Paper: IMAGE RECOLORIZATION FOR THE COLORBLIND, Jia-Bin Huang, Chu-Song Chen et al
About the paper:

3 types of CVD(Color vision deficiency):

1. Trichromacy: a mild color deficiency, is often characterized by the defect of one of the three cones
2. Dichromacy: Dichromacy, a more severe color deficiency, is present when one of the three cone types is absent
3. Monochromacy: Monochromacy is the severest but rarest type of CVD, lacking all types of cone cells and perceiving brightness variation only

Previously used methods:

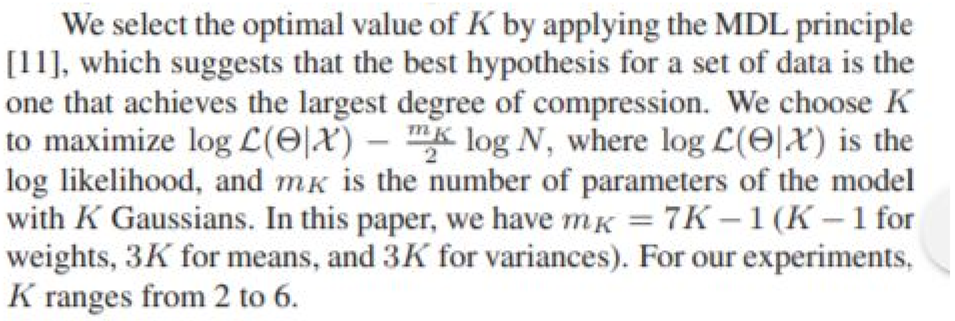
1. Describe the color information in the original image through color sampling to alleviate the computational burden. These representing colors are called “key colors”.
2. Define the target distance. The Euclidean distance between two key colors in the color space is usually used.
3. Obtain the optimal mapping of these key colors through minimizing the difference between the original distances and the perceived distances by CVD viewers for every pair of key colors.
4. Interpolate between the corrected key colors through a distance-weighted nearest neighbor interpolation to compute the final CVD corrected image.

Problems:

1. Representing color information in the original image using color quantization may not be depictive enough
2. The second problem with existing approaches is that the recoloring process is excruciatingly slow, which prohibits itself from many applications

Approach in the paper

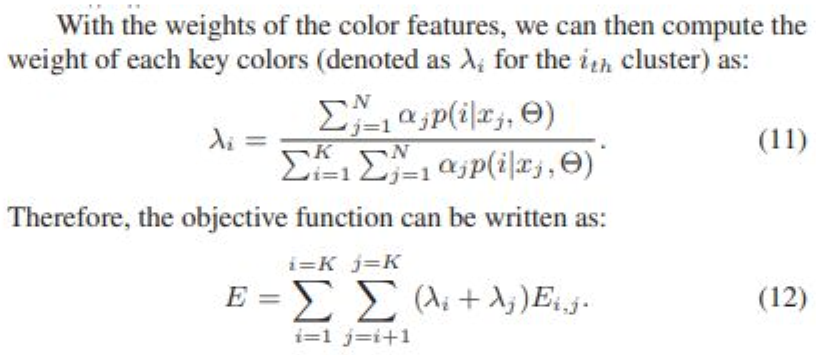
1. Describe the color features using GMM modelling, which is refined using the EM algorithm. The number of gaussians in the mixture can be approximated using MDL(minimum description length) principle.
2. Distance is described using the KL divergence scheme which is a measure of the dissimilarity between two prob density functions
3. Weights have been added to every color, during objective function calculation. Weights are a measure of the difference between the actual and the stimulated color. Higher the weight, higher the change in colour we need, higher the difference.
4. Gaussian mapping for interpolation. In the L*a*b* space, rotation is only done in the a*b* plane for color change, To avoid producing an unnatural corrected image

5. 

We select the optimal value of K by applying the MDL principle [11], which suggests that the best hypothesis for a set of data is the one that achieves the largest degree of compression. We choose K to maximize $\log \mathcal{L}(\Theta|\mathcal{X}) - \frac{m_K}{2} \log N$, where $\log \mathcal{L}(\Theta|\mathcal{X})$ is the log likelihood, and m_K is the number of parameters of the model with K Gaussians. In this paper, we have $m_K = 7K - 1$ ($K-1$ for weights, $3K$ for means, and $3K$ for variances). For our experiments, K ranges from 2 to 6.

Points to note:

1. To avoid producing an unnatural corrected image, we restrict the mapping functions to be rotation operators in the a*b* plane.
2. The error introduced by the ith and jth key colors is defined as:
    a. Ei,j = [DsKL(Gi, Gj)−DsKL(Sim(Mi(Gi)), Sim(Mj (Gj)))]^2
3. The objective function can be computed by summing all errors introduced by each pair of key color, and adding weights to these colors



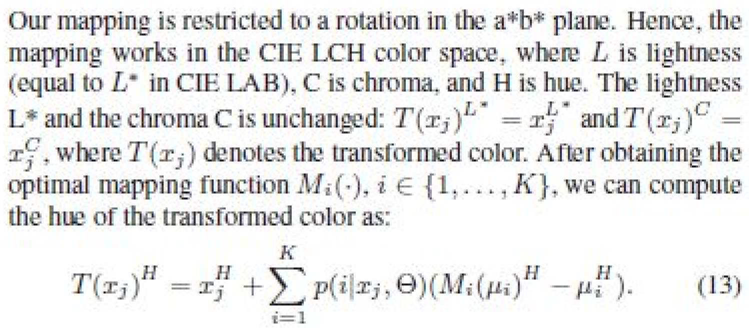
With the weights of the color features, we can then compute the weight of each key colors (denoted as λ_i for the i_{th} cluster) as:

$$\lambda_i = \frac{\sum_{j=1}^{N} \alpha_j p(i|x_j, \Theta)}{\sum_{i=1}^{K} \sum_{j=1}^{N} \alpha_j p(i|x_j, \Theta)}. \qquad (11)$$

Therefore, the objective function can be written as:

$$E = \sum_{i=1}^{i=K} \sum_{j=i+1}^{j=K} (\lambda_i + \lambda_j) E_{i,j}. \qquad (12)$$

4. Interpolation using posterior probability.



Our mapping is restricted to a rotation in the a*b* plane. Hence, the mapping works in the CIE LCH color space, where L is lightness (equal to L^* in CIE LAB), C is chroma, and H is hue. The lightness L* and the chroma C is unchanged: $T(x_j)^{L^*} = x_j^{L^*}$ and $T(x_j)^C = x_j^C$, where $T(x_j)$ denotes the transformed color. After obtaining the optimal mapping function $M_i(\cdot)$, $i \in \{1, \ldots, K\}$, we can compute the hue of the transformed color as:

$$T(x_j)^H = x_j^H + \sum_{i=1}^{K} p(i|x_j, \Theta)(M_i(\mu_i)^H - \mu_i^H). \qquad (13)$$

The “**posterior probability**” of a **Gaussian** is the **probability** that this **Gaussian** generated the data we observe

Paper procedure runtime:

It takes O(KN) for key color extraction using GMM, O(K2) for optimization process, and O(KN) for interpolation, where K represents the number of Gaussian in the mixture, and N is the number of pixels in the image.

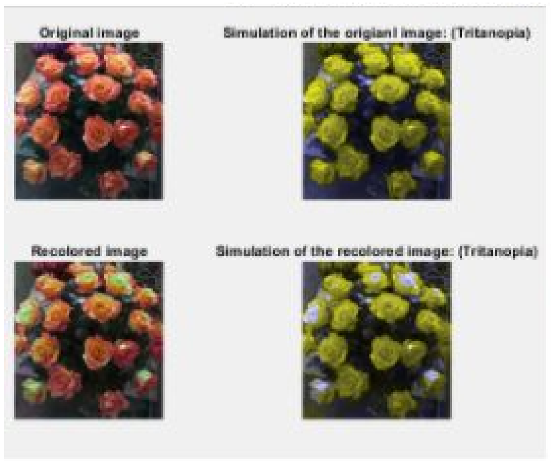
Code for the paper: https://github.com/jbhuang0604/RecolorForColorblind

About the code:

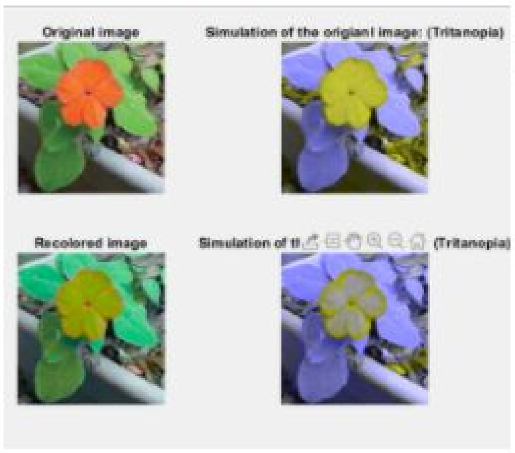
1. demo.m : the main run file, calls other functionalities.
2. simColorBlindImg:
    a. Gamma correctness with gamma = 2.2
    b. Convert to lms color space using pre defined matrices
    c. Compute LMS values perceived by color blind using pre defined projection matrices.
    d. Convert back to RGB.
    e. Gamma inverse-corrected
3. ImgRecolor:
    a. Resize to half the image shape
    b. Convert to CIE lab color space
    c. Fit GMM using BIC approach
    d. Optimise using objective function, compute the rotations angle in CIELAB image color space
    e. Weighted interpolation using posterior probability

Shortcomings in the code(and the paper):

1. The model is not robust at all. It seems to be only working on a very small subset of images.
2. Here are some images on which the model does not work as expected.


Original image
Simulation of the origiani image: (Tritanopia)
Recolored image
Simulation of the recolored image: (Tritanopia)

Noise



Inaccurate color transformation

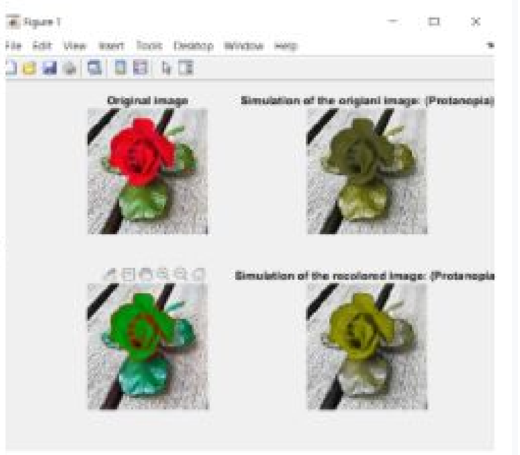


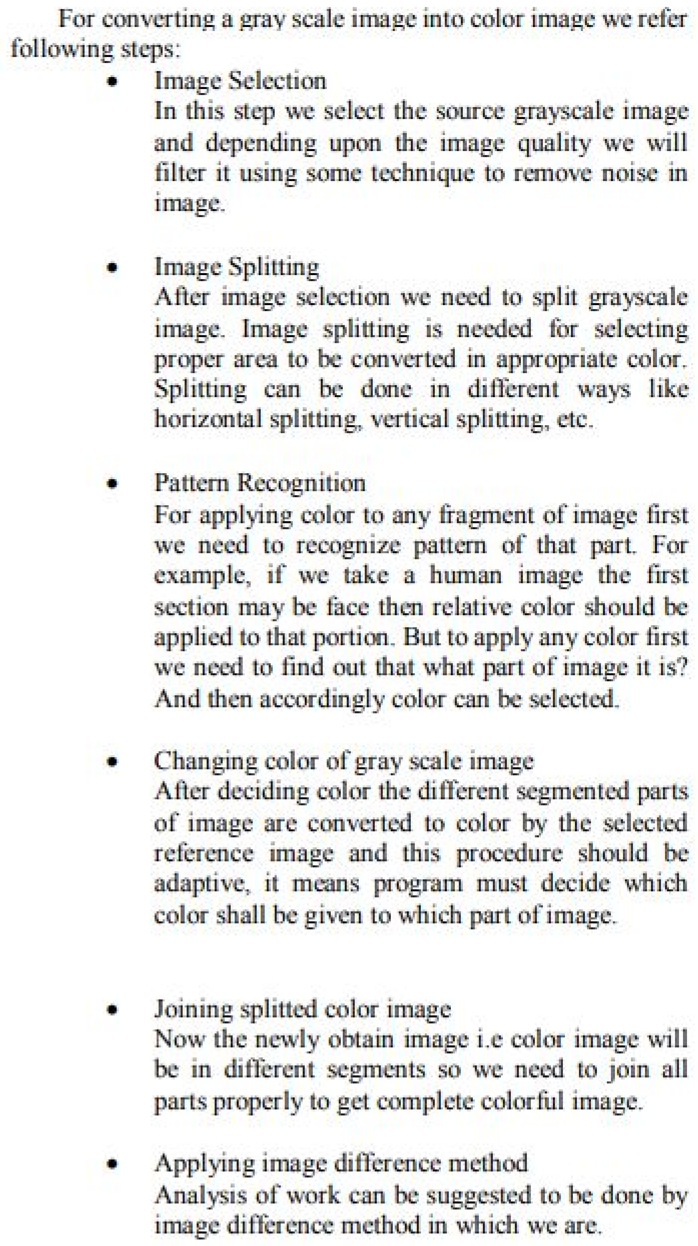
Figure 1
File Edit View Insert Tools Desktop Window Help
Original image
Simulation of the origiani image: (Protanopia)
Simulation of the recolored image: (Protanopia

Noise

I then researched on the current DL approaches that are being used to solve this problem.

Current DL approaches:

Papers

1. http://ijarcet.org/wp-content/uploads/IJARCET-VOL-3-ISSUE-1-143-146.pdf
    a. Talk about the approach but provide no details of how to do it.
    b. 

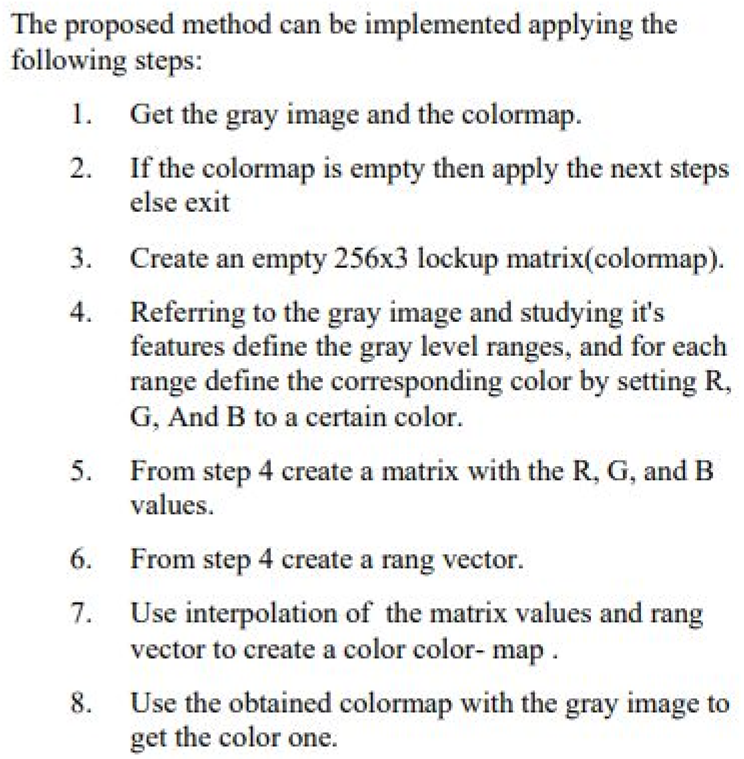
For converting a gray scale image into color image we refer following steps:

- Image Selection
  In this step we select the source grayscale image and depending upon the image quality we will filter it using some technique to remove noise in image.
- Image Splitting
  After image selection we need to split grayscale image. Image splitting is needed for selecting proper area to be converted in appropriate color. Splitting can be done in different ways like horizontal splitting, vertical splitting, etc.
- Pattern Recognition
  For applying color to any fragment of image first we need to recognize pattern of that part. For example, if we take a human image the first section may be face then relative color should be applied to that portion. But to apply any color first we need to find out that what part of image it is? And then accordingly color can be selected.
- Changing color of gray scale image
  After deciding color the different segmented parts of image are converted to color by the selected reference image and this procedure should be adaptive, it means program must decide which color shall be given to which part of image.
- Joining splitted color image
  Now the newly obtain image i.e color image will be in different segments so we need to join all parts properly to get complete colorful image.
- Applying image difference method
  Analysis of work can be suggested to be done by image difference method in which we are.

2. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.132.6364&rep=rep1&type=pdf
    a. Image is converted to lxy space
        i. lxy space was developed to minimize correlation between the coordinate of the color space. The color space provides the decorrelation, corresponding to an achromatic luminance, l and two chromatic values x and y, which roughly correspond to yellow-blue and red-green opponent channels. Thus, changes made in one color should minimally affect values in the other. This allows us to selectively transfer the chromatic x and y values from the color image to the gray-scale image without cross-channel artifacts

b. Next, we go through each pixel in the gray-scale image in scan-line order and select the best matching sample in the color image using neighborhood statistics.
c. The best match is determined by using a weighted average of pixel luminance and the neighborhood statistics.
d. The chromaticity values (x,y) of the best matching pixel are then transferred to the gray-scale image to form the final image.
e. SOME OTHER STEPS:
    i. In order to account for whole differences in luminance between the two images we perform luminance remapping [5] to linearly shift and scale the luminance histogram of the source image to fit the histogram of the target image. This helps create a better correspondence in the luminance range between the two images but does not alter the luminance values of the target image.
    ii. for each pixel in the gray-scale image in scan-line order the best matching color sample is selected based on the weighted average of luminance (50%) and standard deviation (50%)
    iii. Once the best matching pixel is found, the x and y chromaticity values are transferred to the target pixel while the original luminance value is retained.
    iv. In order to allow more user interaction in the color transfer procedure and to improve results, sample blocks are used between corresponding regions in the two images.

3. Creating a Color Map to be used to Convert a Gray Image to Color Image - Jamil et al



The proposed method can be implemented applying the following steps:

1. Get the gray image and the colormap.
2. If the colormap is empty then apply the next steps else exit
3. Create an empty 256x3 lockup matrix(colormap).
4. Referring to the gray image and studying it's features define the gray level ranges, and for each range define the corresponding color by setting R, G, And B to a certain color.
5. From step 4 create a matrix with the R, G, and B values.
6. From step 4 create a rang vector.
7. Use interpolation of the matrix values and rang vector to create a color color- map .
8. Use the obtained colormap with the gray image to get the color one.

After the research phase, we then started to build our own model.

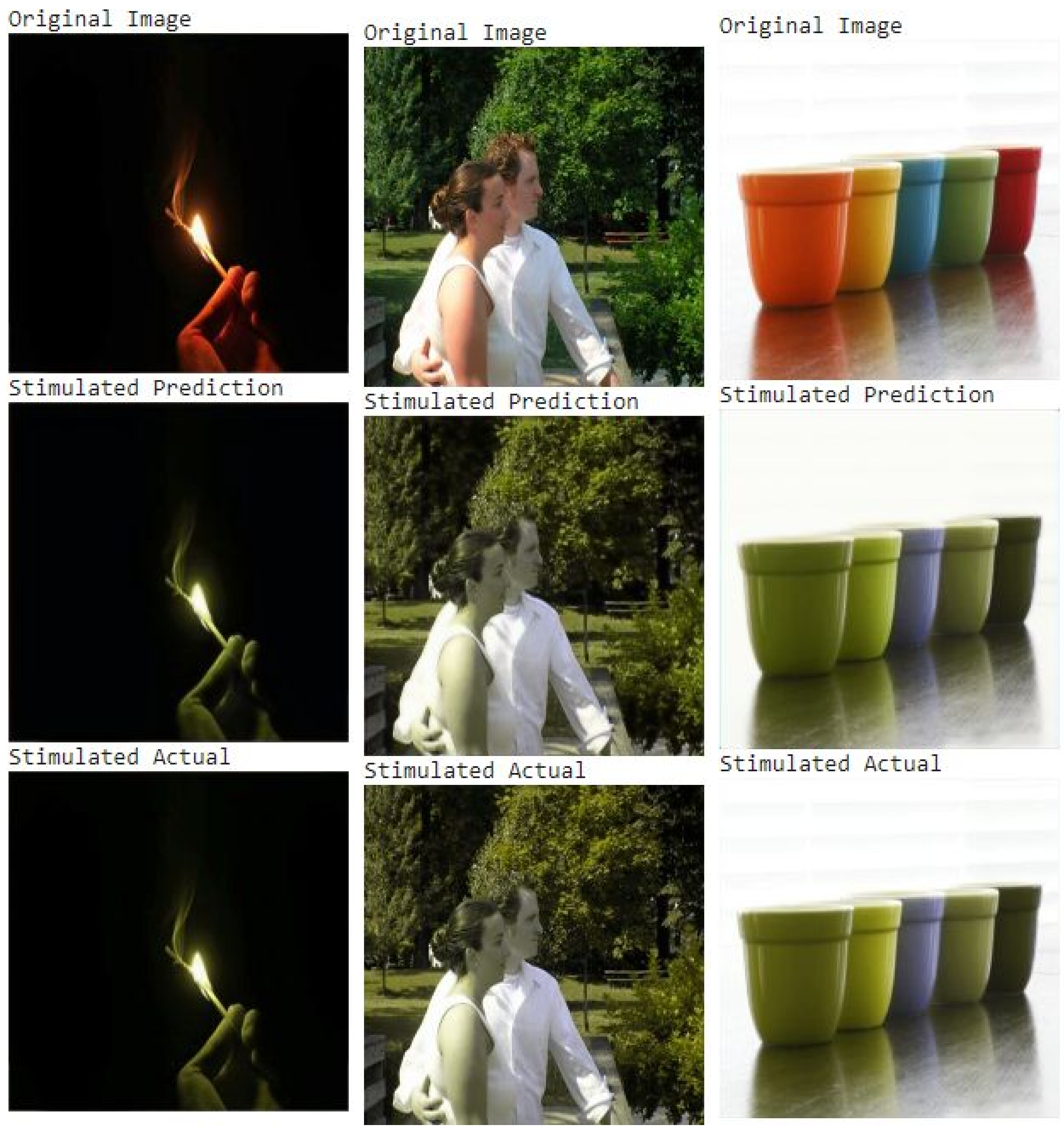
My work:

1. **Autoencoder approach:**

    Train an auto encoder - cascaded, using pairs of input and output of stimulator. Train 2 auto encoders, transform real world image into cvd stimulation, and then back to the real input image.

    For the simulation, I wrote a python script that takes an image and the type of CVD deficiency as input and outputs how a CVD user would see the image. The function was built by referencing the code file (from the paper) attached above.

Results:



For the model to create back the original image, from the simulated image, I used the difference parameter(difference of the stimulated image form the input image). Thus the input to the model was the stimulated image and the difference between original and simulated image and the output was the original image.
Analysis: For analysing the workings of the model, the activation layers of the model were visualised(plotted) and analysed

For protanopia(i.e. Red color deficiency)
How the model is working on different colors:

1. Blue: The model does not change the blue color much. Reduces the intensity by around 10%
2. Green: The model adds a little green to the blue region(around 5%) and a major chunk to the red region(15-20%). It leaves the green regions intact.
3. Red: The model dramatically scales down the red intensity values in the red regions and adds red to the green region.
4. Thus the model does not change the blue regions. It basically tries to mix red and green as much as possible. It adds green to red regions, while keeping the green values in green regions intact. And adds red values to green regions, while scaling down red regions red intensity values.

2. Cluster Based approach:

Define mappings on the whole dataset, that define how much a color differs from its simulated value. Basically I captured this essence, by defining a parent array, that had all the values that a given color was transformed to by my auto encoder model, using the whole dataset. Then using this parent array I defined 2 child arrays:

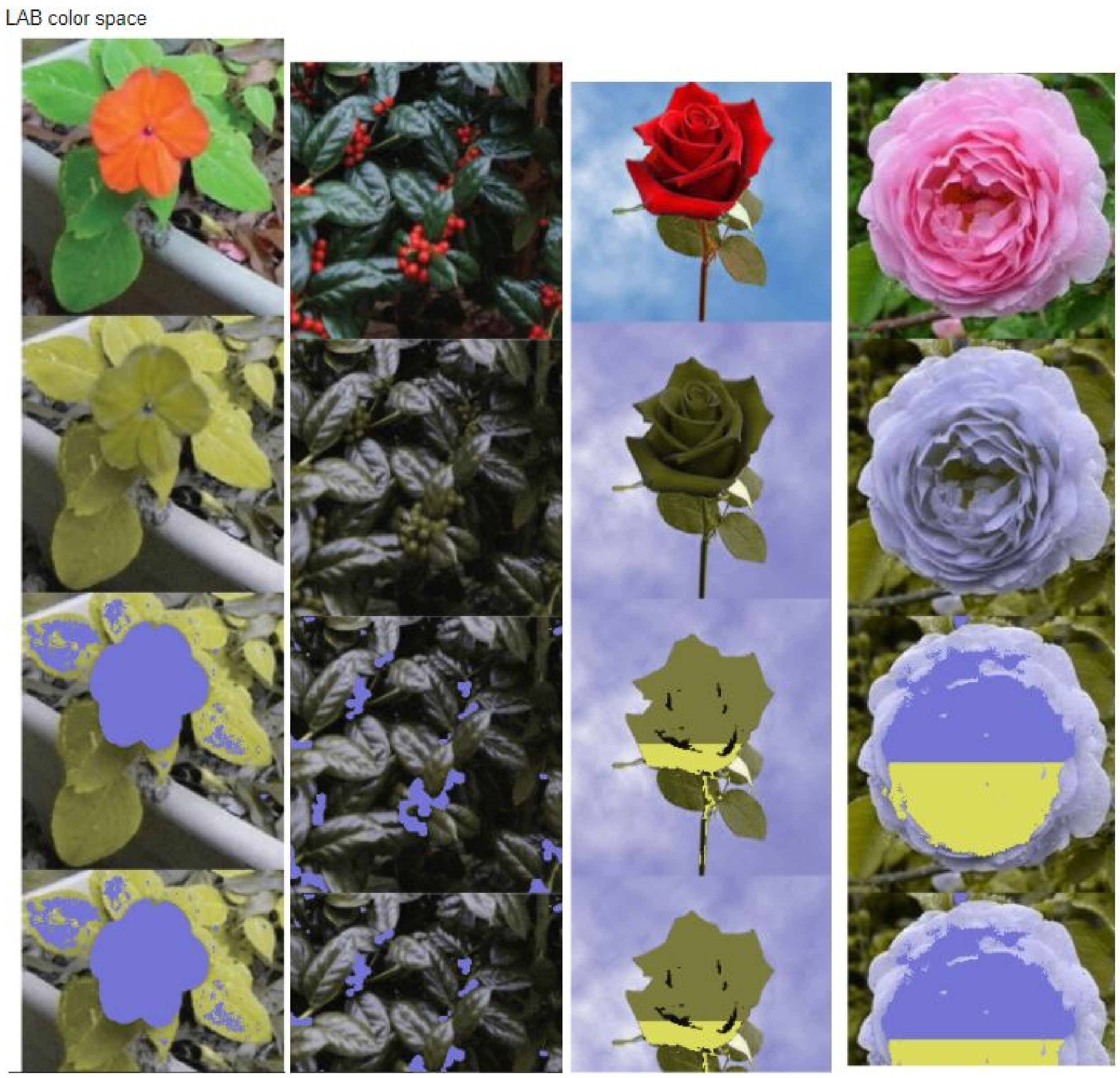
1. one for the mean i.e. the mean of the color Ci’s that a color C1 is being transformed to, and
2. the variance between the color C1 and the transformed color Ci (used to check the spread of a color)

Using these 2 arrays, we now know which colors transform the most, and which deviate the least from their simulated value.

Approach 2.1:

1. Apply kmeans on map
2. Define 2 clusters
3. From original image, change colors that belong to cluster 2 => cluster 1
4. The transformation should be such that the new color should be distinct from colors already present in cluster 1
5. Another condition should be that the final image shouldnt change much from the original image.

Results for 2.1:

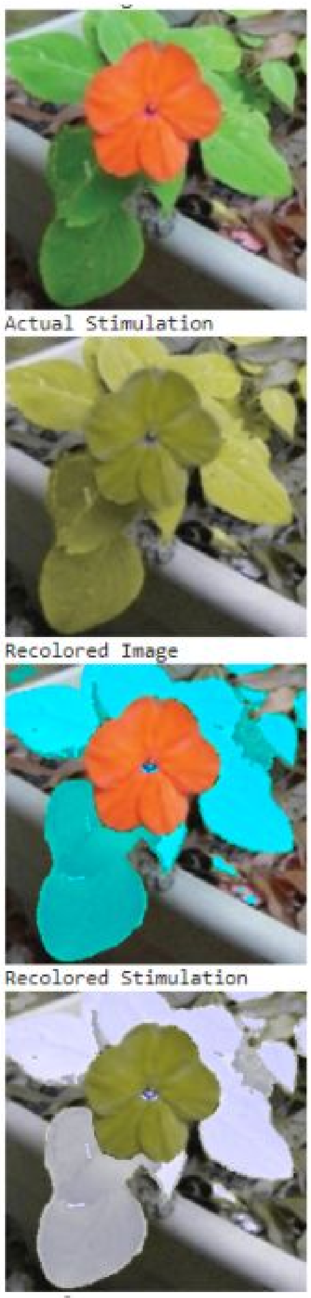


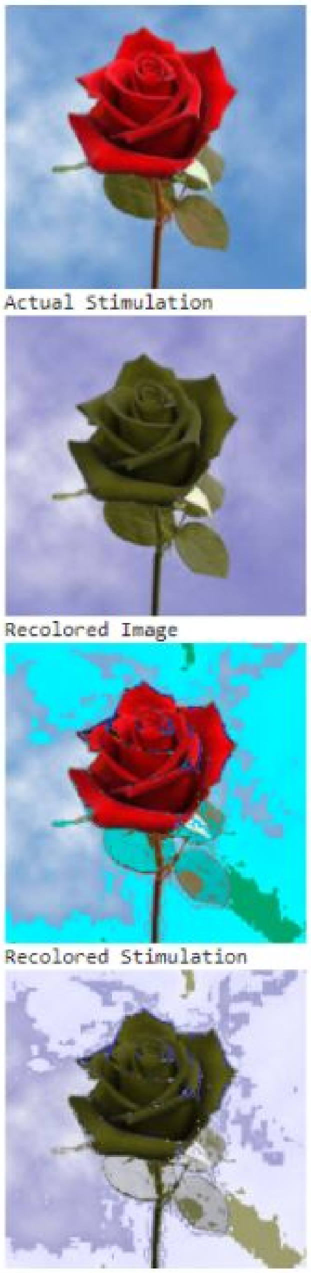
Approach 2.2:

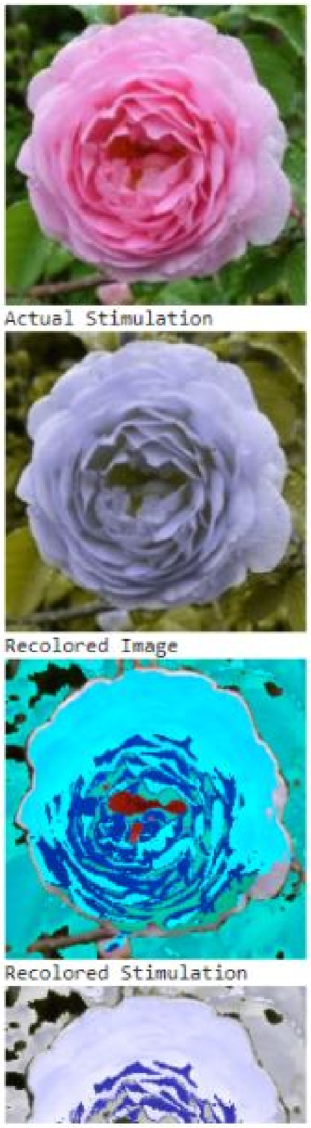
1. Using the given map, define different color groups. For example, let's say we define 500 color groups for the 255X255X255 BGR color space.
2. Now our aim is to change the color of the region that belonged to different color groups in the original image, but the same color group in the stimulated image.
3. First method:
    a. We could define some threshold distance, and then check
    b. So let's say for pixel x, y we could check till x+ 10 and y + 10. IF original image and stimulated color groups match
    c. Problems:
        i. Not efficient
        ii. Recoloring could be problematic because if we recolor one, then we need to consider its neighbouring pixels again and so on
4. Second method:
    a. We could use the difference from the original and the stimulated image. Wherever the difference is high(some threshold) we know that these areas can cause problems.

5. Now we have the areas, we can simply change the color group of the area, and assign it some color group which is as far as possible from the original/neighbouring color groups.
    a. So what we can do is, define clusters on the entire image size(255^3) or just define clusters on the current image. Now take the difference of the input image and the stimulated image. Wherever the difference is above a certain threshold, we can look into those pixel values, let's say they belong to the cluster 5. Now after going through the entire image, we now know which clusters are causing problems, we thus map these clusters to entirely different clusters.
    b. Which Can be done by either taking a sense of which cluster is the furthest from the current cluster. Now we go through the image again and change these colors.

Results for 2.2



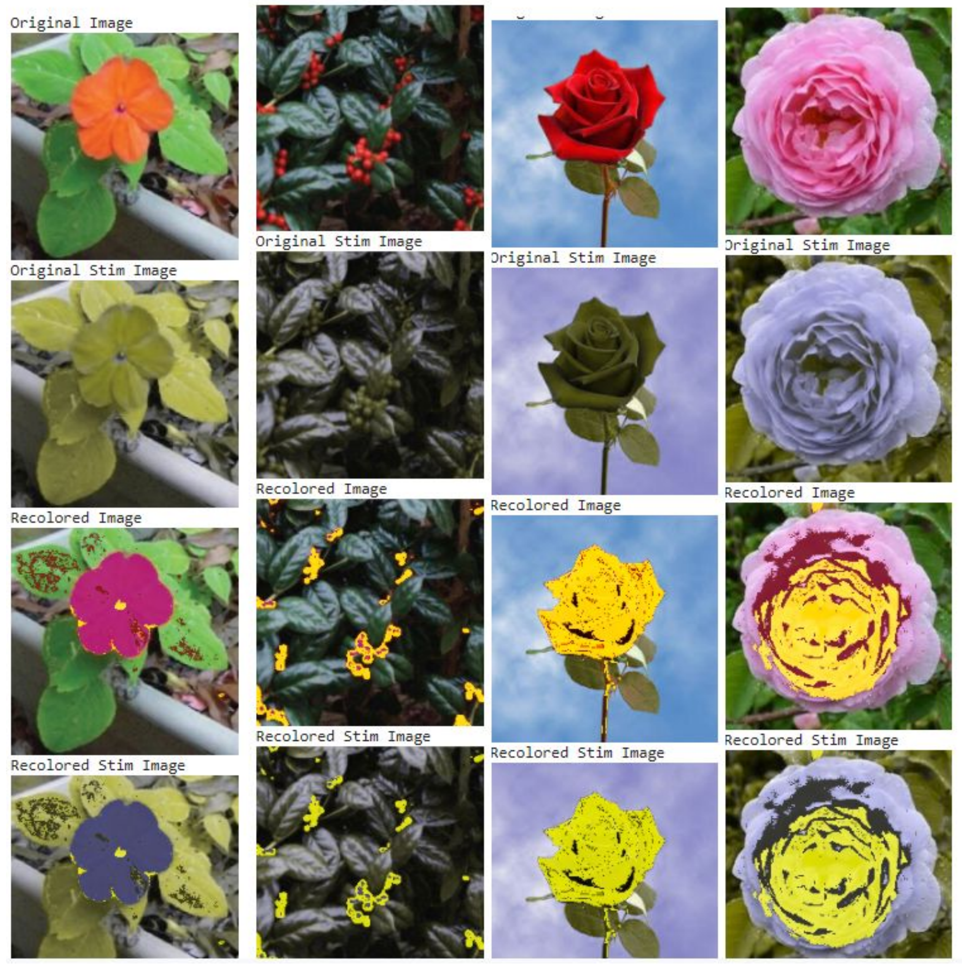




Approach 2.3:

An amalgam for the 2.1 and 2.2 approaches discussed above

Results for 2.3



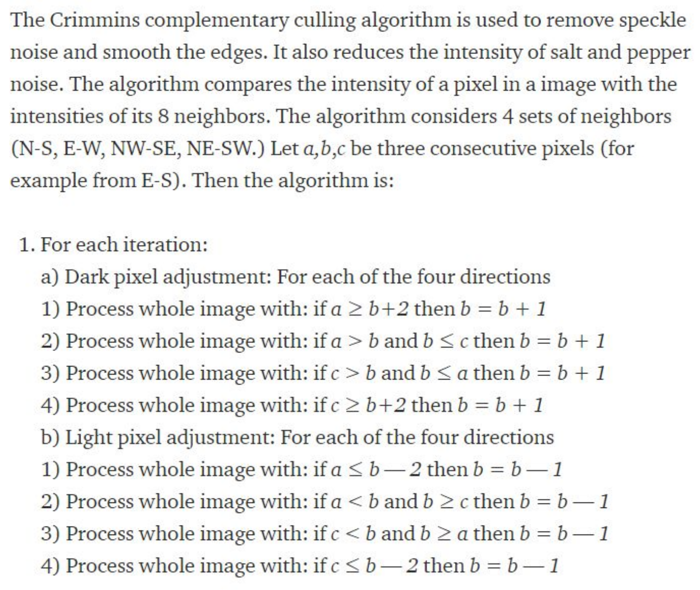
Still as can be seen from the results, there is noise present in the image.

Approach 2.4

I worked on Denoising the output image from approach 2.3:

Denoising approaches covered:

1. Rudin, Osher and Fatemi algorithm.
2. Crimmins algorithm:



The Crimmins complementary culling algorithm is used to remove speckle noise and smooth the edges. It also reduces the intensity of salt and pepper noise. The algorithm compares the intensity of a pixel in a image with the intensities of its 8 neighbors. The algorithm considers 4 sets of neighbors (N-S, E-W, NW-SE, NE-SW.) Let a, b, c be three consecutive pixels (for example from E-S). Then the algorithm is:

```
1. For each iteration:
    a) Dark pixel adjustment: For each of the four directions
    1) Process whole image with: if $a \geq b+2$ then $b = b + 1$
    2) Process whole image with: if $a > b$ and $b \leq c$ then $b = b + 1$
    3) Process whole image with: if $c > b$ and $b \leq a$ then $b = b + 1$
    4) Process whole image with: if $c \geq b+2$ then $b = b + 1$
    b) Light pixel adjustment: For each of the four directions
    1) Process whole image with: if $a \leq b — 2$ then $b = b — 1$
    2) Process whole image with: if $a < b$ and $b \geq c$ then $b = b — 1$
    3) Process whole image with: if $c < b$ and $b \geq a$ then $b = b — 1$
    4) Process whole image with: if $c \leq b — 2$ then $b = b — 1$
```

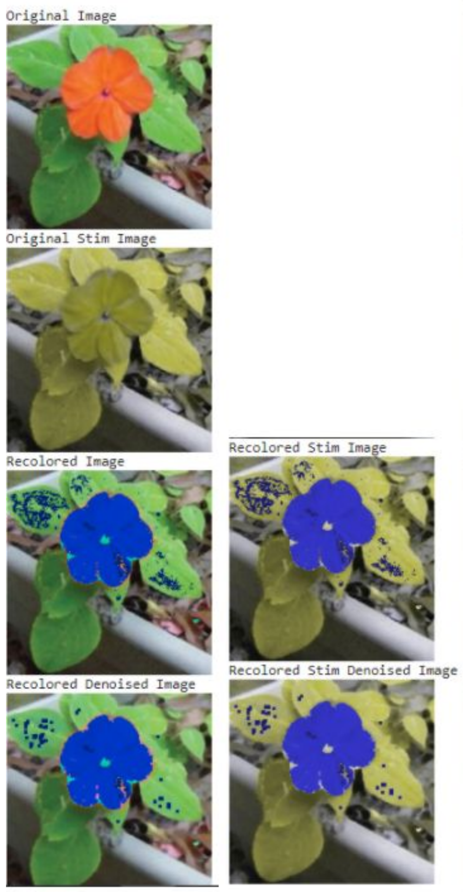
a.

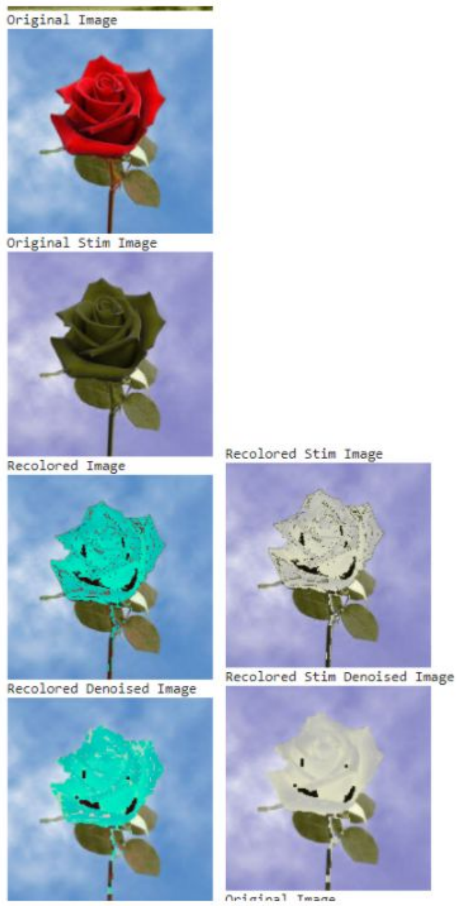
3. Conservative smoothing, single channel:
    a. The conservative filter is used to remove salt and pepper noise. Determines the minimum intensity and maximum intensity within a neighborhood of a pixel. If the intensity of the center pixel is greater than the maximum value it is replaced by the maximum value. If it is less than the minimum value then it is replaced by the minimum value. The conservative filter preserves edges but does not remove speckle noise.
4. img = cv2.fastNlMeansDenoisingColored(img, None, 10, 10, 11, 21)
5. img = cv2.dilate(img,kernel,iterations = 1), cv2.erode(img,kernel,iterations = 1)
    a. Morphological technique used to add/remove pixels from the boundary of the image.
6. img = cv2.medianBlur(img, kernel_Size)
    a. Median filter, used to blur the image and remove salt and pepper noise.
7. img = cv2.bilateralFilter(img,9,75,75):
    a. A non linear filter, bilateral filter is used for smoothening images and reducing noise, while preserving edges
8. Upsharping**:** img = np.array(img.filter(ImageFilter.UnsharpMask(radius=2, percent=150)))
    a. The Unsharp filter can be used to enhance the edges of an image
9. Custom Technique:
    a. Define clusters in the current image.
    b. Change the color of the current pixel based on the colors of the neighbouring pixels.
    c. If they all belong to a certain cluster, and the current pixel is not in that cluster. Treat this as noisem and give it the same color as others.

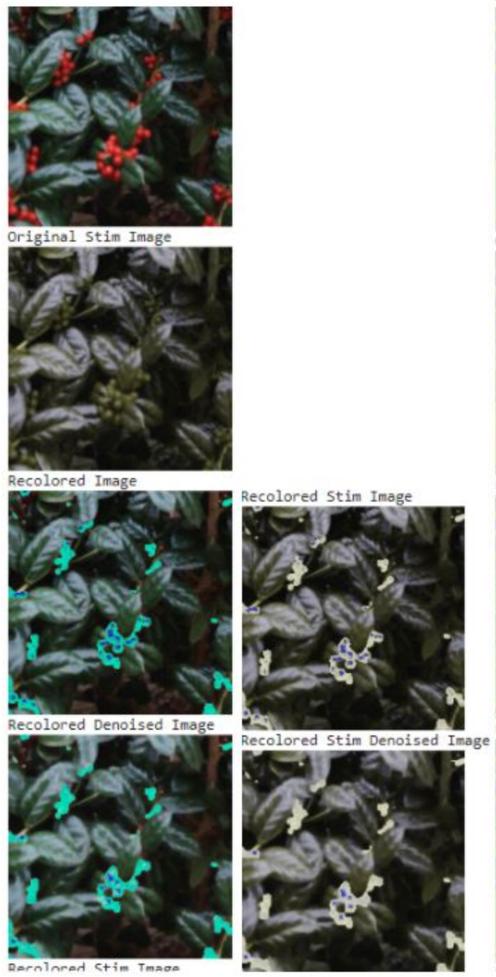
Denoising:

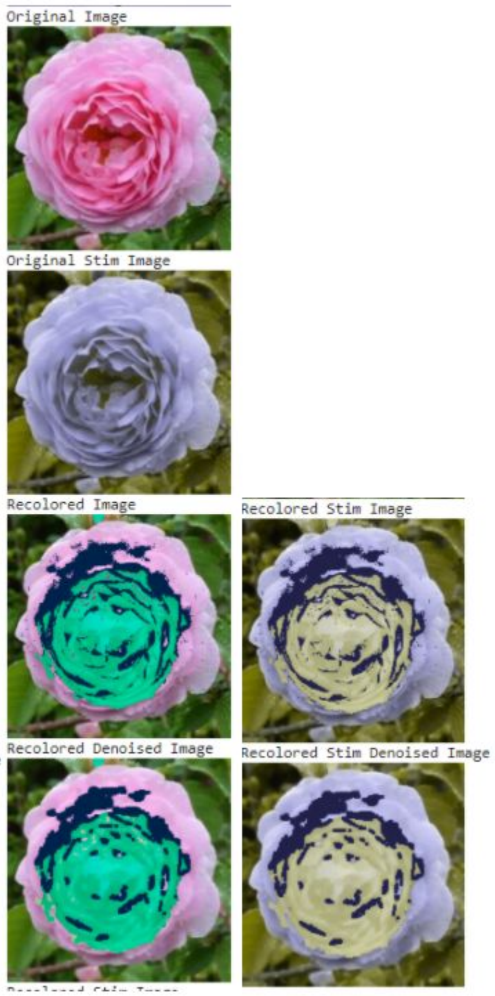
1. https://github.com/m4nv1r/medium_articles/blob/master/Image_Filters_in_Python.ipynb
2. https://towardsdatascience.com/image-filters-in-python-26ee938e57d2

Results for 2.4

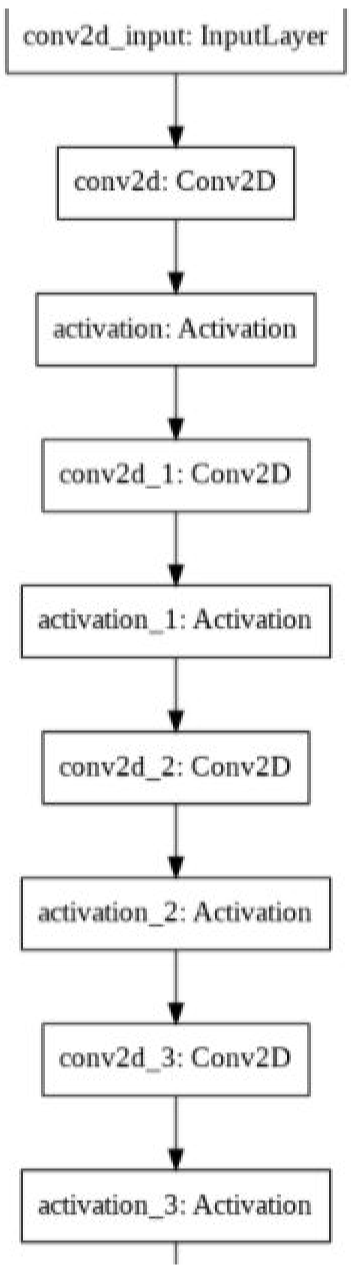








Approach 3: Custom neural nets



This is a screenshot of the model. For the full model picture, refer to the Colab notebook.

Approach 3.1

ModelB = modelA.layers(freezed weights) -> modelA.layers.

Input image = x

Stim_Image = x_stim

Output Image = y

The first model outputs the CVD stimulation of the input image(x_Stim) and has been previously trained.

Loss = x_stim - y(the output image should not vary from the stimulated version of input image) + y - x(output image should not vary much from the input image) *This approach is wrong because we don't need the first loss term. It has no significance.*

Loss2 = x_stim - y *wrong approach again*

Loss3 = y - x *This is not complete because we also need to ensure that the stim of final output doesn't differ from it much*.

Approach 3.2:

Add the trainable model first. Attach to it the stimulation model.

ModelB = modelA.layers -> modelA.layers(freezed weights).

Input image = x

MidModel = y

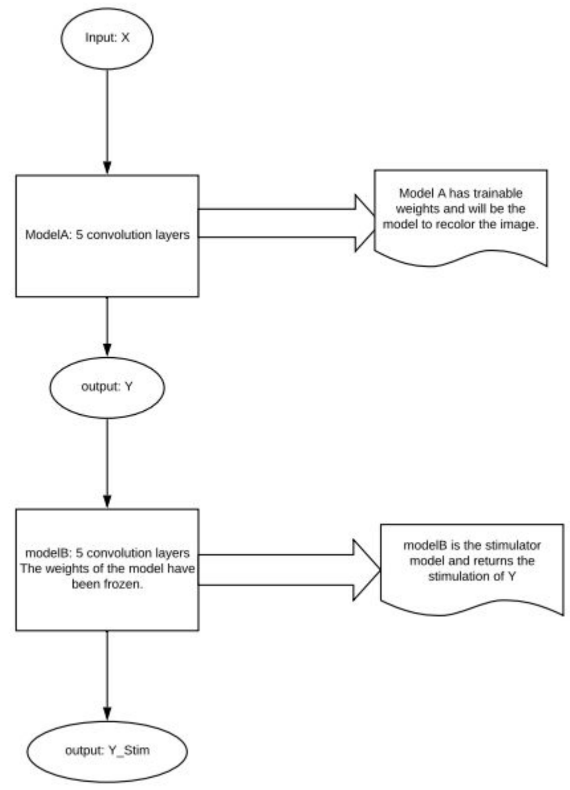
Output Image = y_stim

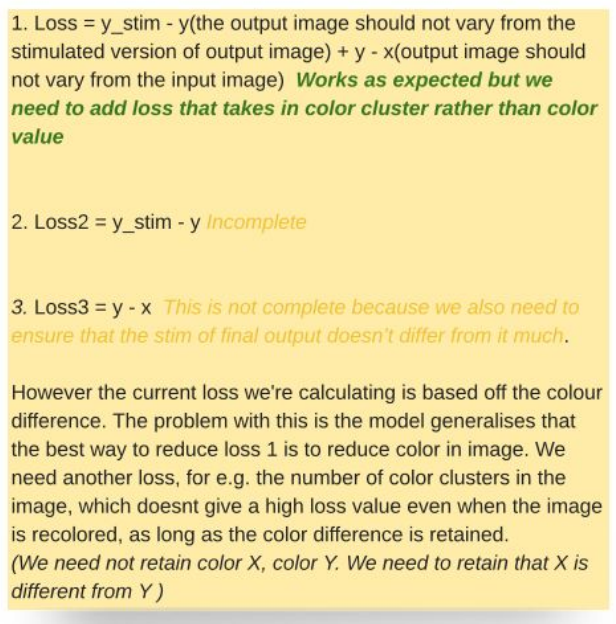
The first model outputs the CVD stimulation of the input image(x_Stim) and has been previously trained.

Loss = y_stim - y(the output image should not vary from the stimulated version of output image) + y - x(output image should not vary from the input image) ***Works as expected but we need to add loss that takes in color cluster rather than color value***

Loss2 = y_stim - y *Incomplete*

Loss3 = y - x *This is not complete because we also need to ensure that the stim of final output doesn't differ from it much*.


Input: X
ModelA: 5 convolution layers
Model A has trainable weights and will be the model to recolor the image.
output: Y
modelB: 5 convolution layers The weights of the model have been frozen.
modelB is the stimulator model and returns the stimulation of Y
output: Y_Stim

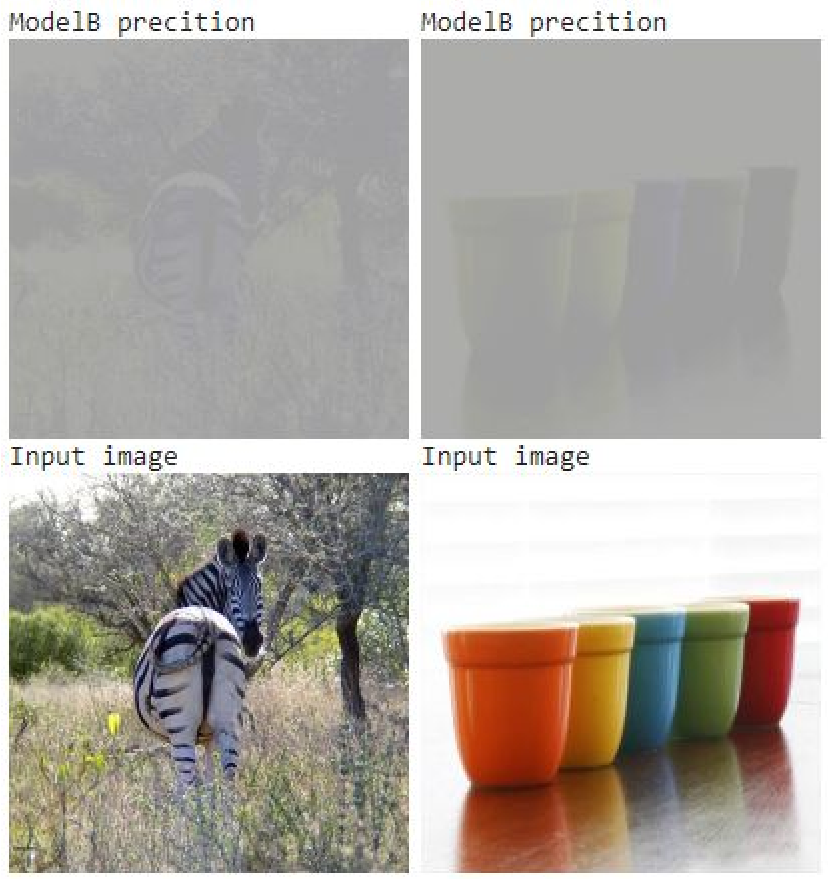

1. Loss = y_stim - y(the output image should not vary from the stimulated version of output image) + y - x(output image should not vary from the input image) ***Works as expected but we need to add loss that takes in color cluster rather than color value***

2. Loss2 = y_stim - y *Incomplete*

3. Loss3 = y - x *This is not complete because we also need to ensure that the stim of final output doesn't differ from it much*.

However the current loss we're calculating is based off the colour difference. The problem with this is the model generalises that the best way to reduce loss 1 is to reduce color in image. We need another loss, for e.g. the number of color clusters in the image, which doesnt give a high loss value even when the image is recolored, as long as the color difference is retained.
(We need not retain color X, color Y. We need to retain that X is different from Y)

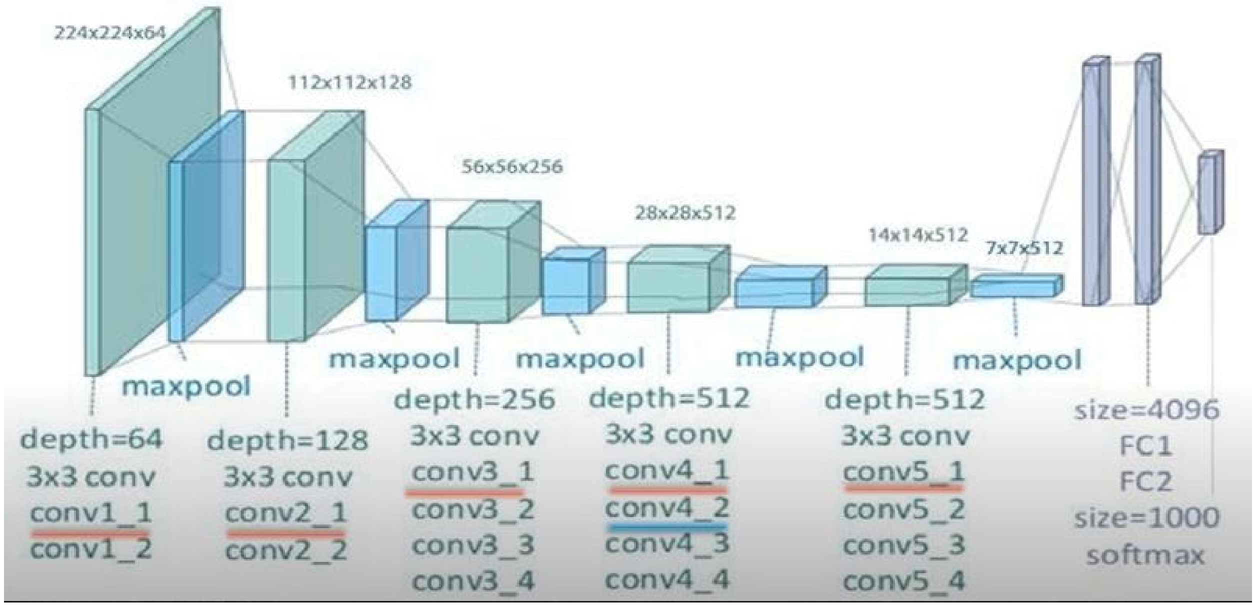
Results:

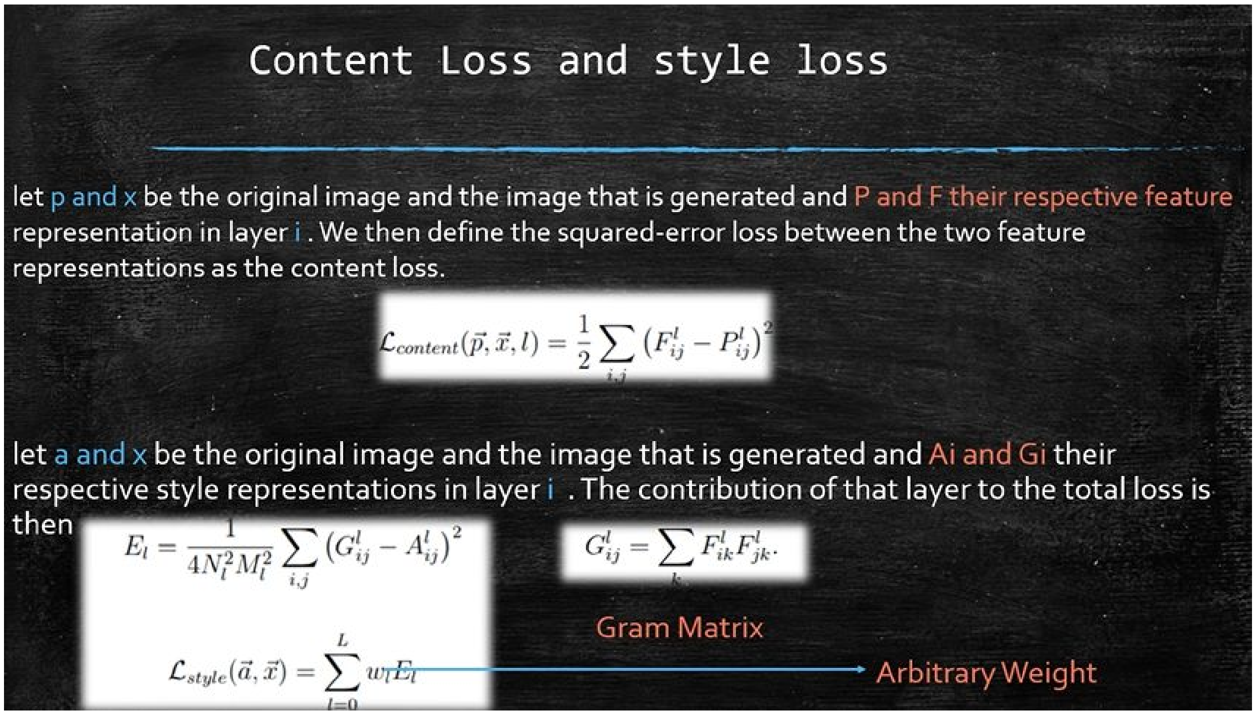


Approach 4: Neural Styling

Neural styling: Using a set of images that have colors that do not vary much from their simulated value, as in the neural style transformation, we can transform an image with high degree of color divergence, into an image that has colors that vary less.
https://github.com/anishathalye/neural-style

1. To capture the essence of style and color in the image, we use correlations between filter responses over the extent of the feature maps. We do not use a single filter response because that will only give us localized observations, which could contain info about the shapes, edges etc. But texture and color are global features of the image. Thus they need to be studied by analysing the correlation betweens these filter responses, as texture and style variables will be present in all these responses.
2. These correlations between feature maps is known as gram matrix
3. • The layers used for calculation of gram matrix are 1_1 , 2_1, 3_1, 4_1, 5_1 with varied style weight constant for each layer.
4. Style uses lower layers more. Thus weights will be higher for 1_1 than 4_1 or 5_1.
5. A pre-trained model, say VGG 19 is used. Then its certain layers, and the filter responses are extracted(as shown in the figure below). These responses tell us about the style and content(color texture etc of the image).

6. 

7. 

Content Loss and style loss

let p and x be the original image and the image that is generated and P and F their respective feature representation in layer i . We then define the squared-error loss between the two feature representations as the content loss.

$$\mathcal{L}_{content}(\vec{p},\vec{x},l)=\frac{1}{2}\sum_{i,j}(F^l_{ij}-P^l_{ij})^2$$

let a and x be the original image and the image that is generated and Ai and Gi their respective style representations in layer i . The contribution of that layer to the total loss is then

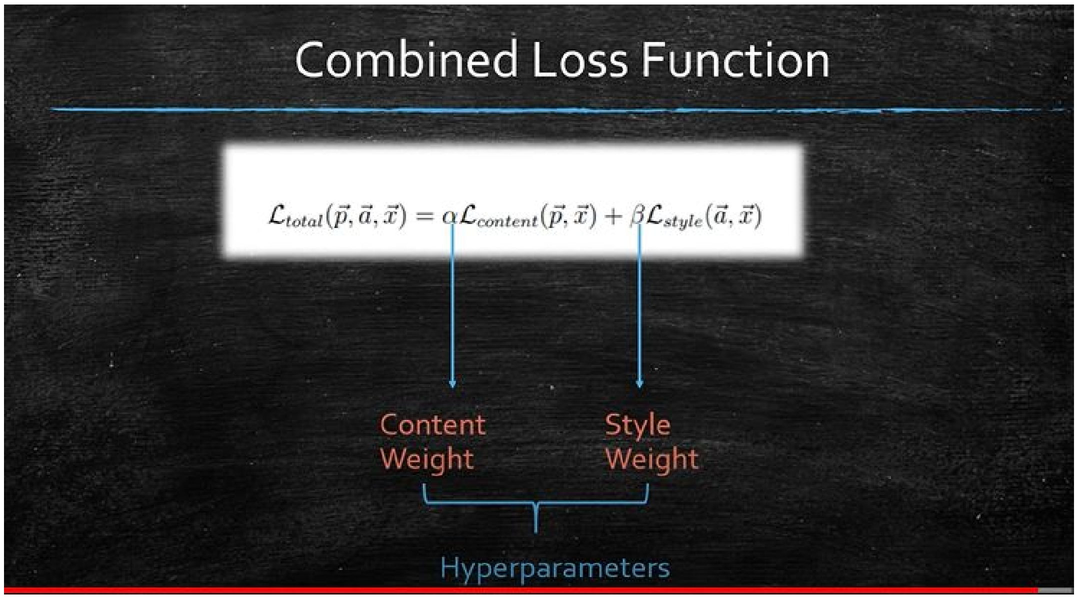
$$E_l=\frac{1}{4N_l^2M_l^2}\sum_{i,j}(G^l_{ij}-A^l_{ij})^2$$

$$G^l_{ij}=\sum_k F^l_{ik}F^l_{jk}.$$

Gram Matrix

$$\mathcal{L}_{style}(\vec{a},\vec{x})=\sum_{l=0}^{L}w_l E_l$$

Arbitrary Weight

8. 

Combined Loss Function

$$\mathcal{L}_{total}(\vec{p},\vec{a},\vec{x})=\alpha\mathcal{L}_{content}(\vec{p},\vec{x})+\beta\mathcal{L}_{style}(\vec{a},\vec{x})$$

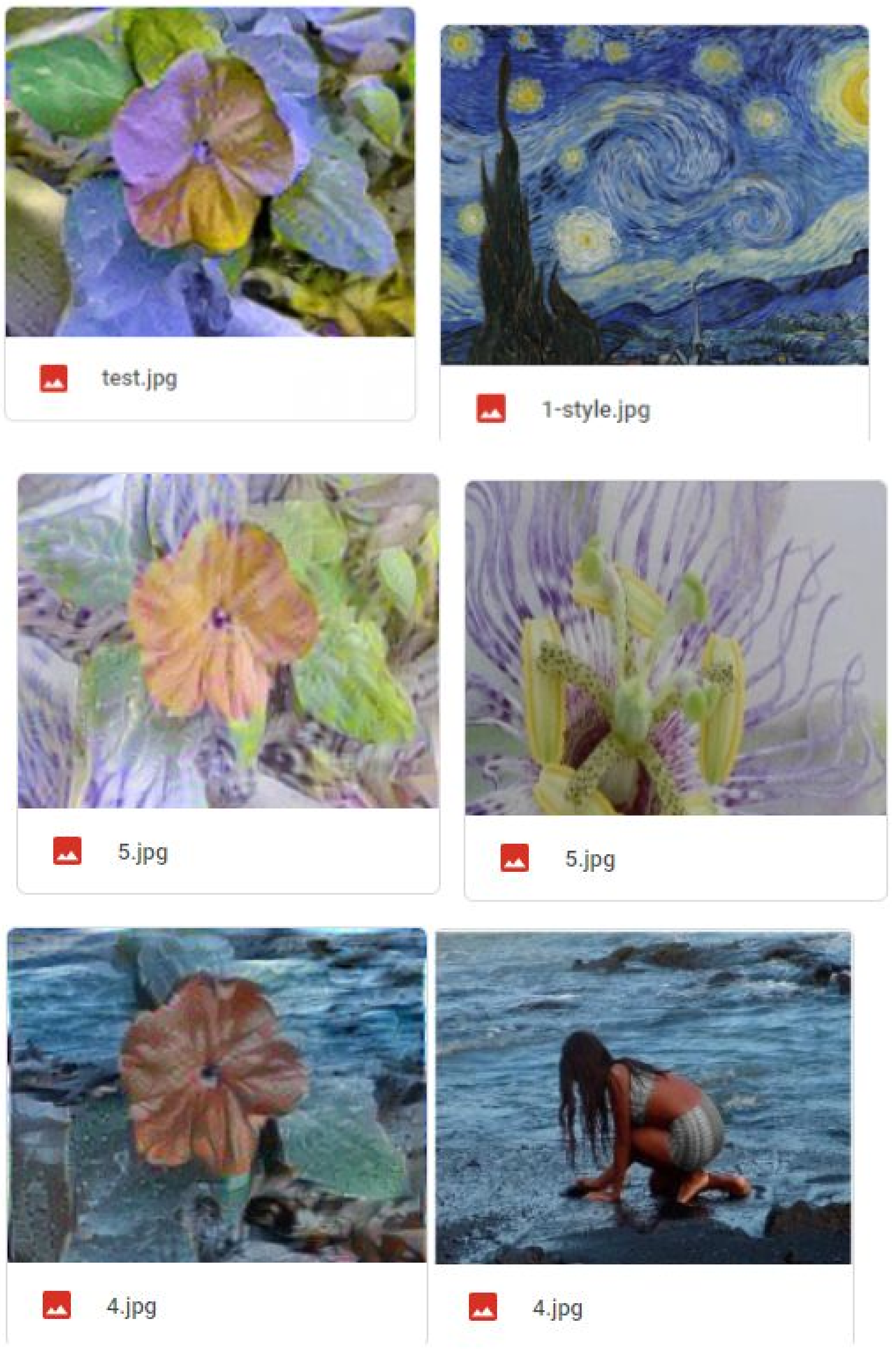
Content Weight

Style Weight

Hyperparameters

Results:

Here are some screenshots of the recolored images, with their style image to the right:



Analysis:

300 best images were selected from the dataset (for the simulation styling), based on the mapping that was built and saved earlier(in approach 2). Simply put, I've chosen the top 300 images that change the minimum during CVD simulation.

The results however are not consistent with what we want to achieve. I believe the reason behind this is the inherent structure of Neural styling, which does not recolor the image based off of some localized information. The recoloring is done on a global scale, thus the whole image gets recolored, leading to loss of information.