Colorization Via Reference Image
Final Project Report

Killian Susini
killian.susini@student-cs.fr

March 6, 2022

Abstract

In this paper we focus on the task of image colorization, and discuss the different challenges that are encountered when one tries to solve the problem, such as the ill-posedness of the problem in the general case. Then we go over an overview of the traditional techniques used to solve the problem, and finally we go over the techniques and algorithm that will be used to try and complete the task. We compute discriminative features for each pixels, such as the standard deviation, but also the response level of Gabor kernels as well as SIFT descriptors [5], and then we segment the target grayscale image. In addition, we do the same for a user provided reference image that is similar to the target. Between those segmentation, we find the one that correspond to one another the most, where the segments closeness is define using the average of the features of the pixels that compose the segment. Then we do the same at the pixel level within the selected pairs of segmentation, finding the closest matching pixel from the reference to the target. Finally, we transfer the color of the best matching pixels from the reference image onto the matching pixels of the target image, and apply an optimization method [2] to obtain a fully colorized image. We finally go over the results obtain by the method.

Contents

1. Introduction

Image colorization is the task of adding color to a grayscale image. We expect images today to be in color, since any phone cameras today are able to record them faithfully. However, there are wealth of old photos and movies that were obtained when black-and-white (grayscale) cameras were the norm. It is therefore of prime interest to find techniques to color those images. Also, one could gain Today deep learning methods are particularly successful at this task, especially in terms of speed and being human involvement free. Nevertheless they still sometimes fail to find the right natural color for some class of items. Fruits and vegetables are one example, as demonstrated using the Natural Colour Dataset (NCD)[6]. The main problem of colorization is to be able to find the color of an image with known intensity, but that already raise some problems. First is the fact that the problem is under constrained in the sense that even with the intensity the number of potential colors is high. Second, even when the object in an image is known (e.g. an apple), there can be difficulties in assigning a convincing color to the whole object. This is probably due to the fact that the same object can have many potential colors (red or green apple) while it is necessary that the entirety of the object be colored consistently [6]. Therefore achieving consistency is important, in addition to finding a natural color for an object. Traditional deep learning algorithm can achieve impressive results, but sometimes fail to keep a consistent color throughout some objects, making the colorization obvious. A solution to the second problem is to constrained similar neighbor to have similar colors [2]. For many objects, given the right color, this produce a convincing colorization. Then to solve the first problem (finding the right color), a traditional solution is to ask for the help of a human, which can give an appropriate color for each part/object of the image. This is used in [2] to color images. However, we may still want to reduce human involvement to a minimum. For example, by one only requiring one reference image from which we can extract the right colors for each part of the objects. In this paper, we will explore one such method.

2. Problem Definition

Most images are stored using the three color channels Red, Green and Blue (RGB), and so we would like to estimate the three channels from the grayscale image. However any three channels can do the trick, and one of particular interest to us is the YUV channel encoding. This one only has two color (called chrominance) channels, U and V, and one intensity (or luminance) channel Y, which is the grayscale channel. That is, we only need to estimate two channels from our image[6]. We use I_{rgb} to denote the R, G and B channels of the image I. To obtain the grayscale image I_y from our RGB image, we typically don't use a simple averaging over the channels, because of the human perception of red green and blue. Instead, we use

$$I_y = 0.2989 * I_r + 0.5870 * I_g + 0.1140 * I_b$$

which match the relative importance of each color to our eyes.

So our problem is, from I_y, produce $I'_u \approx I_u$ and $I'_v \approx I_v$. As explained in [paper], using traditional image reconstruction metrics is not entirely appropriate, and a relatively appropriate metric will be to use a loss over the U and V channels of the YUV color space. Here, we could try to minimize the mean squared error of those two channels produced by our method and the original image U and V channels. That is,

$$\texttt{min} \sum_{i,j} (I'^{ij}_u - I^{ij}_u)^2 + (I'^{ij}_v - I^{ij}_v)^2$$

Where I^{ij} is the pixel at position (i, j). However, I did not find the results particularly meaningful and so I will stick to provide a qualitative assessment in the evaluation part.

Immediately it is easy to understand that we cannot guess I_u and I_v from I_y without additional information. However, it is not necessarily due to the lack of knowledge about what object is in the image. For example, while if we have the grayscale image of a grown banana over a white background (no obstruction), it is indeed true that we could imagine an algorithm that detect the banana and guess that the appropriate color is yellow, certain object cannot be assigned such a natural color. The worst example would be a t-shirt; indeed, a t-shirt would be fine with about any color given a grayscale intensity, be it red, green or even letting it gray. In fact it could also be argued that it could be colored with multiple color each at the same grayscale intensity, and we could never properly guess the color. Conversely, one could say that any sensible t-shirt coloration given as output would be proper, which makes the problem ill-posed (there is no *right* color, only apropriate ones). For this paper, we will limit ourselves to objects that have such natural colors, namely fruits and vegetables. Remark that there is still some problem in defining what is the right color to assign to certain fruits (e.g. apples, pears, ...), however others are much more straightforward (e.g. oranges).

Still, the colorization problem restricted to certain class of object is still affected by the same challenges as many in the field of image enhancement, namely changes in illumination, differing viewpoints, and occlusion of part of the image [6]. The NCD dataset [6] that we will use is composed of fruits and vegetable over a white background with only sometimes slight difference in illumination, making the problem much easier.

The ultimate goal of image colorization is to please human vision sensibilities. The metric described before sometimes fails to remark glaring defect of the result, so for image colorization a qualitative assessment is almost always made, so as to show some results where the method produce good results and some results were it fails to do so.

3. Related Works

There have been many works in image colorization over the years. The most recent methods are the most human-interaction free, and they traditionally use deep learning architectures ranging from CNNs to GANs[6]. Some models use additional text information to color the image[6]. Some others are user guided, where a user interactively provide help to the model to progressively color the image[6].

Beyond the new deep learning models, traditional methods can be divided into two categories, interactive and example-based methods [7]. For example, Levin et al. [2] introduced an interactive method that works by asking a user to provide color scribbles over an image and propagate those scribble over the image. Then if the result is not yet satisfying, the user can go back and add new scribble to improve the result.

Next are example based methods, which is what we will do. In addition to the grayscale input, it is useful to add realistic color image similar to the one we are trying to colorize. Indeed, there have been many papers that explored this path. To obtain such images. For example (use Image color. using similar image paper to go over the techniques quickly)... This paper won't deal with the automatic selection of a reference image, whether by text or by other means, although an exploration will be discussed in the conclusion.

In [1], the user provide a reference image that is similar to the target image to colorize. Then, both are segmented into superpixels. Then, superpixels in the target image and the reference image are matched using cascading feature matching, where the features are computed from the respective images in a similar way as it will be done here, describe later. Finally, the image is colored using a voting framework to enforce spatial coherence (superpixels with the same luminance should be colored the same way). Then it colors the center pixel of the superpixel with the superpixel average color (the average U and V chroma), and finally use [2] to propagate the artificial pixel "scribble" throughout the image. In this project, we will do the same thing.

In [7], similar to [1], the user provide a reference image to help the colorization. Also like in [1], a feature vector is computed for each pixels in both images. This time, each pixel in the target image is assigned to a pixel in the reference image. To do so, the image is first segmented not by superpixels as in [1], but using a meanshift segmentation [3]. Then the features of each pixels in a segment are averaged to produce a feature vector for the segment, which is then match against a segment in the original image. Then each pixel can be matched only against the pixels in the matched segment. Finally, the color (UV) of the best matched pixels in each segment are applied to the target image. Finally, The optimisation technique used in the propagation of those user provided color scribble in [2] is used to propagate the colors thoughout the image. In this paper, we will replicate very closely this work. The main differences between this project and the paper are the feature vector computed. In [7], 5 features are computed for each pixel; luminance, standard deviation, Gabor features (40-dims), SURF descriptors (128 dims) , and a high-level saliency map. In this project, the three first feature are replicated. Instead of the SURF descriptor, I extract SIFT descriptors from each pixels. The reason is that I could not find any implementation of SURF [4] or SIFT [5] that could give me a descriptor for every point in an image, as they usually first find keypoints in an image and the compute the descriptors. Therefore, I found a tutorial that explained how to compute the descriptors for SIFT and implemented it. Also, I could not find an implementation for the high level saliency map used in the paper, and the algorithm used to compute it required more than a little more work to get (it requires to train a Conditional Random Field on a dataset of salient objects). I tried to replace it with a simple saliency algorithm implemented in the opencv library, but it did not gave good result, so I just went without it.

Regardless, I use the same weight as in [7] to compute the distance between the pixels/segments in the image, ignoring the saliency map. In the paper, the saliency map improved the result a bit, but overall the result is still good enough in the scope of this project.

4. Methodology

As a zeroth step, we need to find an image that will be used as a reference to color our target grayscale image. For us this is relatively easy, since we know the object in the image and therfore can obtain a suitable reference over the internet. For this particular project, we can use any of the other provided fruits/vegetables that are contained in the NCD dataset [6] to be used as a reference.

4.1 Segmentation

We first need to segment the image into meaning full parts. This helps speed up tremendously the pixel matching part. Indeed, If you had to try to find the best matching pairs of pixels for each pixel in the target image, it would be order $O(N^4)$, supposing images are square with side length N. In short, it would be at least quartic in the minimum side length of any of the two images, prohibitively expensive. Furthermore, nothing tell us that the best matching pixel would be a good match. A better way is therefore to first segment the image into spatially coherent chunks so has to match those together first, which would ensure both a quicker matching process and a better odd of finding a good match. More on this later.

For each image, target and reference, we get a segmentation by applying the meanshift segmentation algorithm presented in [meanshift]. Like in [7], we use a spatial and range distance of 8 so has to get a good number of segments (better for speed) while having those segment big enough to represent interesting part of the images. For the reference image, I had to choose between segmenting without or without the color; after some test, it seemed more sensible to segment without using the color information. That is, I first convert the reference to a grayscale image (using the equation presented earlier in the paper, implemented in Opencv), and then compute a segmentation. I did not program the meanshift segmentation myself, I found a library (pymeanshift) to do it.

4.2 Features Extraction

To match the pixels and segments, we need to represent the characteristics of each pixels. We want to represent as much of their local characteristics as possible, in order to match them with pixels in a similar representation (which should be therefore a good candidate to copy the color of). So here are the steps to produce the 170 dimensional feature vector for each pixels. Note that for all features (but Luminance), to computes them for pixel near the edge of the image requires points outside of it, and in this case I simply padded the image using the value at the edge.

4.2.1 Luminance

Here, we simply use the grayscale image of the target image (the input) and we obtain the luminance of the reference input using $I_y = 0.2989 * I_r + 0.5870 * I_g + 0.1140 * I_b$ as explained before. This gives us our first dimension.

4.2.2 Standard Deviation

our second feature dimension is obtained by computing the standard deviation of the pixel luminance with respect to its neighboring pixels. As in [7], we use a 5×5 window as neighbor. To speed up the computation, I compute the integral image of the luminance as well as the integrale of the square of the luminance. We can use the formula

$$(\sigma^{ij})^2 = (S_2^{ij} - (S_1^{ij})^2/n)/n$$

to compute the variance, where $S_1^{ij} = \sum_{i',j'} I_y^{i'j'}$ and $S_2^{ij} = \sum_{i',j'} (I_y^{i'j'})^2$ are the sum of the (square) luminance over the 5×5 area around i, j. Then the integral images allow us to avoid computing the 25 operation to obtain the area S_1 and S_2, and instead do 4 operations. As a remainder, the integral image of an image is define as such

$$B^{ij} = \sum_{i'=0}^{i} \sum_{j'=0}^{j} I_y^{i'j'}$$

similarly for the integral of the square, and then we can compute any sum over a rectangular area by adding and removing the sums at the corner of the area we desire, like so for our example

$$S_1^{ij} = B^{ab} - B^{cb} - B^{ad} + B^{cd}$$

with $a = i + 2$, $b = j + 2$, $c = i - 3$ and $d = j - 3$. Notice that (a, b) is inside the 5×5 area but the other points are outside (You can imagine it as adding the whole corner, removing two unused parts, and then adding what was removed twice).

4.2.3 Gabor Features

Here we use Gabor kernels to generate some features from the responses. In total, like in [7], we generate 40 kernels by using 8 different directions and 5 different exponential sizes (using $i = \{0, 1, 2, 3, 4\}, e^{i\pi}$). In short, the directions makes the filter respond to edges at different angles and the size makes the filter respond at different object size. I used Opencv to generate and apply the kernels. Here is an example of a Gabor kernel.

It is important to note that each Gabor filters here respond to a specific corner size and orientation. This makes this algorithm not invariant to size and orientation. It is not a problem as long as we are aware of it, and find an appropriate reference image.

4.2.4 SIFT Features

As explained, I compute a 128 SIFT (Scale Invariant Feature Transform) descriptor vector instead of the 128 SURF (Speeded-Up Robust Features) descriptor vector used in the paper I am following. The paper recommend using SURF for their greater discriminative power, however I only understood how to compute the SIFT Descriptors, and found no on-the-shelf python method to make either.

The SIFT (or similarly SURF, I won't mention it again) algorithm is used to find keypoints in an image and assign them with a descriptor vector. Then if you have two images, because those descriptor are both size and rotation invariant, you can match them between the images. This could ultimately be used to, for example, build an object classifier. Here, we only care about the descriptor, and we compute one for each pixel. However for those who are aware of the SIFT algorithm, the way the algorithm ensure size and orientation invariance is by being aware at which size and orientation the keypoint of interest was detected. I won't describe how it is done here since we do not compute any keypoint, but since we are not substracting the shape/orientation when computing the descriptor, it effectively makes the descriptor vector for each pixel **not** size or orientation invariant. This is in line with the Gabor filters above, and so we need to pay the same attention to this when finding a good reference image.

I describe how the 128 dimensional SIFT descriptor for each pixel is calculated. In a 16 by 16 window around the pixel, compute the direction and magnitude in between each pixel. That is, instead of computing the derivative of each pixel using the sobel operators, which are 3×3 kernels, you use two $2by2$ kernels ($[[1, 0], [0, -1]]$ and $[[0, 1], [-1, 0]]$) to compute the derivative in between each two by two pixels. Then, you can compute the direction and magnitude by computing $D = arctan(G_y/G_x)$ and $G = \sqrt{G_x^2 + G_y^2}$. Technically, since the directions of our kernels are slanted, this does not give the proper direction, but if we are consistent it doesn't matter for our purpose. Now we weight each magnitude by the distance of the pixel from the center, using a gaussian kernel (I used $\sigma = 4$). Then, once you have that, divide the 16 by 16 window into 4 by 4 windows of 4 by 4 magnitude/derivative. For each 4 by 4 windows, we discretize the directions into 8 bins, representing 8 directions (0 to 44, 45 to 89,...). We add the magnitude of each (in-between) pixel to its corresponding bin (could be more fancy by dividing the weigh proportionally to the direction, instead of just discretizing, didn't do that here). Now you have 4 by 4 times 8 bins, or 128 bins. This will be our feature vector, after we normalize it, then setting every value above 0.2 to 0.2, and we normalize it again a final time. We do that for each pixel and we obtain our final 128 features.

This gives us our $1 + 1 + 40 + 128 = 170$ features by pixel.

Finally, to get the feature vectors for each of our segments, we simply average the vectors of each pixels.

4.3 Segment/Pixel Matching

Now that we have our feature vectors, we can compute the best segment(pixel) pairs. For each segment(pixel) t in the target image, we seek the the segment(pixel) r in the reference image which is the closest, where we compute the distance using

$$\mathbf{w}D(r,t) = w_1 D_1(r,t) + w_2 D_2(r,t) + w_3 D_3(r,t) + w_4 D_4(r,t)$$

Where $D_1(r,t)$ is the Euclidean distance between the luminance(1) of r and t, $D_2(r,t)$ is between the standard deviation, $D_3(r,t)$ the 40 gabor features, and $D_4(r,t)$ the Euclidean between the 128 SIFT features, and where $\mathbf{w} = (0.4, 0.1, 0.2, 0.3)$ are the weights for each distances (from [optim], adapted). First we pick the best segments pair, and within each segment compute the best pixel pairs. The weights $\mathbf{w}$ are important to the final result, as sometimes a wrong segment is chosen because the one important discriminative feature was not given enough importance. I could always try to play around the weights to improve the result by a good margin, without adding any new features.

Here, it helps a lot to have small segments, as it reduces the search for the best pixel pairs tremendously. In this step, the background for the images are often one segment, and they match, so they take the most time compute by far despite being straightforward (most often, pure white). I guess this step could be optimized by setting a good enough threshold distance were anything below is considered equivalent, however in my implementation I used Numba and numpy to parallelize and vectorize the process, without this python could never have run the algorithm. Remark, for about any image algorithm, it is better to use C/C++ or at least C#, never python. Thankfully for both the SIFT and pixel pair computation, I knew how to use Numba, which is a JIT compiler for python which speed up python loops 100 folds usually, and is fully compatible with numpy.

4.4 Color Transfer

Now that we have the segments and pixel pairs, we will transfer the color from the reference to the target image U and V dimensions. However, there may be some wrong pixel assignments, which would make the coloration fail. So, for each segment, we only transfer only the best 15% (as in the paper) of the pixels pair (that is those with the shortest distances score within each region). The rest of the pixels will be assigned using an optimization technique described next.

4.5 Colorization By Optimization

Now, we consider the pixel color that we transfered as pseudo scribble, and apply the optimization algorithm used by [2]. We are working on the YUV space. Let $U(\mathbf{r})$ be the color value of channel U at pixel position $\mathbf{r} = (i, j)$. To summarize, we will enforce a constraint that state that each neighbor pixel with similar intensity (Y) have a similar color (U, V). We can treat the channel U and the channel V separately, and I will only describe it for U. Thus, we minimise the squared difference between the color $U(\mathbf{r})$ and the weighted average of the neighbors of $\mathbf{r}$. That is, we aim to minimize the loss function

$$J(U) = \sum_{\mathbf{r}} \left(U(\mathbf{r}) - \sum_{s \in N(\mathbf{r})} w_{\mathbf{rs}} U(\mathbf{s}) \right)^2$$

where $N(\mathbf{r})$ denote the neighbors of $\mathbf{r}$ (with a window size of 1 in this project), and

$$w_{\mathbf{rs}} \propto e^{-(Y(\mathbf{r}) - Y(\mathbf{s}))^2 / 2\sigma_{\mathbf{r}}^2}$$

Here, σ^2 is the variance of the intensity Y around the pixel $\mathbf{r}$, and the sum of the $w_{\mathbf{rs}}$ over the neighbors $\mathbf{s}$ sums to 1 (hence $\propto$).

Essentially, this is the problem of the least square. In fact, here our constraints matrix will be square so we don't even need the pseudo inverse. To explain briefly, we want to minimise $J(U)$, so we set its derivative to 0. By rewriting $J(U)$ in a nice vectorized way and taking the derivative, we can reduce the problem to

$$WU = b$$

Where U is the unknown vector to be solved, W represent the linear constraints matrix between each pixel and b is a known vector, explained shortly. W is a square matrix of size (number of pixels $\times$ number of pixels). It is huge (cannot fit into memory), however it is very sparse (only linear in the number of pixels in the image). I had to use scipy sparse solver, and its Compressed Sparse Row (csr) sparse matrices. I found an implementation using precisely that, I adapted it and commented over it (I could have rewrote more properly the few lines, sorry, but it is just setting up the matrix W and the vector b precisely as explained below).

For each pixel we have one linear constraint. If it is one of the pixel that has a color transferred from the reference image, then we set $b_{\mathbf{r}} = \nu_{\mathbf{r}}$, where $\nu_{\mathbf{r}}$ was the color U that transferred, and we set the corresponding row of W be 1 at the pixel $\mathbf{r}$. This effectively set up the constraint $U(\mathbf{r}) = \nu_{\mathbf{r}}$. If it was not one of the pixel for which we transferred a color, then we set $b_{\mathbf{r}} = 0$ and set the corresponding row of W to 1 at the pixel $\mathbf{r}$ and $w_{\mathbf{rs}}$ for pixel neighbor $\mathbf{s}$ where $w_{\mathbf{rs}}$ is defined as above. This effectively set up $U(\mathbf{r}) - \sum_{s \in N(\mathbf{r})} w_{\mathbf{rs}} U(\mathbf{s}) = 0$. And that is it. The solver takes some time to solve the problem, but it is relatively fast even as the size of the image increase. Once U is solved, we can set up the exact same problem for channel V (same matrix, but the values of b changes since the U and V colors that were transferred are not necessarily the same).

And once that is done, we have our colorized image.

5. Evaluation

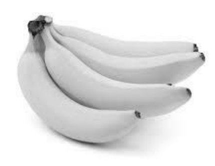
As I said, I will only be doing a qualitative assessment of the results. The quality are not good nor consistent enough to make a proper computational and meaningful comparison, and I found more interesting to talk about the way the algorithm behave given the inputs.

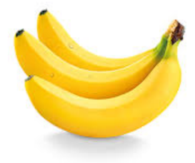
In figure 1 we can see some results of the algorithm. The segmentation that is obtained by the meanshift algorithm give an over segmentation of the bananas (c, d), some large chunks but a lot of small segments too, mostly along the different shadings of the banana (esp. the target segmentation (c)). The colorization of the banana is imperfect, as can be seen by the yellowish color that its shadow took (f). Furthermore, notice how the stem of the banana turned green (f), which was not present in the original picture, but is present in the reference banana stem (b). There are also some light discoloration on the banana, some part took a grey color.

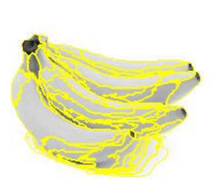
The result obtained with the eggplant shows a typical failure case. The original (i) shiny skin produced a light part on the grayscale image (g), and the algorithm associated that with the top of the eggplant of reference (h), which is green in color, and the wrong pixel input propagated throughout the eggplant. In general, shiny parts will require finding an equally shiny part on the reference image (otherwise the difference in shade will not match), and hope that the part do not match with another light part of the reference image.

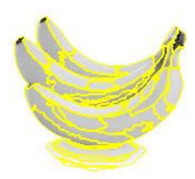
Otherwise, when looking at the part that were well colored, we can conclude that the color chosen and the propagation of it gives a convincing color. This is mostly due to optimising in the YUV colore space, as long as we get the yellow chroma of the reference banana, we are almost garanteed to get a convincing result as an output.

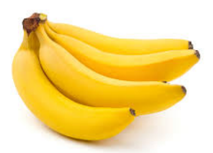
I will go over more examples in the presentation, but overall those represent the typical results. If the result is somewhat convincing like the banana, it will some flaws upon closer inspection. And sometimes (somewhat too often) a wrong match will make the algorithm fail hard. I do not exclude the possibility that this is a result of my poor programming skill, but I suspect that it is in part due to the lack of a feature vector for saliency, used in [7]. Still, even in the original papers results often had some slight imperfection, less that in my results though.

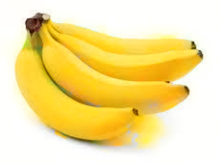

(a) target

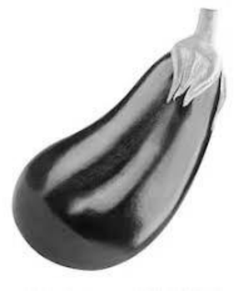

(b) reference

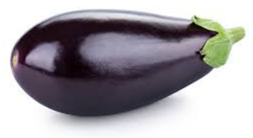

(c) segmentation target

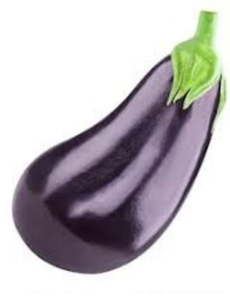

(d) segmentation ref.


(e) original


(f) result


(g) target


(h) reference


(i) original

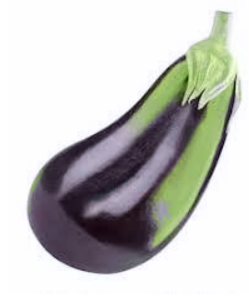

(j) result

Figure 1. Results

6. Conclusion

Image colorization is a hard task of computer vision, which requires the algorithm to make minimal mistakes, otherwise the approximate results will be obvious to the human eye, which for most of us is more than expert at seeing colored images. Still, the optimization method introduced in [2] does a very good job at colorizing the image, if given a correct input. It is also interesting how small details are well grabbed by the algorithm of segment/pixel selection. Some future directions would include augmenting the feature space representation with new and more discriminative features, especially dealing with the shadows and light parts of the image that are caused by the way and angle with which the light hits the object, and also the texture of it. Also, automatic reference selection should be feasible for example by computing SIFT or SURF correspondence, although multiple object in one image may be hard to color that way. However, given that nothing prevented me to add another reference image (which I could have treated as just an extension of the first reference image), an automatic image selection may effective. Furthermore, the SURF correspondences may be useful (for example, they could be use to provide rotational and size information about the object, and this could help improve the resulting segments matching).

References

[1] R. K. Gupta et al. *Image Colorization Using Similar Images.* 2012.

[2] Yair Weiss Anat Levin Dani Lischinski. *Colorization Using Optimization.* 2004.

[3] Dorin Comaniciu and Peter Meer. *Mean Shift: A Robust Approach Toward Feature Space Analysis.* 2002.

[4] Tinne Tyutelaars Herbert Bay and Luc Van Gool. *SURF: Speeded Up Robust Features.* 2009.

[5] David G. Lowe. *Distinctive Image Features from Scale-Invariant Keypoints.* 2004.

[6] Muhammad Tahir et al. Saeed Anwar. *Image Colorization: A Survey and Datatset.* 2022.

[7] Yipin Zhou. *Example Based Colorization Using Optimization.* 2013.