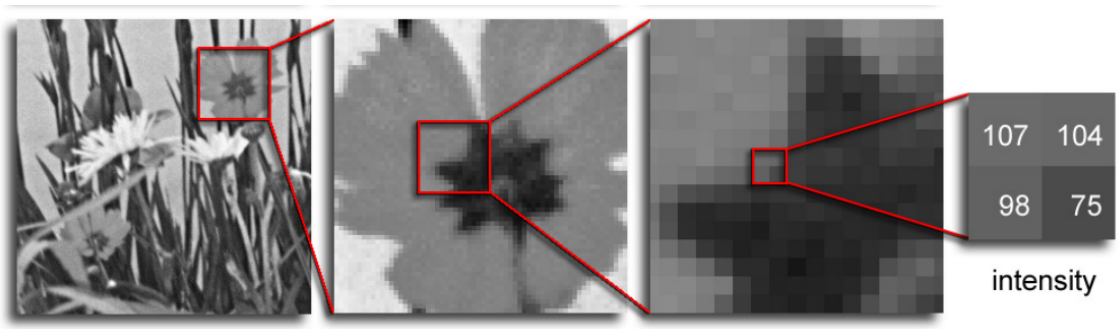

Figure 2: Grayscale representation [2]

Higher intensity means brighter, so black pixel has 0 intensity, while white pixel has 255. Pixel-wise operation usually means to do transform on the matrix.

Representing color image is a much complicated question. The most simple way to do it is to separate the image into three channels: red(R), green(G), and blue(B). By physics, RGB can form all light colors, so a color image can be represented as the combination of three grayscale images, see **figure 3**.

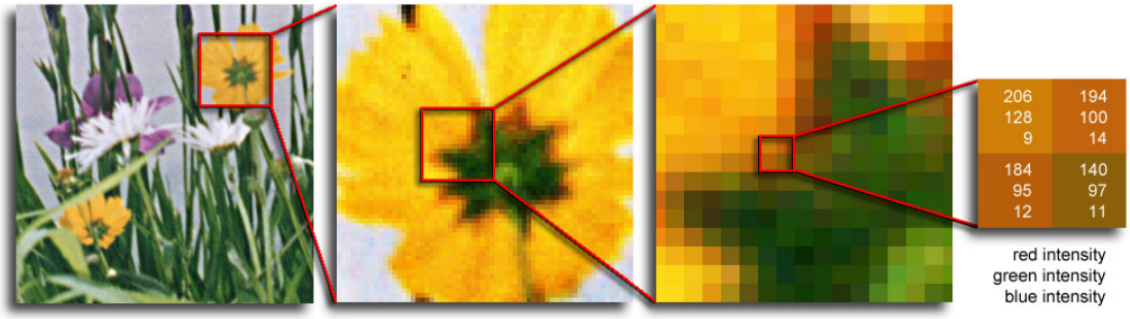

Figure 3: RGB representation [2]

Though RGB representation is intuitive and easy for the computer to show, it doesn't fit our visual perception of colors. Our sensitivity to luminance is much stronger than sensitivity to color intensity, thus we can do linear transform to RGB space to get luminance:

$$\begin{pmatrix} Y \\ U \\ V \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.1687 & -0.3313 & 0.5 \\ 0.5 & -0.4187 & -0.0813 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} + \begin{pmatrix} 0 \\ 128 \\ 128 \end{pmatrix} \tag{1}$$

Y just means the luminance of the image, or using a simpler word, "lightness". There are different weights to average RGB, resulting in different representation of luminance. U and V are called "chrominance", the color of that pixel, still having the range of $[0, 255]$. This representation form is called YUV.

By cutting luminance from chrominance, it is easier to dealing with problems relating to grayscale. Linear transforming also makes it faster to restore RGB image. Still, U and V don't conform to out perception, a better but much more complex representation is HSI: Hue, Saturation and Intensity.

$$\begin{aligned} I =& \frac{1}{3}(R+G+B) \\ S =& 1 - \frac{3\min\{R,G,B\}}{R+G+B} \\ H =& \begin{cases} \theta & B \le G \\ 2\pi - \theta & B > G \end{cases} \\ \theta =& \arccos \frac{2R-B-G}{2(R-G)^{1/2}(R-B)^{1/2}} \end{aligned} \tag{2}$$

Here the luminance is represented as the arithmetic average of RGB. Unlike RGB or YUV form, HSI representation can't be easily discrete into uint8 data type (integer between 0 and 255).

With YUV or HSI representation, the re-colorize problem can be summarized as following: given matrix Y (I), find the most likely matrix of U and V (S and H). It is obvious that such things can't be done with no prior information, for we are trying to construct three dimension from one. From different information, we can get different views of the problem.

1.2 Two Views of Colorization

1.2.1 From Certain Color Style

A common situation is that we have already known the color style of a grayscale image. For example, an image of a flower might be re-colorized to any color, but if we have a color image of the same flower, we can fill the grayscale with similar color, which is called to *transfer* the color *style*. Just as **figure 4** shows, if we know the light condition in the left image is same as the middle, we can transfer it to the right.

Figure 4: Style-transferring

Different ways to transfer leads to different effects, but the principle is clear: let S be the source image that contains color style, T be the target grayscale image, style-transferring is a problem like

$$I = \text{argmin}_X\{\alpha d_c(X,T) + \beta d_s(X,S)\}$$

Here d_c, d_s means the distance on content and style, while α, β are weight parameters. A simple thought is to let $d_c(X,T) = \|Y_X - Y_T\|_F^2$, use Frobenius norm to judge the distance, but for d_s, to find a proper form is quite uneasy: color style should be independent of location, and still show overall message.

In the section that follows, we will talk about some empirical representation of metric d_c and d_s, with the methods elicited by it. It is worth noting that style-transferring is not necessarily an explicit optimization problem. In many cases, d_c and d_s are independent, thus can be minimized respectively.

With style-transferring and an image gallery large enough, we can form a straightforward semi-interactive method: compare the target image with every image in the gallery and find one with the nearest content, then transfer the style to the grayscale. Write in formula, that is to choose S by ($\mathcal{G}$ means the gallery):

$$S = \text{argmin}_{X\in\mathcal{G}} d_c(T,X)$$

Semi-interactive implies that the method doesn't need to designate a certain image, but still the range of the gallery is important.

1.2.2 From Given Points

Another view comes from the cases when part of the image has already been colored. Use the same image as an example, **figure 5** shows a partially colored image with a mask informing where has been colored (the white pixels).

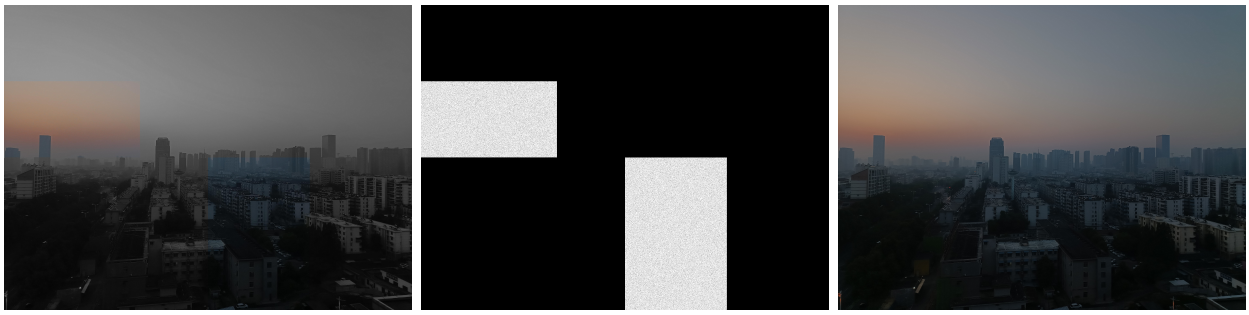

Figure 5: Given mask and color

Re-colorization is therefore converted to the problem of coloring the black part in the mask to minimize a function that indicates the conformity of the result. Because the white pixel in the mask is randomly distributed, there can't be an algorithm to get results directly, making the use of optimization essential.

One attribution is significantly important in this kind of optimization, the *gradient* of the image. For space continuity, the matrix can be regarded as uniform sampling of some smooth function, so it is possible to estimate its gradient. Since the step size is 1, define difference operators (here i and j cannot be on the edge):

$$\begin{aligned}
\partial_x a_{ij} =& \frac{a_{i+1,j} - a_{i-1,j}}{2} \\
\partial_y a_{ij} =& \frac{a_{i,j+1} - a_{i,j-1}}{2} \\
\partial_x^2 a_{ij} =& a_{i+1,j} + a_{i-1,j} - 2a_{ij} \\
\partial_y^2 a_{ij} =& a_{i,j+1} + a_{i,j-1} - 2a_{ij} \\
\Delta a_{ij} =& a_{i-1,j} + a_{i+1,j} + a_{i,j-1} + a_{i,j+1} - 4a_{ij}
\end{aligned} \tag{3}$$

We can portray local intensity difference by these operators. Because they're all linear, after combining with 2-norm, the optimization problem becomes the least squares problem, which can be precisely solved.

2 Style Transferring Methods

2.1 Pixel-wise LUT

2.1.1 Description of Content and Style

Both RGB and YUV have discrete values, so the problem can be regarded as a combinatorial optimization problem. For two pixels $(g, h), (i, j)$, we can define distance content between two

matrices of the same dimension:

$$d_{gh,ij}(X,Y) = \begin{cases} 1 & (x_{gh} - x_{ij})(y_{gh} - y_{ij}) \leq 0 \text{ and } y_{gh} \neq y_{ij} \\ 0 & \text{Otherwise.} \end{cases}$$

$$d_c(X,Y) = \sum_{(g,h)(i,j)} d_{gh,ij}(X,Y)$$

This "distance" is not symmetric, for its null point only requires two pixels that are the same in X to be the same in Y, but not the reverse. In addition to that, it requires X and Y to have the same relationship of partial order.

Visually speaking, this distance means that we recognize the content by the order of intensity in pixels. So long as we preserve the order, we can see the same contents.

For style, the metric is described as (I_l means characteristic function, which is 1 when l is true, otherwise 0):

$$h(X) = (h_k(X)), h_k(X) = \frac{1}{\text{size}(X)} \sum_{ij} I_{x_{ij} \leq k}$$

$$d_s(X,Y) = \|h(X) - h(Y)\|$$

Instead of comparing every point pairs like content, style metric just counts how many pixels of a certain intensity are in the matrix, and compare the intensity distribution. From probability view, here the h is just the distribution function of random sampling in X, called the *histogram* of a matrix.

2.1.2 Look-Up Table

Letting $\alpha = \infty, \beta = 1$, the optimization problem turns to the case which the content must be preserved. For equal pixels in X determines equal pixels in Y, we need to find a monotonic increase function f, then $y_{ij} = f(x_{ij})$. Because x and y have discrete values, this f can also be recorded as an array, which is called a look-up table.

It is not hard to construct the table by the following code:

```
j = 0
for i = 0:255
    while (g(j+1) < h(i+1)) && (j < 255)
        j = j + 1;
    end
    LUT(i+1) = j;
end
```

Here g means the histogram of the target image, while h means the histogram of the source image. By such matching, the color style of the source image can be transformed to the target image.

By independently transforming on R, G, B or Y, U, V, we can get three look-up tables. From equation(1), the inverter equation is also clear:

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1.402 \\ 1 & -0.34414 & -0.71414 \\ 1 & 1.772 & 0 \end{pmatrix} \begin{pmatrix} Y \\ U - 128 \\ V - 128 \end{pmatrix} \tag{4}$$

2.1.3 RGB Matching Results

Matching result by the algorithm upward is like **figure 6**.

Figure 6: Pixel-wise LUT result

By checking the histograms on **figure 7**, we can see how the histogram is transformed from the source image.

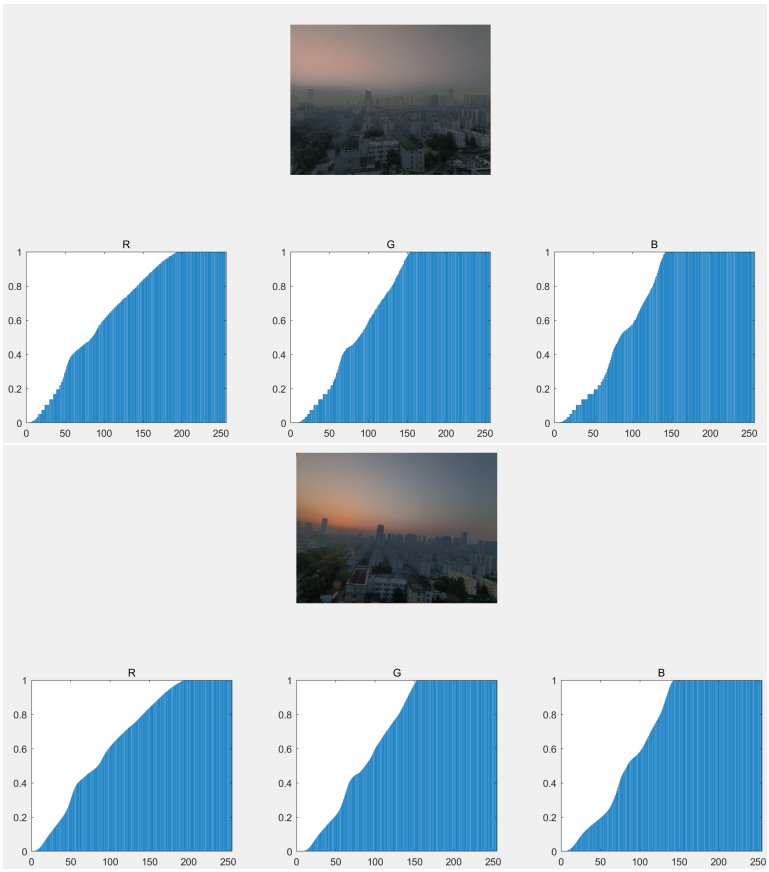

Figure 7: Histogram of result and source image

Though the result seems to color the rising sun right, we can still see two main problems. First, transforming on RGB might cause the sharpness of the color to decrease, and makes the image seem like "dirty"; second, it totally ignored information about places, which is sometimes useful to color similar images. To solve the first problem, we tried to transform on YUV space.

2.1.4 YUV Matching

However, using the transform and inverse transform formulas directly does not work well, and gets result like **figure 8**.

Figure 8: Matching by YUV

The problem is, by transforming formula, if R, G and B are all in $[0, 255]$, then Y, U and V are all in $[0, 255]$, but controlling the range of YUV can't control the range of RGB into uint8. So we need an additional normalization:

```
if min(min(R)) < 0
    R = R - min(min(R));
end
if max(max(R)) > 256
    R = R / max(max(R)) * 256;
end
if min(min(G)) < 0
    G = G - min(min(G));
end
if max(max(G)) > 256
    G = G / max(max(G)) * 256;
end
if min(min(B)) < 0
    B = B - min(min(B));
end
if max(max(G)) > 256
    B = B / max(max(B)) * 256;
end
```

What is more, if the original chrominances are all the same (for a grayscale image, they are always 128), this matching method can't work on U or V. So, a right method on YUV must have the ability of distinguishing different colors from the same grayscale.

One thought is to use the result of RGB matching as the input of YUV matching like this:

```
function J = match_YUV(T, S)
    T = match_RGB(T, S);
    T = to_YUV(T);
```

```
    S = to_YUV(S);
    J = to_RGB(match_RGB(T, S));
end
```

The matching result is like **figure 9**.

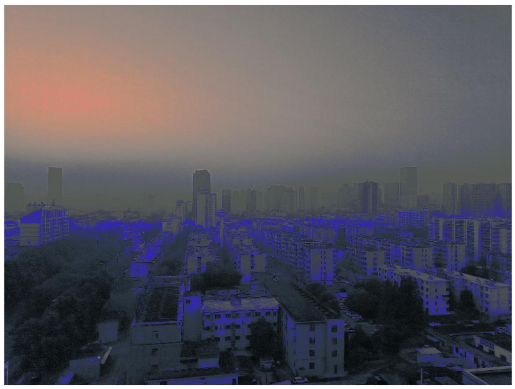

Figure 9: Matching by YUV with RGB first

While the color does get brighter, the blue region caused by the sky in source image shows again the importance of taking place into account.

Figure 10: Grayscale, color style, matching by RGB, and matching by YUV

In another example shown by **figure 10**, because the chromatic imbalance, matching by RGB has a better quality than modified by YUV. In all, although YUV matching may let the color brighter, because the serial is not as clear as RGB intensity, use YUV to build a look-up table is probably not a good choice.

2.2 Wavelet Methods

2.2.1 Frequency

Fourier transform is a well-known method to extract frequency message from a timing signal. If the frequency message of an image can be written as $f_k(X)$ ($f_k(X)$ is a vector for each natural number k), then we can write the distance function as:

$$d_c(X,T) = \sum_k c_k \|f_k(X) - f_k(T)\|^2, d_s(X,S) = \sum_k s_k \|f_k(X) - f_k(S)\|^2$$

By experiences, c_k should be higher in smaller k, while s_k should be higher in larger k, meaning that low-frequency terms are more likely to become the distance of color styles. Assuming f_k can variate independently, best $f_k(X)$ in every level k can be written as a linear combination of $f_k(T)$ and $f_k(S)$.

So, under this kind of assumption, the result can be written as

$$\mathcal{F}^{-1}(a_1 f_1(S) + b_1 f_1(T), \ldots, a_N f_N(S) + b_N f_N(T))$$

Here $\mathcal{F}^{-1}$ means some kind of Fourier inverse transform.

In spite of modifying the parameters, the key question is, how to construct different levels of "Fourier coefficients" that describe the message of an image, and also, how to add information about position to this model.

2.2.2 Wavelet Transform

Instead of using simple Fourier transform that loses all place information, wavelet transform is usually a better way to do image-processing [3].

2-dimensional discrete wavelet transform can get the low-frequency part, vertically high-frequency part, horizontally high-frequency part and diagonally high-frequency part of an image as **figure 11** (the image of high frequency is not clear because to scale it in $[0, 255]$ is unreasonable), called *wavelet coefficients*.

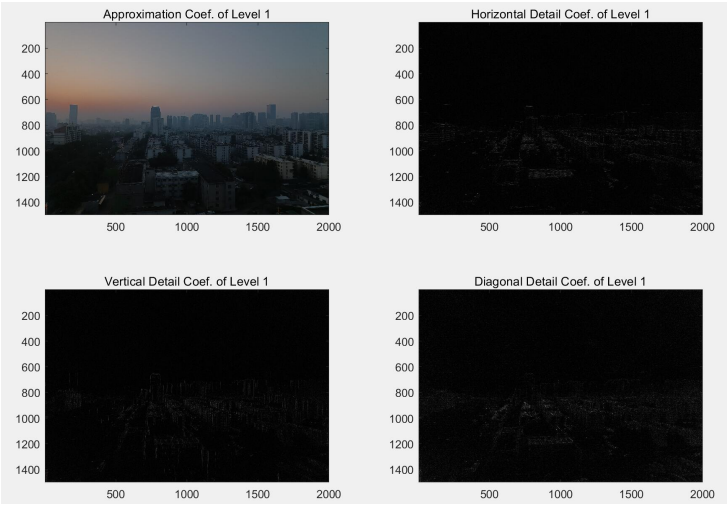

Figure 11: The low frequency and high frequency part

Continuing with the wavelet transformation on the low-frequency term, we can get different layers of wavelet coefficients. As has been required, different layers of coefficients can change independently, and the image can be reconstructed by a sequence of coefficients, so merging different layers of coefficients becomes a feasible option.

Denote the wavelet coefficients as $h_1(X), \ldots, h_k(X), l_k(X)$. Here $l_k(X)$ means the low-frequency term in the last transformation, while $h_1(X)$ meaning the three high-frequency terms in the first transformation, $h_2(X)$ meaning the three high-frequency terms in the second transformation, and so on. Given a $k+1$ dimensional vector λ with each component between 0 and 1, the result can be written as:

$$I = \mathcal{W}^{-1}(\lambda_k h_1(T) + (1-\lambda_k)h_1(S), \ldots, \lambda_1 h_k(T) + (1-\lambda_1)h_k(S), \lambda_0 l_k(T) + (1-\lambda_0)l_k(S))$$

The following **figure 12** shows result when $\lambda_0 = 1$ and $\lambda_i = 1$ for $i > 0$.

Figure 12: The influence of λ for 6, 7, 8, 9 dimensions

From these hard coefficients ($\lambda_i \in \{0, 1\}$), we can see the nature of merging wavelet coefficients, that is, to split the image into squares, and change the color of each square to the style of source image. The higher layers of coefficients are merged, the larger the squares are.

2.2.3 Soften the Result

The result above has clear edges where the square changes. To soften these pseudo-edges, λ need to be smoother, which means its difference need to be lower.

The cases in **figure 13** has λ of:

```
[0, 0.1, 0.3, 0.5, 0.7, 0.9, 1, 1, 1, 1, 1]
```

```
[0, 0.2, 0.4, 0.6, 0.8, 1, 1, 1, 1, 1, 1]
[0, 0.3, 0.6, 0.9, 1, 1, 1, 1, 1, 1, 1]
[0, 0.4, 0.8, 1, 1, 1, 1, 1, 1, 1, 1]
```

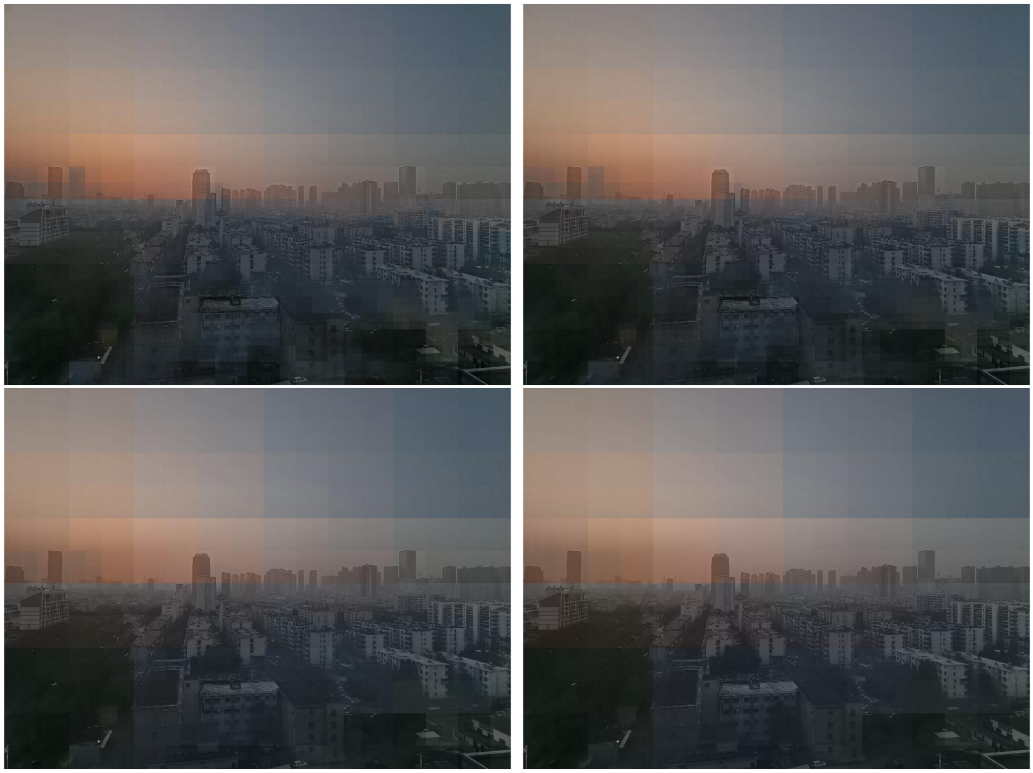

Figure 13: The influence of smoother λ

Clearly, when λ becomes smoother, the edges are weakened, but more regional information of source image is displayed in the result, going against the requirement of "color style". Another attempt is to combine merging with matching methods, using merging as a fine-tuning after matching.

For instance, the result of matching RGB and then let $\lambda = (1, 0.95, 0.8)$ is **figure 14**.

Figure 14: Combine with matching

Another way to smooth the merging result is localized contrast enhancement method, as **figure 15**.

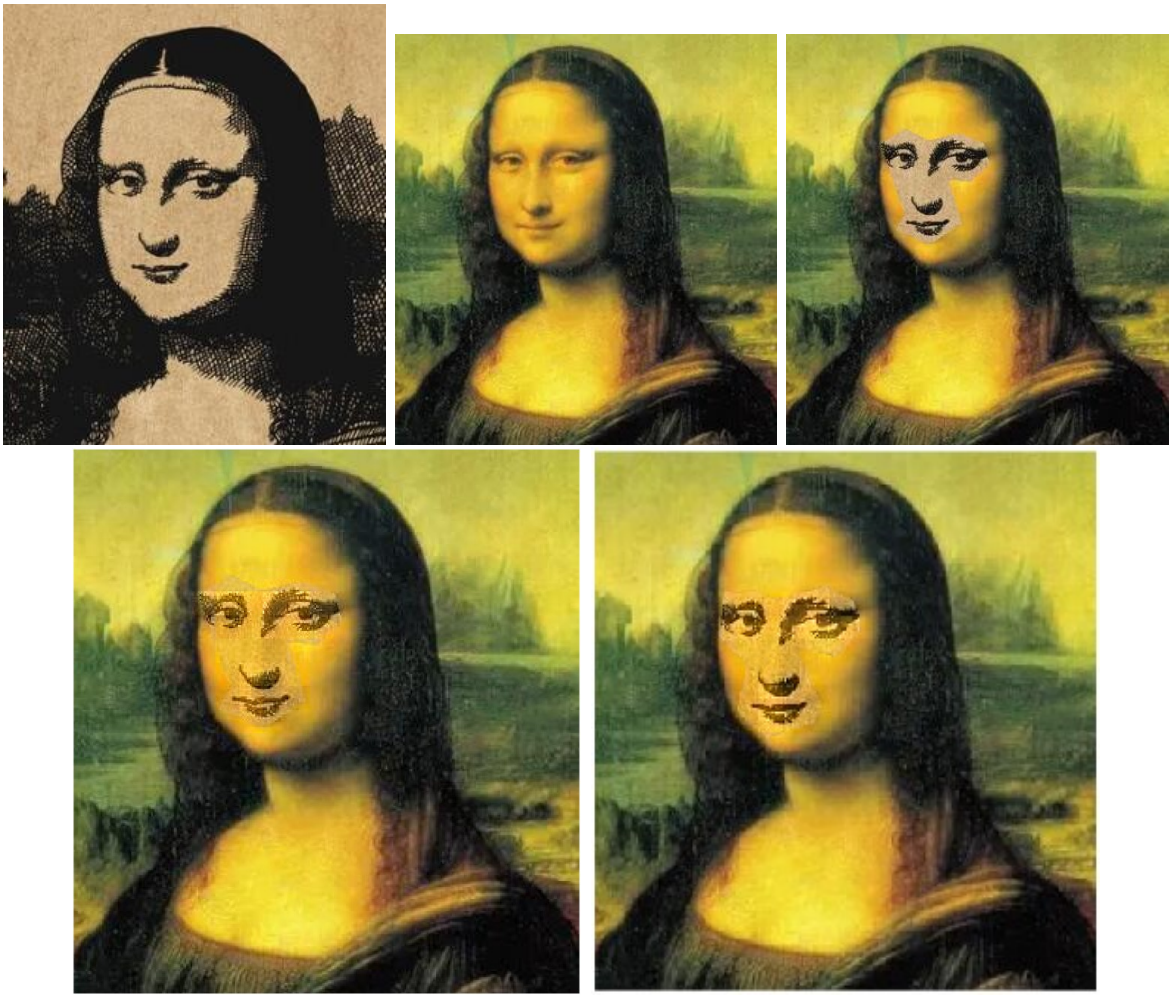

Figure 15: Localized contrast enhancement

The three images upward are the target image, the source image and the result of splicing, while the image at bottom-left shows the result of merging by wavelet coefficients. We can first enhance the contrast of target image by:

```
function J = contract(I, region, d)
    J = I;
    h = histogram(I, region);
    t = 1;
    while h(t) < 0.5 && t < 256
        t = t + 1;
    end
    [R, C, ~] = size(I);
    for i = 1:R
        for j =1:C
            if region(i, j) > 0
                J(i, j, :) = (double(I(i, j, :)) - t) * d + t;
            end
        end
```

```
        end
    end
```

Before merging was done, the amplified target image gives stronger boundaries, weakening the visual effect of pseudo-edges. However, this can only work well with locally coloring like **figure 15**, coming to global re-colorization, the range of intensity is too large to amplify.

Needless to say, the merging can also be done in YUV space. Actually, we only need to merge coefficients on U and V. With $\lambda = (0.2, 0.4, 0.6, 0.8, 0.9, 1, 1, 1, 1, 1, 1)$, the result is **figure 16**.

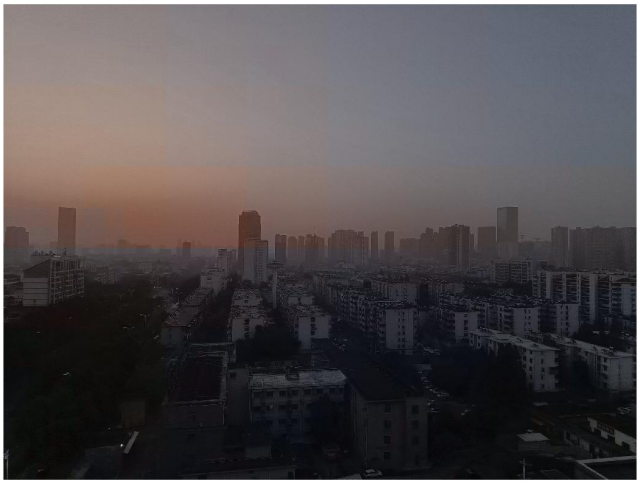

Figure 16: Merging in chrominances

We can see that settling Y can restrain the luminance, which also weakens the pseudo-edges.

2.3 Edge Detection

2.3.1 Convolution Methods

One of the most important way for us to recognize the content is by its shape, and the shape is determined by the edges of regions. This subsection, we will talk about ways to detect edges and use them to optimize transferring results.

Unlike frequency information in wavelet coefficients, the result of edge detection doesn't need to be able to reconstruct the image. It can be displayed as logical data, or a possibility of a pixel being edge.

Convolution methods use a matrix M with $m_{st}, -N \leq s, t \leq N$ to do transform:

$$C_M(X)_{ij} = \sum_{s,t} x_{i-s,j-t} m_{st} \tag{5}$$

By using different M, $C_M(X)$ can have different effects. If M is non-negative and has a sum of 1, it is called a filter. Some common filters include mean filter $m_{st} = \frac{1}{(2N+1)^2}$ and Gauss filter $m_{st} = \alpha \exp(-\beta(s^2 + t^2))$.

If M has negative components, it can distinguish the difference between pixels. In fact, every linear difference operator can be regarded as a convolution.

Probability that a pixel is an edge can be determined by the changes in the neighborhood, so the matrices below can be applied to detect edges:

$$\begin{pmatrix} 0 & 0 & 0 \\ -1 & 2 & -1 \\ 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & -1 & 0 \\ 0 & 2 & 0 \\ 0 & -1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix}, \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}$$

The effect of first three matrices is **figure 17**.

Figure 17: Merging in chrominances

After processing with more precise filters and dropping out noises, the edges can be more accurate.

2.3.2 Transforming with Edge Orientation Information

There are many ready-made ways to localize edges, so we don't have to build from scratch. The more important thing is to use edges to build a better distance function.

Assume $P_e(X)$ is the possibility of edge at each pixel, then the higher $P_e(X)_{ij}$ is, the clearer the pixel can be distinguished from its neighbors. From this, we can construct a content metric:

$$d_c(X,T) = \|P_e(Y(X)) - P_e(T)\|_F^2$$

Here $Y(X)$ means the Y component of X, the luminance. Also, the style metric can be written as:

$$d_s(X,S) = \|h(R(X)^-) - h(R(S)^-)\|^2 + \|h(G(X)^-) - h(G(S)^-)\|^2 + \|h(B(X)^-) - h(B(S)^-)\|^2$$

$$A^- = \{a_{ij} \in A \mid P_e(a_{ij}) < t\}$$

By adding a threshold t, the singular points can be ignored when counting histograms, which further enhances the characterization of the target image.

Despite the simple expression of d_c and d_s, to find the best X is particularly difficult. The comparison of histogram is a discontinuous problem, so methods like gradient descent cannot be applied. Neither histograms nor wavelet coefficients are easy to derive, so when comes to the situation where empirical methods cannot work, style-transferring seems useless. To solve this, we must first take a look at the other view, optimizing from partially colored image.

3 Colorization by Optimizing

3.1 Continuity Preserving Methods

3.1.1 Mask

This section, we try to solve the problem about how to colorize the whole image if part of it has been colored. Having a loss function $l(X)$ to determine the degree of realism of the picture, the problem can be written as:

$$I = \text{argmin}_{X\in\mathcal{M}}\{l(X)\}, \mathcal{M} = \{X \mid x_{ij,k} = c_{ij,k}, (i,j,k) \in M \text{ and } Y(X) = T\} \tag{6}$$

For a color image, $k \in \{1,2,3\}$, and $c_{ij,k}$ means the colored part. The pixels in the mask needn't be optimized, so if $X \in \mathbb{R}^{m\times n\times 3}$, the problem is an optimization of $2(mn - \#M)$ variables.

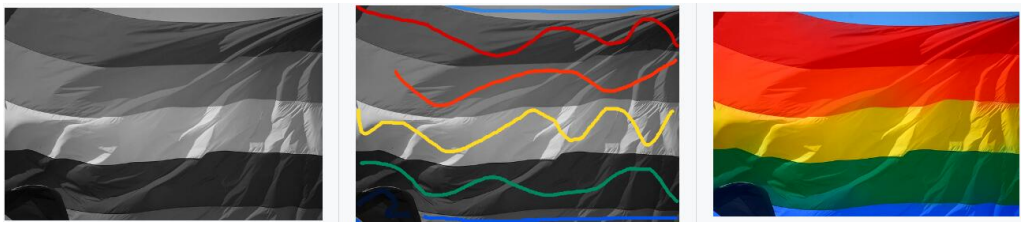

Figure 18: Optimizing results [4]

Like **figure 18** has shown, to get an ideal result, the mask need to cover pixels where chrominance change.

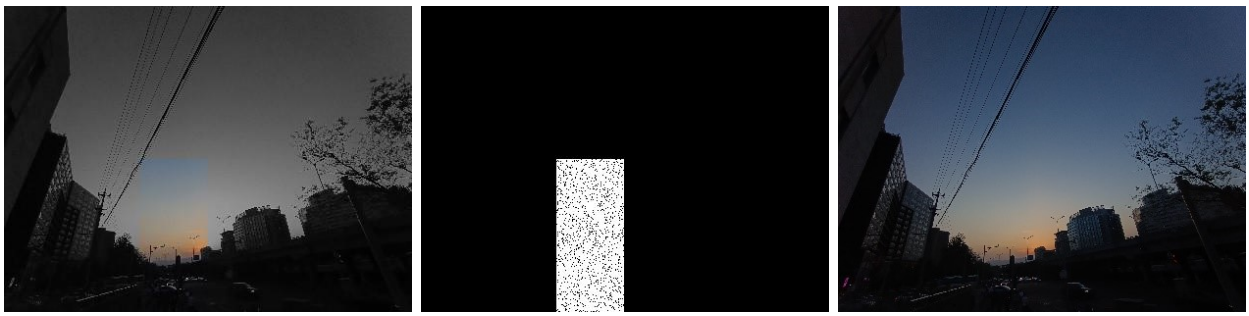

Figure 19: Another mask

Another example of mask is **figure 19**, though it only covers a small range in the image, it has described the main color of it. Noting that complexity of optimization problems grow fast with the increase of parameters, we have to first compress the image.

3.1.2 RGB Optimizing

In RGB space, we can describe the problem as:

$$\text{argmin}_{X\in\mathcal{M}} \sum_{I\in\{R,G,B\}} (d_c(I(X),T) + \lambda l_c(I(X),T))$$

Here l_c means the loss provided from continuity.

A natural form of l_c is

$$l_c(f,g) = \iint (\Delta f - \Delta g)^2 \mathrm{d}\Omega$$

Written in the form of matrix, it is

$$l_c(X,T) = \sum_{ij} (\Delta x_{ij} - \Delta t_{ij})^2$$

or

$$l_c(X,T) = \sum_{|i-t|+|j-s|=1} (x_{ij} - x_{ts} - t_{ij} + t_{ts})^2$$

In fact, as has been said, this form of l_c protects the edges in the target image. Let d_c here be

$$d_c(X,T) = \sum_{ij} (x_{ij} - t_{ij})^2$$

The optimization is just the least squares problem. For three channels are independent, just take R as an example.

Given every x_{ij} a serial number x_s, and use $\mathcal{N}(s)$ to mark the neighborhood of s, derive by x_s, we can find that

$$x_s - t_s + \lambda \sum_{i\in\mathcal{N}(s)} (x_s - x_i - t_s + t_i) = 0$$

$$(1 + \lambda \#\mathcal{N}(s)) x_s - \lambda \sum_{i\in\mathcal{N}(s)} x_i = (1 + \lambda \#\mathcal{N}(s)) t_s - \lambda \sum_{i\in\mathcal{N}(s)} t_i \tag{7}$$

In equation(7), the x_i might not be a variable to solve, for if it has been colored, it is actually a constant. Building such matrix is difficult in coding, because we need to maintain two bijection from pixel (i,j) to index s:

```
% 2 to 1 and 1 to 2
t_to_o = zeros(R, C, "int32");
o_to_t = zeros(d, 2, "int32");
count = 0;
for i = 1:R
    for j = 1:C
        if ~M(i, j)
            count = count + 1;
            t_to_o(i, j) = count;
            o_to_t(count, 1) = i;
            o_to_t(count, 2) = j;
```

```
        end
    end
end
```

Then, we need to construct A and b and solve $r = A^{-1}b$, use o_to_t to fill in back result:

```
A = sparse(1:d, 1:d, ones(1, d), d, d);
b = zeros(d, 1);
for i = 1:d
    x = o_to_t(i, 1);
    y = o_to_t(i, 2);
    b(count) = T(x, y);
    if x ~= 1
        A(i, i) = A(i, i) + lambda;
        b(i) = b(i) + lambda * (T(x, y) - T(x-1, y));
        if t_to_o(x-1, y) ~= 0
            A(i, t_to_o(x-1, y)) = -lambda;
        else
            b(i) = b(i) + Color(x-1, y) * lambda;
        end
    end
    if y ~= 1
        A(i, i) = A(i, i) + lambda;
        b(i) = b(i) + lambda * (T(x, y) - T(x, y-1));
        if t_to_o(x, y-1) ~= 0
            A(i, t_to_o(x, y-1)) = -lambda;
        else
            b(i) = b(i) + Color(x, y-1) * lambda;
        end
    end
    if x ~= R
        A(i, i) = A(i, i) + lambda;
        b(i) = b(i) + lambda * (T(x, y) - T(x+1, y));
        if t_to_o(x+1, y) ~= 0
            A(i, t_to_o(x+1, y)) = -lambda;
        else
            b(i) = b(i) + Color(x+1, y) * lambda;
        end
    end
    if y ~= C
        A(i, i) = A(i, i) + lambda;
        b(i) = b(i) + lambda * (T(x, y) - T(x, y+1));
        if t_to_o(x, y+1) ~= 0
            A(i, t_to_o(x, y+1)) = -lambda;
        else
            b(i) = b(i) + Color(x, y+1) * lambda;
        end
```

```
    end
end
```

This method to optimize gradient is also known as *Poisson merging*.

3.1.3 Poisson Results

For different λ, the result is clearly not the same, and **figure 20** shows the influence.

Figure 20: Influence of lambda

From left to right, λ goes through $0, 1, 10^3, 10^{10}$. With larger lambda, more color information proliferates into the image. Even so, the same gradient of R, G, B in target grayscale image prevents the color from getting more real. The result of uncompressed target image with $\lambda = 10^{10}$ is **figure 21**.

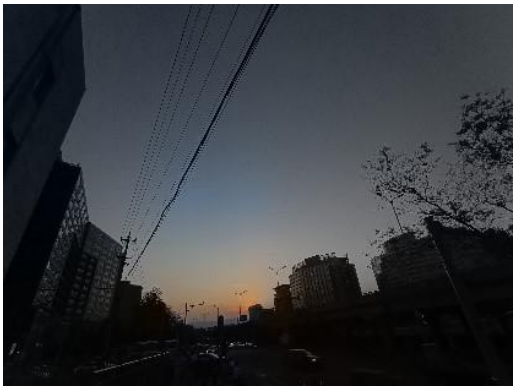

Figure 21: RGB Poisson merging result

Experimentally, it was found that such merging works well when pixels near the edge of a region has been colored, as **figure 22**.

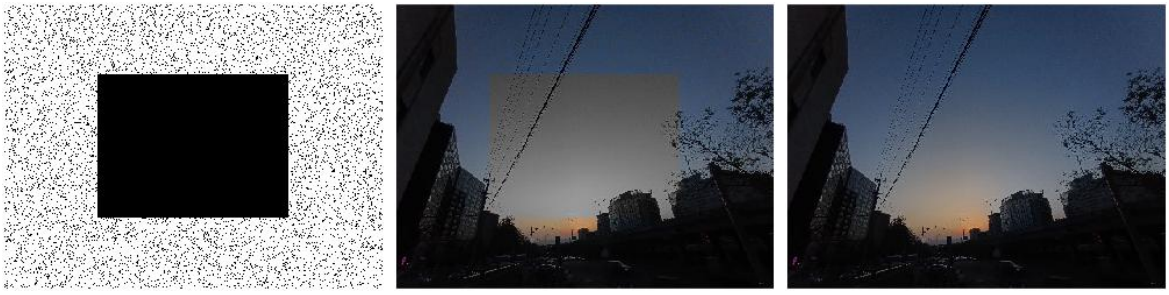

Figure 22: RGB Poisson merging with edges

3.2 Optimizing on YUV Space

3.2.1 Loss Function on YUV

Though such results is acceptable, it cannot be used on YUV space, for the grayscale image doesn't provide gradient information about chrominance.

Nevertheless, the RGB optimizing has 50% more parameters, and cannot ensure the luminance of the result to be the same as target image, so using YUV is a much better option. A form of loss function is equation(8) [4].

$$\begin{aligned} l(X,T) =& J_T(U(X)) + J_T(V(X)) \\ J_T(U) =& \sum_r (u_r - \sum_{s\in\mathcal{N}(r)} w_{rs}u_s)^2 \\ w_{rs} \propto& \exp(-(t_r - t_s)^2/2\sigma_r^2) \\ \sum_s w_{rs} =& 1 \\ \sigma_r =& \mathrm{Var}_{s\in\mathcal{N}(r)}(t_s) \end{aligned} \tag{8}$$

The core of this loss is the weight w_{rs}. Having a form like Gauss filter, it converts luminance information to chrominance information. According to the principle that points having short distance in space and brightness should also be close in color, if all t_s is the same, the loss term u_r becomes the difference of u_r with the average of u_s.

Derive to u_r, we can find that

$$(u_r - \sum_{s\in\mathcal{N}(r)} w_{rs}u_s) + \sum_{r\in\mathcal{N}(s)} w_{sr}(\sum_{t\in\mathcal{N}(s)} w_{st}u_t - u_s) = 0$$

For $r \in \mathcal{N}(t)$ is equal to $t \in \mathcal{N}(r)$, the system of equations is

$$(1 + \sum_{s\in\mathcal{N}(r)} w_{sr}^2)u_r - \sum_{s\in\mathcal{N}(r)} (w_{sr} + w_{rs})u_s + \sum_{s\in\mathcal{N}(r)} \sum_{t\in\mathcal{N}(s)} w_{sr}w_{st}u_t = 0 \tag{9}$$

Counting w_{rs} when i, j is not on the edge can be implemented by

```
function res = w(i, j, ori, T)
    % ori = 1: i - 1
    % ori = 2: i + 1
    % ori = 3: j - 1
    % ori = 4: j + 1
    now = T(i,j);
    ori = int32(ori);
    a = [T(i-1,j),T(i+1,j),T(i,j-1),T(i,j+1)];
    sigma_r = var(a);
    if sigma_r == 0
        sigma_r = 1e-2;
    end
    wt = exp(-(now - a) .^ 2 / (2 * sigma_r ^ 2));
    wt = wt / sum(wt);
    res = wt(ori);
end
```

After similarly but much more complex construction the matrix, we can solve U and V:

```
A = sparse(1:d, 1:d, ones(1, d), d, d);
b = zeros(d, 1);
for i = 1:d
    x = o_to_t(i, 1);
    y = o_to_t(i, 2);
    t = [w(x-1, y, 2, T), w(x+1, y, 1, T), ...
        w(x, y-1, 4, T), w(x, y+1, 3, T)];
    A(i, i) = A(i, i) + sum(t .^ 2);

    if t_to_o(x-1, y) ~= 0
        A(i, t_to_o(x-1, y)) = - t(1) - w(x, y, 1, T);
    else
        b(i) = b(i) + Color(x-1, y) * (t(1) + w(x, y, 1, T));
    end
    ...
        (x+1, y), (x, y-1), (x, y+1) is similar to this
    ...

    if t_to_o(x-2, y) ~= 0
        A(i, t_to_o(x-2, y)) = w(x-1, y, 1, T) * t(1);
    else
        b(i) = b(i) - Color(x-2, y) * w(x-1, y, 1, T) * t(1);
    end
    ...
        (x+2, y), (x, y-2), (x, y+2) is similar to this
    ...

    if t_to_o(x-1, y-1) ~= 0
        A(i, t_to_o(x-1, y-1)) = w(x-1, y, 3, T) * t(1) ...
            + w(x, y-1, 1, T) * t(3);
    else
        b(i) = b(i) - Color(x-1, y-1) ...
            * (w(x-1, y, 3, T) * t(1) + w(x, y-1, 1, T) * t(3));
    end
    ...
        (x+1, y-1), (x-1, y+1), (x+1, y+1) is similar to this
    ...

end

res = A \ b
```

3.2.2 YUV Optimization Results

The algorithm on YUV space has result like **figure 23**.

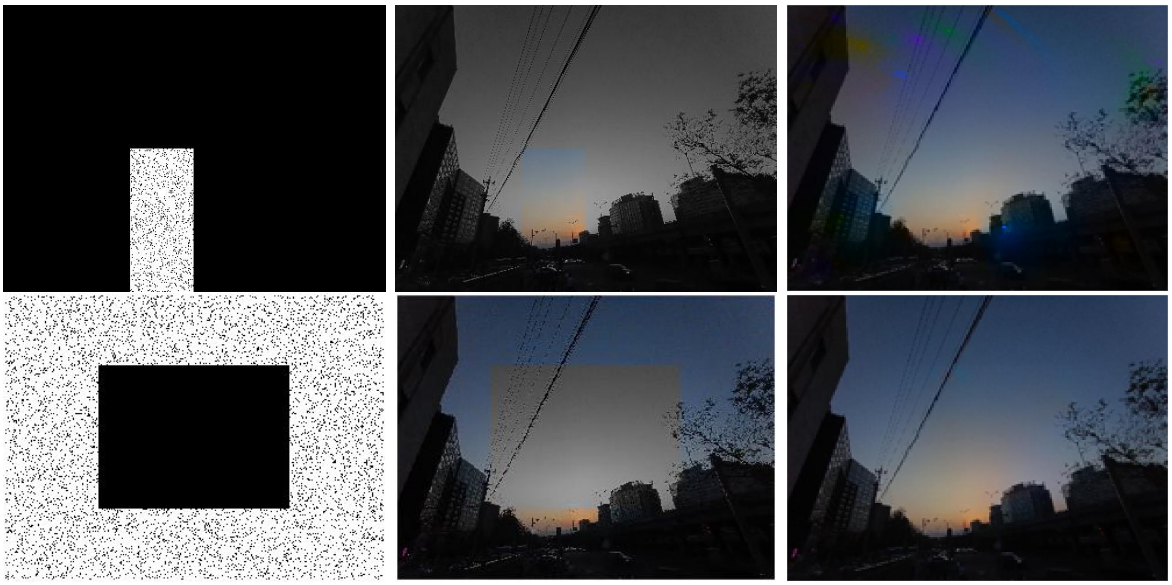

Figure 23: YUV results

Comparing result with RGB, it is more natural, what is more, the color in the masked region can radiate further; **figure 24** are some more results from the original paper.

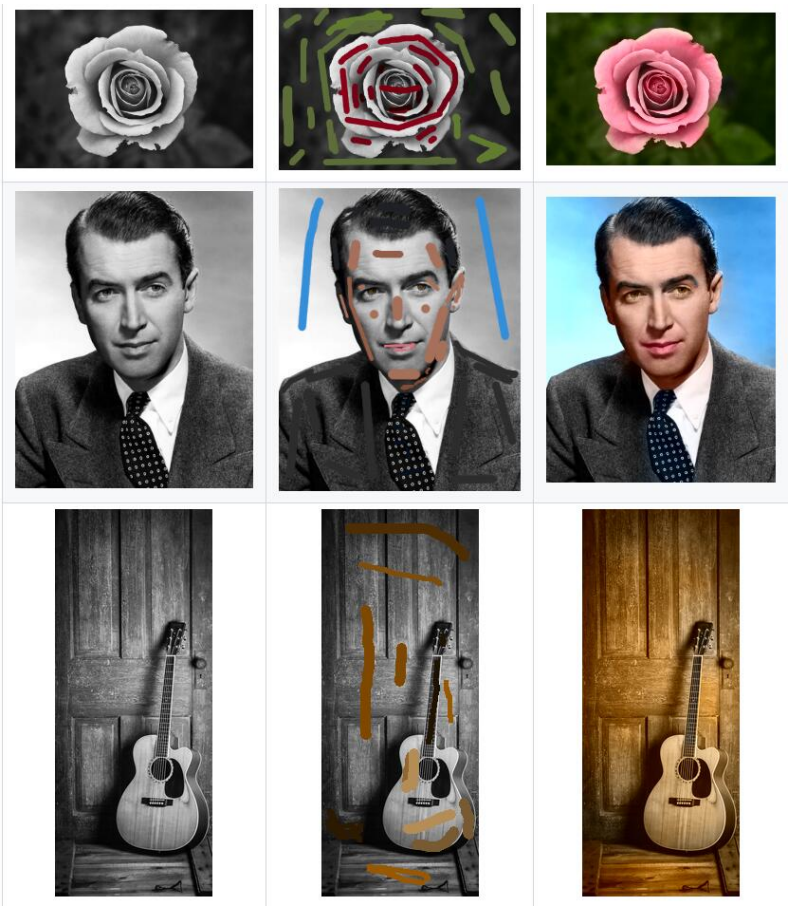

Figure 24: More results [5]

From these results, we can see the benefit of optimization on YUV. The edge information and the chrominance in the masked region can be combined better when solving $U(X)$. That demonstrates the importance in feature-choosing. Though transforming from RGB to YUV is linear, it can already give a more appealing result. Then, can we choose features of an image automatically to get better? This brings us to the realm of deep learning.

4 Introduce of CNN

Figure 25: Different results by different nets [1]

Before studying how can deep learning methods take effect on this problem, there is one thing to know: deep learning methods will almost definitely drop to local optimum solution, regardless of global solution. Thus, there is a high degree of randomness in the results of deep learning, causing it difficult to make precise measures of effectiveness.

Still, considered in terms of large datasets, it has clear measures of overall effect, just as **figure 25** shows.

4.1 Colorization Nets

4.1.1 Basic Classification

There are many net structures to do re-colorize, some of them are interactive, while others are automatic. Overall, there are 7 kinds of re-colorization nets [1]:

1. Plain networks

   These networks directly get U and V through the network, with no skip or naive skip connections. Similar to other CNN tasks, early colorization architectures were plain networks with a simple, straightforward architecture. One of the first attempt in *automatic* grayscale colorization is the colorful image colorization CNN [6], which we will study later.

2. User-guided networks

   User-guided networks require input from the user either in the form of in points, strikes, or scribbles. Our two views of given styles or part of the colors are all user-guided methods, also known as *interactive* methods.

3. Text-based networks

   Similar to user-guided networks, text-based networks need users to give prompts to colorize. But, this method is rather automatic than interactive, for a prompt is a very vague information. We call this kind of methods *semi-interactive*, to note its difference from interactive or automatic methods.

4. Diverse networks

   The aim of diverse colorization is to generate different colorized images, rather than restore the original color. For user need to choose from final results, this method is also semi-interactive. Like text-based networks, these networks tend to utilize a generative approach.

5. Exemplar-based networks

   Exemplar-based colorization utilizes the colors of example images provided along with input grayscale images. This is a semi-interactive method as well, for diverse examples can deliver rich results, and even no examples are given, these networks can be automatically used.

6. Multi-path networks

   The multi-path networks follow multiple different parts to learn features at different levels or different paths. Different parts of these networks may use different micro-networks to get a result.

7. Domain-specific networks

   These networks aren't invented for colorizing normal images, but for images from different modalities such as infrared, or different domains such as radar. These specialized functions lead to some special structures in the network.

Because our topic, we will first take a look at a famous plain network, then at how semi-interactive networks are formed.

4.1.2 Plain Network Constructions

In Colorful Image Colorization CNN, the network takes as input a grayscale image and predicts 313 U-V pairs in the chromatic domain based on empirical probability distributions, which are then converted to U and V channels in YUV color space. (In the paper, this space is named as LAB, which is exactly the same thing.)

It stacks the convolutional layers linearly to form eight blocks, each of which consists of two or three convolutional layers followed by a ReLU layer and a batch normalization layer. To reduce the size of the image, striding technique is used instead of pooling, for pooling might lose too much information. The input to the network is 256×256 while the output is 224×224, then, the output is resized to the original image size, as **figure 26**. Here the loss is not represented as L2 loss, for L2 optimizing is kind of a mean value, resulting in less vivid colors.

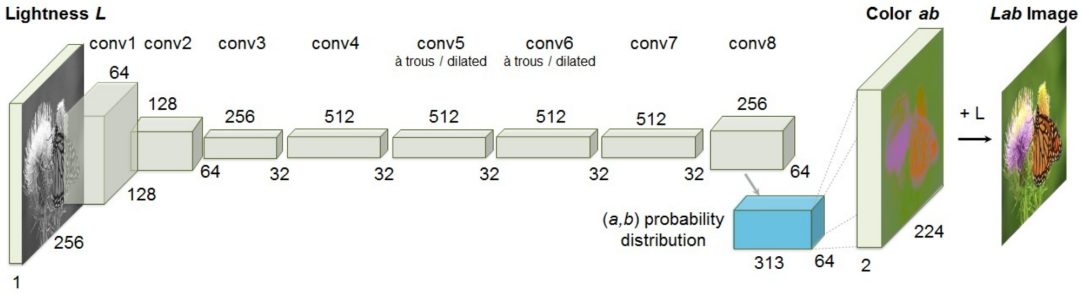

Figure 26: Colorful Image Colorization CNN network construction [6]

While training, Color image coloring removes the effect of low U-V values (due to backgrounds such as sky, clouds, walls, etc.) by re-weighting pixel rarity-based losses. The authors refer to this technique as class rebalancing.

Figure 27: Colorful Image Colorization CNN network results [7]

Some results are in **figure 27**, from which one can tell that it has appealing results when coloring the sky, but for edges and details about ground, the result is evidently unreal. Almost all automatic methods have this difficulty, that's why the most often used nets are semi-interactive.

4.1.3 Colorize by GAN

Generative Adversarial Network (GAN) and Conditional Generative Adversarial Network (CGAN) are widely used to accomplish semi-interactive colorization.

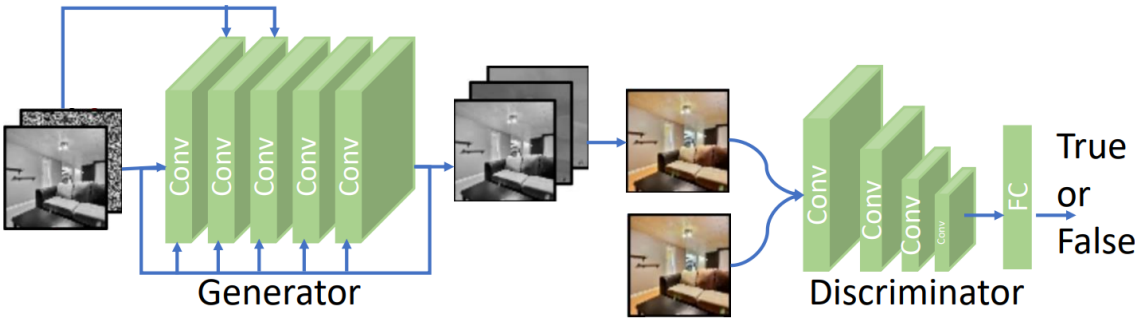

Figure 28: UDCGAN network construction [1]

One of the popular GAN structure is like **figure 28**, in which the generator continuingly generate images for the discriminator to judge. Through training, the image generated gets closer and closer to colorful results.

Unlike the traditional GAN, the generator here takes grayscale input image instead of random white noise. The authors also proposed the generator's modified cost function to maximize the discriminator's probability of being incorrect instead of minimizing the likelihood of being correct. Moreover, the baseline model's generator is used without any modifications, while the discriminator is composed of a series of convolutional layers with batch normalization.

Despite we don't have calculation resources to train a GAN from scratch, the thoughts provided by GAN is useful, such as: using grayscale image as generate beginning; reconstructing the image from net work parameters; and letting the trained network to judge the loss of result.

4.2 VGG-19 and Gram Matrix

4.2.1 VGG-19

Look back to style-transferring problems. To get the semantic information, we can use the network VGG-19 that has been pretrained for target recognition and localization [8].

The overall architecture of this network is as follows, formed by 3×3 convolution layers, ReLU layers, 2×2 max pooling layers, fully connected layers, and a soft-max layer to generate final result: a vector to classify the image.

```
conv_1 relu_1 conv_2 relu_2
pool_3
conv_3 relu_3 conv_4 relu_4
```

```
pool_5
conv_5 relu_5 conv_6 relu_6 conv_7 relu_7 conv_8 relu_8
pool_9
conv_9 relu_9 conv_10 relu_10 conv_11 relu_11 conv_12 relu_12
pool_13
conv_13 relu_13 conv_14 relu_14 conv_15 relu_15 conv_16 relu_16
pool_17
fc_17 fc_18 fc_19
softmax
```

Three adjustments are made to the standard VGG-19 network in the paper:

First, the paper argues that both semantic and stylistic features are sufficiently determined by the output of the convolutional layer, and thus only the computational results of the convolutional layer can be retained, so the fully-connected layer, which has a high computational complexity, can be discarded.

Second, the paper normalizes all the outputs in order to facilitate the retention of the weights of the content with respect to the stylistic migrations in the subsequent control, which corresponds to a uniform linear transformation for all the results, while ReLU and pooling will retain such a global linear transformation, and therefore will not affect the output.

Finally, in the image synthesis task, the average pooling (i.e., divided into a 2×2 grid to take the average value) will perform better than the maximum pooling, and therefore the maximum pooling will be replaced by the average pooling.

4.2.2 Representation of Metrics

For a convolutional neural network, the output of each layer is in fact a matrix, with each position of the matrix corresponding to a nonlinear function about the original image. We write down the output of the lth convolutional layer for the image X as $F^l(X)$, which has a size of $n_l \times m_l$. Since the output of the convolutional layer is assumed to contain all the feature information, d_c and d_s can be written as

$$\begin{aligned} c(X) =& \big(a^1 c^1(X), \dots, a^l c^l(X), \dots\big) \\ s(X) =& \big(b^1 s^1(X), \dots, b^l s^l(X), \dots\big) \\ d_c(X,T) =& d(C(X), C(T)) = \sum_l \|a^l c^l(X) - a^l c^l(T)\|_F^2 \\ d_s(X,S) =& d(S(X), S(T)) = \sum_l \|b^l s^l(X) - b^l s^l(X)\|_F^2 \end{aligned} \tag{10}$$

Here each $c^l(X), s^l(X)$ is a matrix.

Content representation $c^l(X)$ can directly use $F^l(X)$, so content distance becomes (w^c is the weight):

$$d_c(X,T) = \sum_l w_l^c \|F^l(X) - F^l(T)\|_F^2$$

By experiments, $s^l(X)$ can use the *gram matrix* of $F^l(X)$, that is $F^l(X)(F^l(X))^T$, so

$$d_s(X,S) = \sum_l w_l^s \|F^l(X)(F^l(X))^T - F^l(S)(F^l(S))^T\|_F^2$$

To avoid the influence of size, the Frobenius norm can be normalized, which means to divide it by the size of the matrix.

4.2.3 Result Generation

From the input grayscale image, we can use the loss function to optimize the result. $\alpha d_c + \beta d_s$ at each layer is a quadratic function of $F^l(X)_{ij}$, whose derivative is easy to calculate. For we only used convolution layers and ReLU layers, the way to generate $F^l(X)$ is by linear or segmented linear functions, also easy to derive.

Using

$$\frac{\partial L(X,T,S)}{\partial X} = \sum_{ijl} \frac{\partial L}{\partial F_{ij}^l} \frac{\partial F_{ij}^l(X)}{\partial X}$$

together with gradient descent, the optimization can be applied to the grayscale target image, until the loss function converges.

By adding loss layers and using optimizer provided by PyTorch, the derivation can be automatically counted. The loss layers are defined as [9]:

```
class ContentLoss(nn.Module):
    def forward(self, input):
        self.loss = self.criterion(input * self.weight, self.target)
        self.output = input
        return self.output

    def backward(self, retain_graph=True):
        self.loss.backward(retain_graph=retain_graph)
        return self.loss

class GramMatrix(nn.Module):

        def forward(self, input):
            a, b, c, d = input.size()  # a=batch size(=1)
            # b = number of feature maps
            # (c,d) = dimensions of a f. map (N=c*d)
            features = input.view(a * b, c * d)  # resise F_XL into \hat F_XL
            G = torch.mm(features, features.t())  # compute the gram product
            # 'normalize' the values of the gram matrix
            return G.div(a * b * c * d)

class StyleLoss(nn.Module):
    def forward(self, input):
        self.output = input.clone()
        self.G = self.gram(input)
        self.G.mul_(self.weight)
        self.loss = self.criterion(self.G, self.target)
        return self.output
```

```
def backward(self, retain_graph=True):
    self.loss.backward(retain_graph=retain_graph)
    return self.loss
```

For a better comparison, we use **figure 29** as the target image and the style image.

Figure 29: Style transferring from artworks

4.3 Implementation of Transferring

4.3.1 Inserting Loss Layers

Because we need to descend from different layers, the backward function must retain the graph. To control different weight at different layers, we insert style loss and content loss after every convolution layer when the network is constructed:

```
i = 1
# cnn include the layers of vgg-19
for layer in list(cnn):
    if isinstance(layer, nn.Conv2d):
        name = "conv_" + str(i)
        model.add_module(name, layer)
    elif isinstance(layer, nn.ReLU):
        name = "relu_" + str(i)
        model.add_module(name, layer)
        i += 1
    elif isinstance(layer, nn.MaxPool2d):
        name = "pool_" + str(i)
        model.add_module(name, layer)
    else:
        break

    # content_layers include the layers used by dc and the weight
    if name in content_layers:
        # add content loss:
        target = model(content_img).clone()
```

```
            content_loss = ContentLoss(target, content_weight)
            model.add_module("content_loss_" + str(i), content_loss)
            content_losses.append(content_loss)

        # style_layers include the layers used by ds and the weight
        if name in style_layers:
            # add style loss:
            target_feature = model(style_img).clone()
            target_feature_gram = gram(target_feature)
            style_loss = StyleLoss(target_feature_gram, style_weight)
            model.add_module("style_loss_" + str(i), style_loss)
            style_losses.append(style_loss)
```

Specifically, we replicate each layer before the VGG19 fully connected layer, and if this layer is to be used as a Content Loss computation, an additional Content Loss layer is inserted after this layer, and the same is done for Style Loss.

It is worth mentioning that since the model is constructed to the current layer exactly when the output of the current layer is available, $F_{ij}^{l}(T)$ and $G_{ij}^{l}(S)$ can be computed and passed in directly before the construction of the Content Loss or Style Loss layer.

4.3.2 Some Improvements

A result by this method is **figure 30**, the 6 images shows the result at epoch 0, 100, 200, 300, 400 and 500.

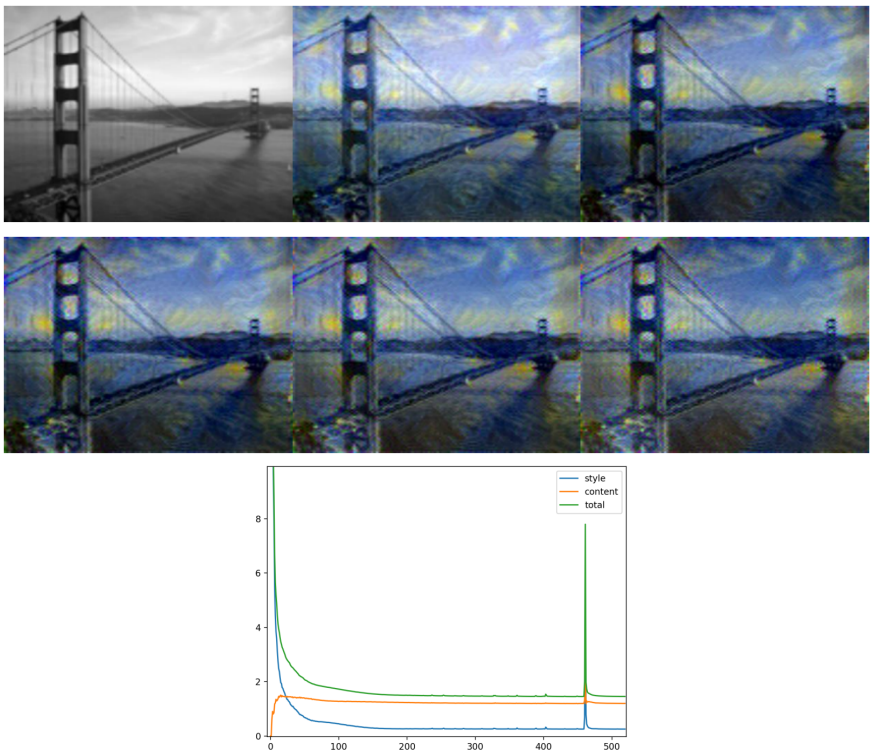

Figure 30: Result and its loss curve

According to the paper, using an average pooling layer will give better results than simple maximum pooling, however, since VGG is trained by maximum pooling, there is no average pooling corresponding to the parameters of the convolutional layer, which can only be modified here directly at the time of construction:

```
elif isinstance(layer, nn.MaxPool2d):
    name = "pool_" + str(i)
    model.add_module(name, nn.AvgPool2d(kernel_size=layer.kernel_size))
```

We originally chose the LBFGS optimizer, which is a pseudo-Newtonian method. But since the ReLU function is not second-order smooth, the second-order matrix modeled by the pseudo-Newtonian method is likely to be singular and accumulate errors. In addition, since the MSE of the Gram Matrix and the MSE of the coefficient matrix are both convex functions on the coefficients, there is in fact not much likelihood of falling into a local optimum that would require a relatively drastic reset (which is what causes the sudden rise in error for 400 to 500 iterations). Therefore, the Adam optimizer can be used directly. That gives rise to **figure 31**.

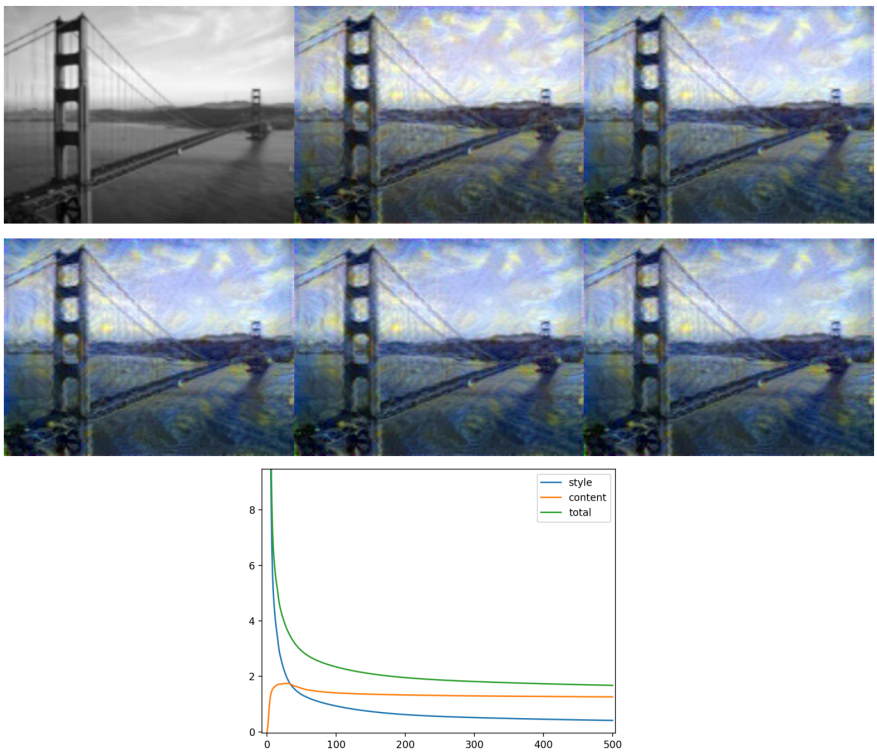

Figure 31: Result and its loss curve with Adam optimizer

In terms of both the loss function and the pictures, it is clear that the Adam optimizer changes are smoother. However, it is also intuitively obvious from the colors that Adam will converge quite a bit slower than the second-order method, but after becoming smoother, it is easier to employ an adaptive stopping strategy with a sufficiently large number of iterations.

Different ways to initialize also lead to different characteristics in the result. If we want to partially preserve the location of the color, we can average the content image and style image as the input, like **figure 32**.

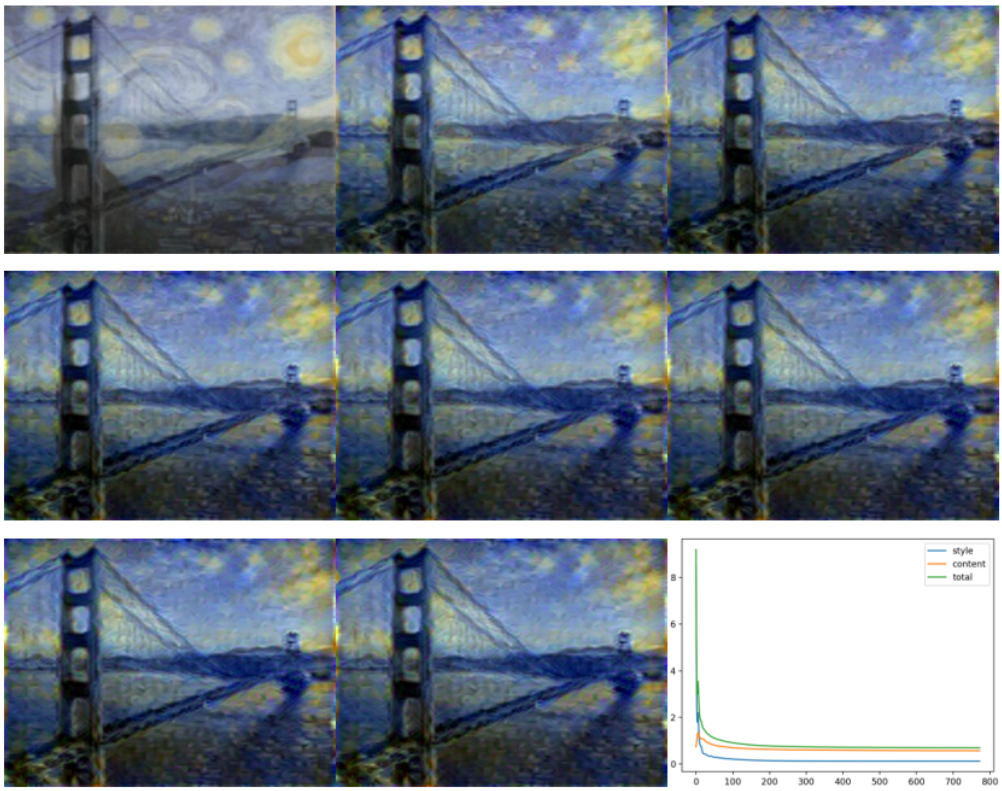

Figure 32: Result and its loss curve while average initialized

Experiments have shown that average initialing can not only preserve location information but also fasten the convergence, therefore, all subsequent initialization will be like this.

4.4 Results and Semi-Interactive Methods

To show more influences by different weights, we use **figure 33** as the example. The right image is complex in information and colors, but has certain style message.

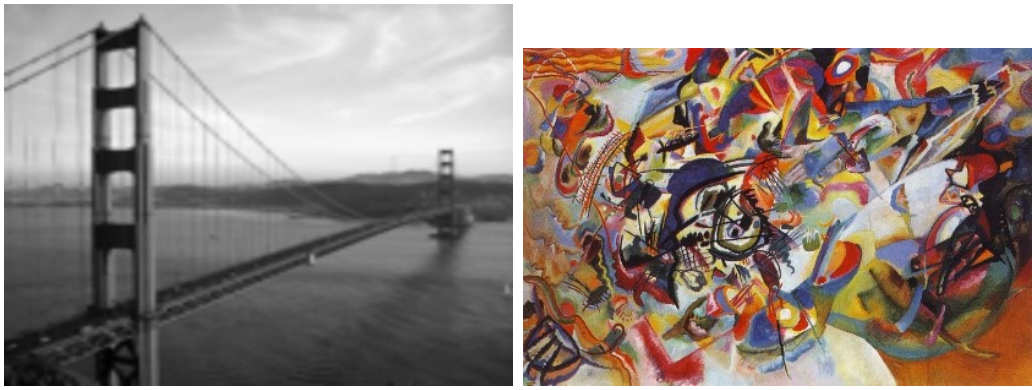

Figure 33: Another case of transferring

For this method is a combination of style-transferring and optimization, changing content weights and style weights are actually modifying the loss function.

4.4.1 Content Weights

Let style weight be 500 at the first 5 convolution layers, place content weight 1 at convolution layer 1, 3, 5, 9, 13 and 16 (these layers are essentially after different count of pooling layers), the result is **figure 34**.

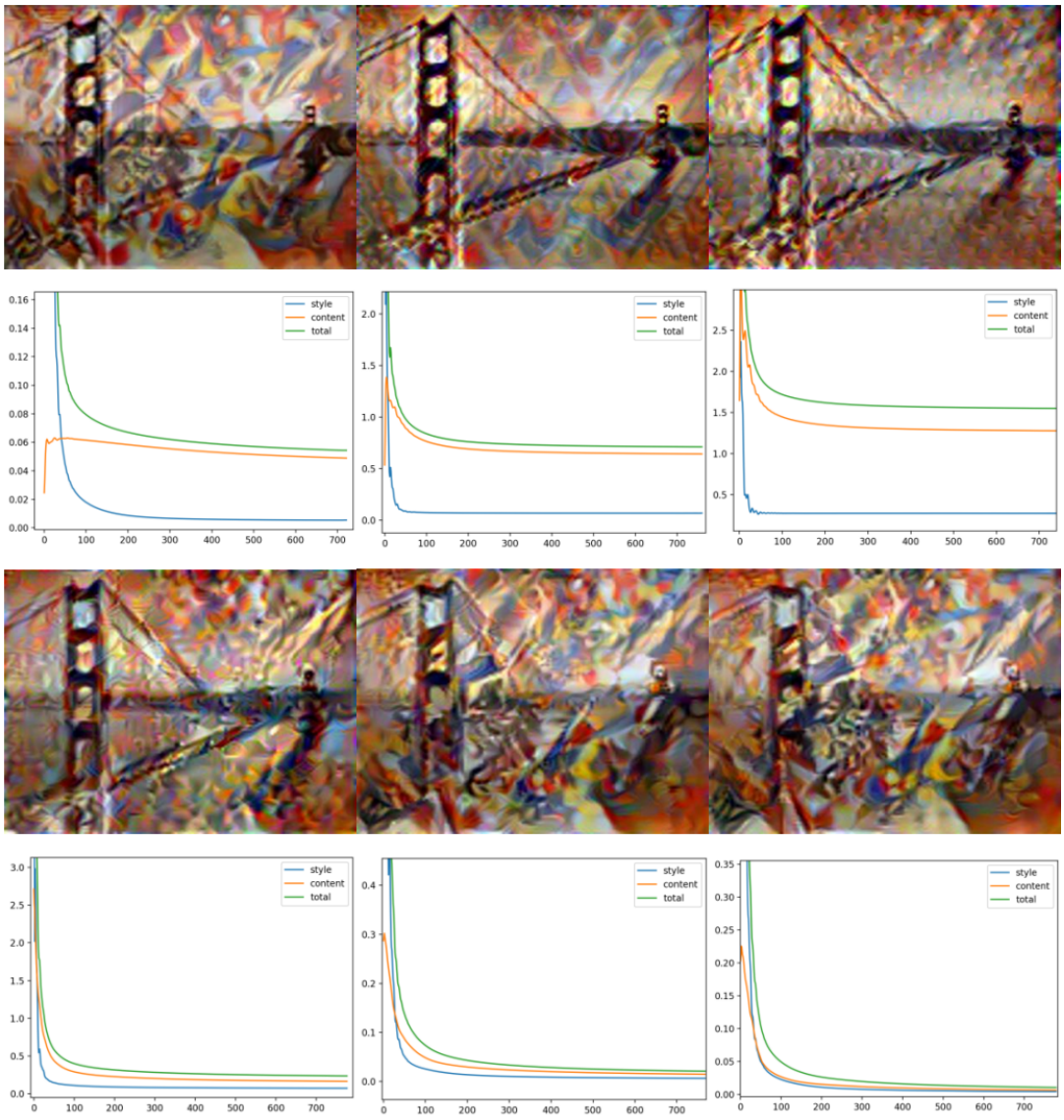

Figure 34: Content loss layer's affection

When the layers are low, the content weights have more influence, so the iterations restore more of the content and end up with smaller losses. While the layers are high, the content weights contain few degrees of freedom and are separated from the styles, making it easy to iterate to a situation with much smaller errors. When content loss layer and the highest numbered style loss layer (we name it as *intrinsic layer*) are the same as the fifth convolution layer, this layer appears to be heavily couple, and therefore the parameters of this layer produce a great loss.

In terms of the effect of the results, the higher the level of content corresponds to the higher the level of semantic information, and therefore the worse the basic information of the

picture is maintained. However, the results show that when the content loss layer is too near the beginning, it may be difficult to achieve overall optimization, but only local distortion, so the content loss layer should be close to the intrinsic layer, in order to ensure a more thorough optimization in the same layer.

4.4.2 Style Weights

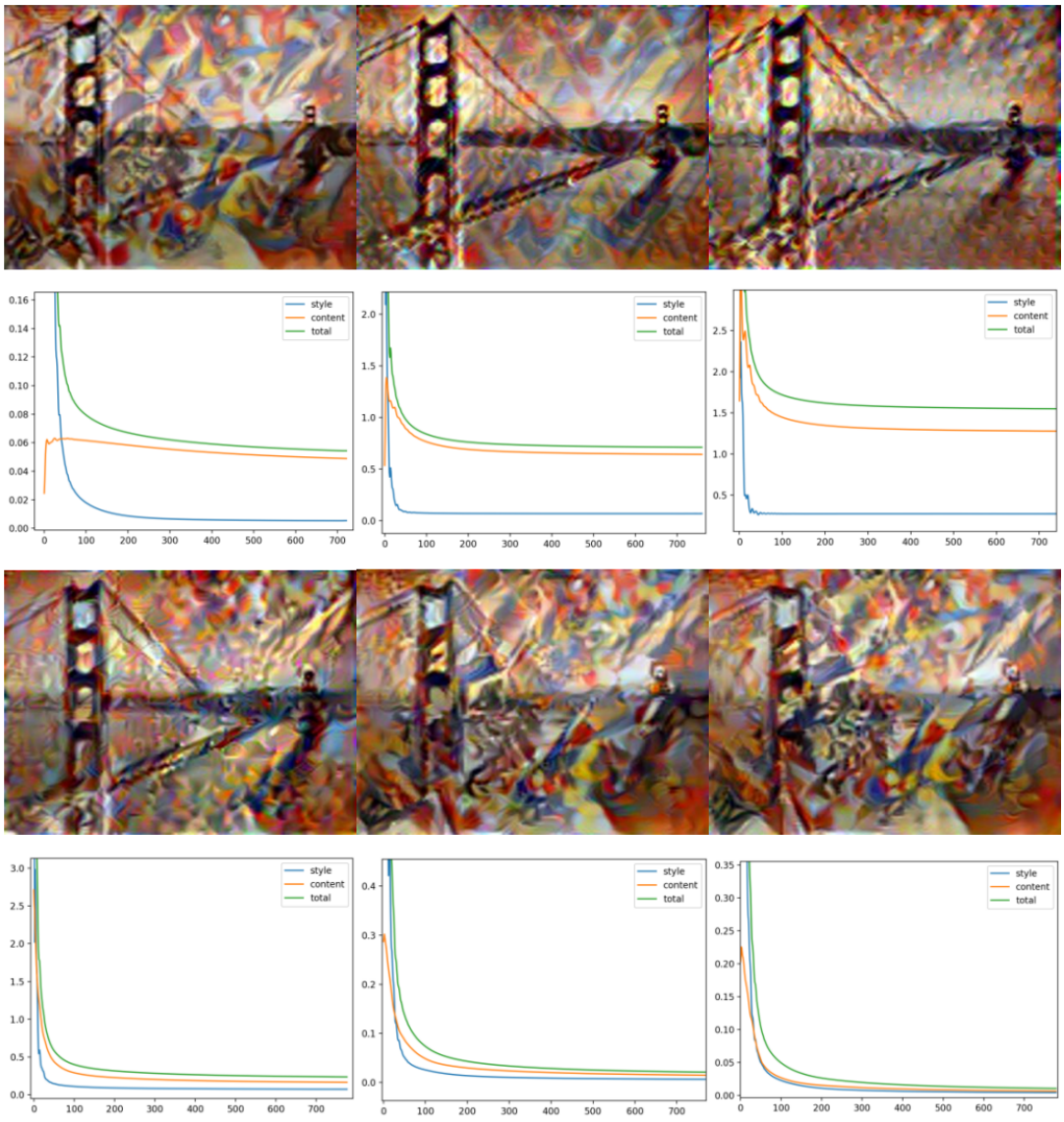

Figure 35: Style loss layers' affection

Let the content loss layer be the fifth convolution layer, and its weight be 1. The style loss layers in the six examples in **figure 35** is in order:

```
{'conv_2':500, 'conv_3':500, 'conv_4':500, 'conv_5':500, 'conv_6':500}
{'conv_3':500, 'conv_4':500, 'conv_5':500, 'conv_6':500, 'conv_7':500}
{'conv_4':500, 'conv_5':500, 'conv_6':500, 'conv_7':500, 'conv_8':500}
{'conv_1':200, 'conv_2':400, 'conv_3':600, 'conv_4':800, 'conv_5':1000}
```

```
{'conv_1':1000, 'conv_2':800, 'conv_3':600, 'conv_4':400, 'conv_5':200}
{'conv_1':300, 'conv_2':600, 'conv_3':900, 'conv_4':600, 'conv_5':300}
```

Since styles are judged by the gram matrix, which ignores positional information, it is intuitively clear that adjustments to styles lead more to changes in overall color.

In the process of moving the style loss layer back, coupling with the fifth convolution layer (where the content loss layer is located) gradually increases, and the error in this layer becomes larger, making it more difficult to keep the content consistent. This is congruent with the results of changing the content loss layer, and still implies that the content loss layer should be near the intrinsic layer.

With that in mind, adjustments to the weights have relatively little effect on the results. Overall, the earlier the center of the style loss is, the better the restoration of the content, while the later it is, the more it will distort towards the source image. For better results, the center of the style loss can be moved forward, but the intrinsic layer remain adjacent to content loss layer.

4.4.3 Semi-interactive Colorization

Going back to the re-colorization problem before, we can get result like **figure 36**.

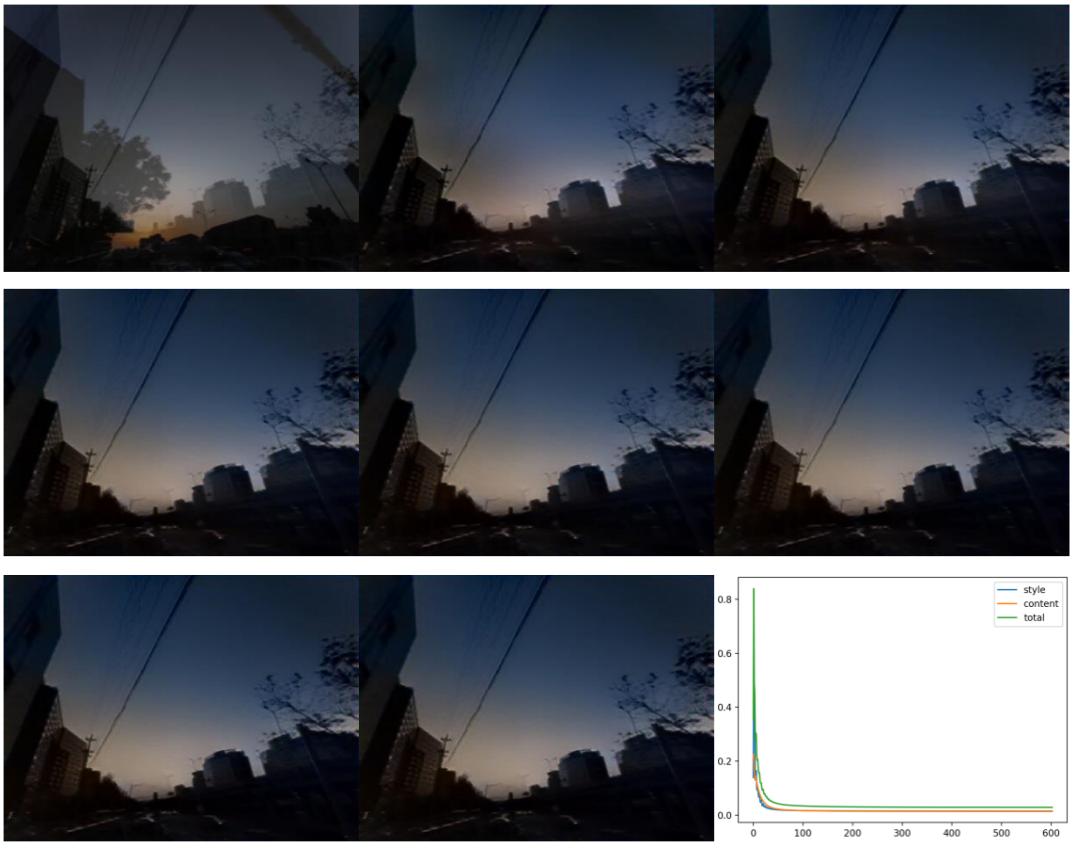

Figure 36: Colorization by CNN

This result is compressed to 256×256, for more realistic results, compress the image to 512×512, set content weight to 0.5 and fully iterative, we can get **figure 37**. It is significantly better than the matching methods before.

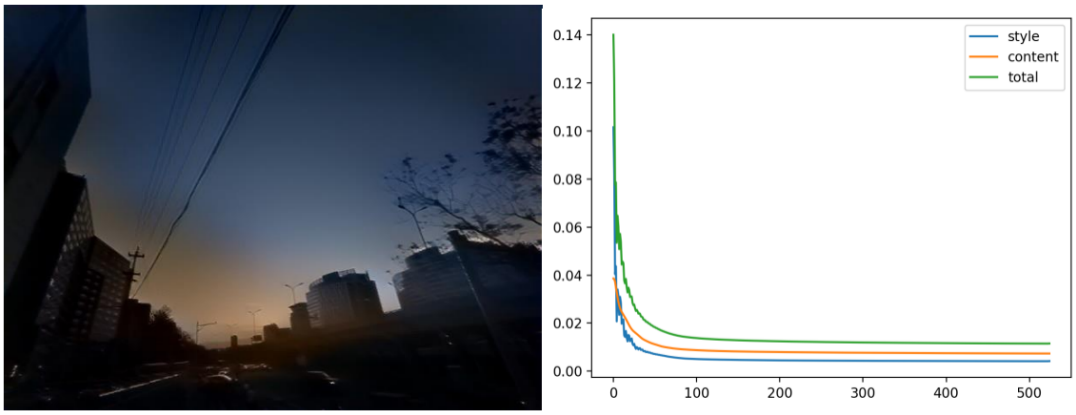

Figure 37: Final colorization result

With an image gallery, the method can also be semi-interactive. We can either use deep learning methods (GAN or CGAN) to generate a style image or find the image with the nearest content ($\text{argmin}_{X\in\mathcal{G}}\|F^l(X) - F^l(T)\|_F$ for some l), and then do transfer. The second way is easy to accomplish and reasonable: for the CNN has already distinguished the semantic information, to count distance on it can find the image with the closest meaning.

5 Conclusion and Discussion

5.1 More Details

5.1.1 Large Datasets and Large Models

Our method can be easily run on personal computers with a normal CPU due to the substantial arithmetic savings of the pretrained model. But such savings come at a price: we can't fine-tune the model to get better results, much less design it from scratch to fit the scene.

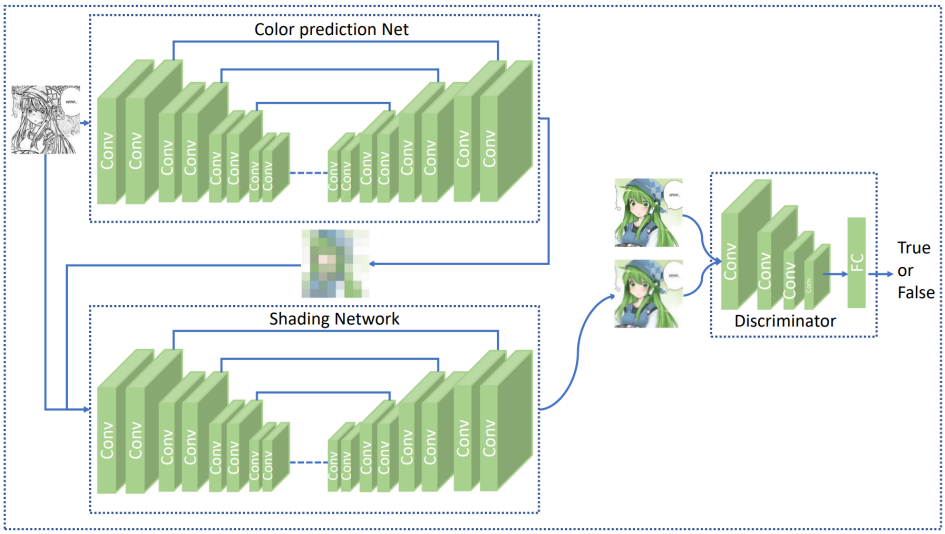

Figure 38: TANDEM architecture [1]

One of the possible improvement that mentioned before is to use GAN to generate a style image rather than choose one in the gallery. By this, the method can be fully automized, as **figure 38** shows, we can replace the shading network part by transferring method. Also, if we have a result provided by a plain network, use transferring to optimize will possibly get a more precise result.

5.1.2 Judging Results

Not every image can be used to judge the result. For instance, to colorize a flower or a car to any color is reasonable. A sample test dataset is like **figure 39**, use things that have certain color in real-life to judge the result.

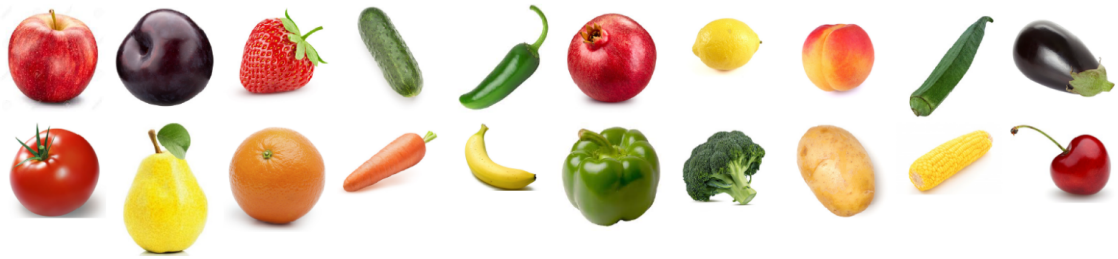

Figure 39: Some datas [1]

The metrics typically used to assess colorization quality are either subjective or commonly used mathematical metrics such as PSNR and SSIM. Subjective evaluation is the gold standard for many applications where humans determine the accuracy of the output of the algorithms, for the quality of colorization is never an objective thing.

5.2 Summary

5.2.1 Comparison

We have gone through four kinds of different methods, they're: traditional style-transferring (with matching or frequency), traditional optimization (on RGB or YUV), colorization nets, and the CNN-based optimization method. On different dimensions, they have their advantages and weaknesses.

1. Matching methods

   Matching is fast and can transfer the overall color style to target image. However, it relies on discretion and ignores the location information. If the histograms of the source image and target image have a huge difference, the result tends to be "abrupt".

2. Wavelet merging methods

   Merging by wavelet coefficient is also fast and can preserve location information, but it does too well that the semantic of the source image is easily carried. In spite of this, the pseudo-edges are difficult to deal with.

3. Poisson merging methods

   Poisson merging does much better on the continuity of color, but have to face the decay of it. The gradient information of RGB are the same for target grayscale image, making it harder to colorize by gradient. Still, having the edge been colored will let Poisson merge to have a much appealing result.

4. YUV optimization

   YUV optimization is more natural, but how Y can effect on U, V is totally not intuitive. Every row of sparse matrix A has at most 13 nonzero elements, while in Poisson merging the number is 5, making it even slower. Despite, it's comforting that with main tones be colored, YUV optimization can get a genuine real result.

5. Colorization networks

   This kind of methods are hardly the same, but the main problem is the calculation resources spent while training the network. If we can ignore the cost, deep learning can give an arbitrarily good result.

6. CNN-based optimization

   This method is our paper mainly talked about, it combines the advantages of the methods before: using deep learning results without training the net, using optimization without offering partial color, and accomplishing matching with a computable loss function. It is relatively slow and lack flexibility, but a good method on balance.

5.2.2 Future Enhancements

One of the most importance enhancement has been talked about in the last subsection, to apply larger datasets and models. Besides, we can also give better solutions by:

1. Integrate additional methods to validate and improve results;
2. Score multiple times in the test set to obtain the optimal parameters for the model;
3. Appropriate preprocessing of the data, for instance, eliminating semantic information from the style image to facilitate the fusion of styles;
4. Adjust the loss of style-transferring to make it more adaptable to re-colorization problems.

After all, from the most basic matching to the most complex neural networks, the fast improvement in colorization shows our love and graceful for this colorful world.

References

[1] S. Anwar, M. Tahir, C. Li, A. Mian, F. S. Khan, and A. W. Muzaffar, "Image colorization: A survey and dataset," *arXiv preprint arXiv:2008.10774*, 2020.

[2] J. Zhang, "Math experiments," 2022.

[3] X. Li and F. Chen, *Basic Wavelet Analysis: From Theory to Applications*. preprint, 2023.

[4] A. L. D. L. Y. Weiss, "Colorization using optimization," *SIGGRAPH*, 2004.

[5] lightalchemist, "colorize-image." https://github.com/lightalchemist/colorize-image, 2019.

[6] R. Zhang, P. Isola, and A. A. Efros, "Colorful image colorization," in *ECCV*, 2016.

[7] R. Zhang, J.-Y. Zhu, P. Isola, X. Geng, A. S. Lin, T. Yu, and A. A. Efros, "Real-time user-guided image colorization with learned deep priors," *ACM Transactions on Graphics (TOG)*, vol. 9, no. 4, 2017.

[8] L. A. Gatys, A. S. Ecker, and M. Bethge, "Image style transfer using convolutional neural networks," in *CVPR*, 2016.

[9] enomotokenji, "pytorch-neural-style-transfer." https://github.com/enomotokenji/pytorch-Neural-Style-Transfer, 2020.

Appendix - Files

```
src
    figs                [sample images]
        color.jpg           color style 1
        color2.jpg          color style 2
        gray.jpg            grayscale image 1
        gray2.jpg           grayscale image 2
        real.jpg            real color of image 1
        real2.jpg           real color of image 2
    matlab              [matlab codes]
        draw_pdf.m          draw histograms of RGB
        edge_detection.m    show edge decetion by convolution
        mask_show.m         show optimization with RGB
        mask_show_YUV.m     show optimization with YUV
        match.m             main function of matching
        match_main.m        show matching results
        merge.m             main function of wavelet merging
        merge_main.m        show merging results
        merge_show.m        show wavelet coefficients
        RGB_optimize.m      main function of optimization with RGB
        YUV_optimize.m      main function of optimization with YUV
    python              [python codes]
        loss.py             define loss layers
        style_transfer.py   main function of CNN style transfering
        utils.py            load and write images
```