Implementation and Analysis of Levin et al. - Colorization Using Optimization

Patrick Di Rita
pdirita@wustl.edu

10 December, 2019

Abstract

Colorization of grayscale images is a process that has evolved greatly since its inception in the early days of photography and film. What started as a fully manual operation transitioned into semi-automated through the use of iterative algorithms, and eventually became fully automated upon the introduction of convolutional neural networks. In this paper, we focus on a semi-automated algorithm presented by Anat Levin, Dani Lischinski, and Yair Weiss in their paper, "Colorization Using Optimization." We implement their algorithm for colorizing still images through user-supplied color annotations using Python 3.7 with NumPy, and demonstrate the algorithm's strengths and limitations involving image resolution, neighborhood window size, color annotation quality, and output saturation.

1 Introduction

The act of image colorization has historically required a combination of significant user input and expensive computations. Traditional image colorization techniques, even those brought on during the digital age, required an artist to tint each individual object entirely by hand [5]. This task becomes exceptionally time consuming when transitioning from still images to entire feature-length films. Even as computational power continued to improve, colorization methods still required the conjunction of two tedious, time-consuming, and computationally expensive tasks: segmenting images into regions and then tracking those regions across image sequences [7]. Eventually, algorithms such as the one proposed by Levin et al. were developed to reduce manual workload, drastically increasing efficiency.

A continuous decrease in both the computational time and manual input required by image colorization algorithms is required in order for image colorization to be viewed as a worthy endeavor. This push for the best performance is due to the two main groups that utilize the technology: image restoration hobbyists/professionals and the film industry. The former group views the issue as mainly a matter of convenience, as someone looking to colorize grayscale photos or videos of relatives, landscapes, etc. likely has a relatively small volume of images to colorize, while the latter group views the issue as a trade-off between cost (monetary and time), quality, and artistic integrity.

The colorization of classic films and television shows has been historically shrouded in at least some amount of controversy. This is clear simply based on the fact that not every movie and show has been colorized and re-released, even though colorized re-releases of shows, even those that saw only moderate success upon their initial release, are very likely to make a healthy return on investment [5], [7]. The controversy stems from the aforementioned trade-off that the film industry must consider in deciding whether or not to colorize a particular piece of media. Back when colorization was done entirely by hand, the cost of the endeavor greatly outweighed the quality achieved by skilled artists, meaning colorization was reserved only for special occasions. The first "generation" of colorization algorithms resulted in fuzzy or complex color-region boundaries due to the difficulty of tracking fluid regions across frames, and these imperfections would then need to be corrected manually, leading to high cost and only moderate quality [7]. The next generation of algorithms, such as the one presented by Levin et al., aimed to decrease the amount of artist intervention required to produce a high-quality colorization, but still required a nonzero amount. In recent years, fully-automatic high-quality colorization processes have been developed, and this has resulted in a decrease in cost and increase in quality to the point where the only barrier a film faces is whether or not the filmmakers and audience see modifying the original film as artistically "adulterous."

In this paper, we focus on implementing and testing the limitations of Levin et al.'s "Colorization Using Optimization" method, which relies on the assumption that neighboring pixels in space-time that have similar intensities should have similar colors [7]. Through this

assumption, the algorithm propagates an artist's color annotations (also referred to as "scribbles" or "markings") on one frame through space and time to fully colorize an image or short sequence. This is done through standard optimization of a quadratic cost function parameterized by monochromatic luminance (i.e. intensity) of the original grayscale image and the chrominance channels of the artist's scribbles.

Although it is outside the scope of this paper (and its inclusion would prove fairly redundant), this algorithm also excels at selective recoloring or color-correction within an image, processes used consistently in digital photography and in special effects [7].

2 Background & Related Work

T. Horiuchi proposed a similar process to Levin et al., in which a user specifies a suitable color on each isolated pixel of an image, and the rest of the image is then colorized through probabilistic relaxation. The algorithm assumes local Markov property on the images, and then minimizes the total of RGB pixel-wise differences [1]. The problem then becomes one of combinatorial optimization, and the algorithm has been shown to work well in practice, especially when (even a small percent of) color pixels are known with high confidence. Other semi-automatic methods such as these are discussed briefly in Levin et al.

These semi-automatic methods, however, still require a non-zero amount of manual color selection, meaning they are entirely unable to generalize to out-of-sample data and require continuous user supervision and tweaking to preform optimally. In a paper presented by Zhang et al., this issue is combated through the use of a convolutional neural network (CNN). By posing the problem as a classification task and class-rebalancing at classsification time, a feed-forward CNN that has been trained on millions of color images is used to colorize grayscale photographs fully automatically. This results in much more vibrant and believable colorizations than the aforementioned approaches, which are notorious for producing desaturated colors, and has been shown to successfully fool humans on 32% of trials [2].

3 Proposed Approach

The algorithm described by Levin et al. can essentially be broken down into three main components: color space adjustment, neighborhood weighting, and optimization. Although the original paper describes these steps briefly, it is this author's opinion that said descriptions are insufficient in helping to understand the entire algorithm at a glance, so we will delve further into each step here. Within the context of this paper, we assume that we are working to colorize singular images, instead of sequences of frames. The original research implements a sequence colorizer using a multigrid solver in C++, the implementation of which is out of the scope of this paper. In our case, we implemented the still-image colorizer (originally in MatLab) using Python 3.7 and NumPy.

3.1 Color Space Adjustment

The algorithm takes two images as input: the grayscale image we intend to colorize and a copy of that image marked with a user's color scribbles (which indicate the true color for that region). Both images exist initially in the RGB colorspace (with all three channels being equal in the grayscale case) and are represented as $n \times m \times 3$ NumPy arrays. Using the sum of the difference between these arrays along their third axis, we generate a boolean *isColored* matrix (of size $n \times m$), where an element is marked $True$ if said sum of differences is greater than 0.01, $False$ otherwise. We then convert each image to the YIQ colorspace using the following matrix multiplication [3]:

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.5959 & -0.2746 & -0.3213 \\ 0.2115 & -0.5227 & 0.3112 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad (1)$$

Note that the original paper indicates that the images are to exist in the YUV colorspace, and makes no mention of YIQ. The authors' implementation, however, makes use of Matlab's *rgb2ntsc* function, which, in fact, converts to YIQ. YUV (the European PAL broadcast standard colorspace) is simply YIQ (the American NTSC broadcast standard colorspace) with the I and Q components rotated $-33°$ about the Y-axis. To keep consistent with the paper, we also opted to use the YIQ colorspace.

Now that the images have been converted, we create a new $n \times m \times 3$ NumPy array by stacking the first channel (Y channel) of the converted grayscale image and the second two channels of the marked image along the third axis, creating a composite YIQ image where Y is the monochromatic luminance of the original grayscale image and I, Q are the chrominance of the markings [7]. This composite image is then passed, along with the *isColored* matrix and an optional window radius parameter (which defaults to 1), to the next step of the algorithm.

3.2 Neighborhood Weighting

We first initialize a $n \times m$ indices matrix (which gives each pixel a unique identifier) along with row indices, column indices, and values matrices of the window (with size

based on the user-specified window radius w_{rad}). The row and column index matrices will eventually be used to index a sparse matrix that we will run linear optimization on. The values matrix will contain the weighs of each pixel, and will be passed as the data to the same sparse matrix. We then iterate over the entire image, and at pixels where *isColor* is false, we set the window row and column index values to the pixel identifier, and we set the window value for that pixel to the corresponding value of the Y channel in the composite image. The window is now populated with luminance values, Y_r through Y_s, of the image for the area spanned by index vectors r through s, which correspond to pixels in the neighborhood (where r is in same neighborhood as s, i.e. r is in the window centered at s). So $Y_r - Y_s$ is a row vector containing difference between the luminance value of each pixel in the window and the luminance value of the center pixel in the window. We then calculate the variance of the window, σ_r^2.

We are now ready to weight the neighborhood using equation 2, which is based on the squared difference between the two intensities (large weight when Y_r is similar to Y_s, small weight otherwise):

$$W_{rs} \propto e^{-\frac{(Y_r - Y_s)^2}{2\sigma_r^2}} \tag{2}$$

[7]. This weight vector is normalized so that it sums to 1, and the slice of the values matrix ranging from r to s is set equal to W_{rs}. For pixels at which *isColor* is true, we simply set the values matrix for that pixel equal to 1 (as they are already solved for), and the row and column indices matrices equal to the pixel identifier. Once this process has been carried out over the entire image, we are ready to optimize.

3.3 Optimization

Our goal is to minimize $J(U), J(V)$, subject to the user's color scribbles, where

$$J(U) = \sum_r (U_r - \sum_{s \in N(r)} (W_{rs} U_s))^2 \tag{3}$$

The notation $s \in N(r)$ means that s is in the neighborhood of r (i.e. s and r are within the same window) [7].

To model this, we utilize SciPy's *sparse* library to populate a Compressed Sparse Row matrix (CSR). The data we use to populate the CSR is our values matrix, the CSR is indexed by our row and column index matrices, and the CSR is shaped using the total number of colored pixels plus the total number of window pixels and the image size. This matrix is our "A" value for the linear equation $Ax = b$.

A = sparse.csr_matrix((vals, (rowInd, colInd)), (pxl, sz))

[6] Then, for both the I and Q channels, we set our "b" vector to be the colored pixels' chrominance values for the channel. We call SciPy's *linalg.spsolve* function on *A* and *b*, reshape the returned matrix to match the image, and set the corresponding channel in the output image to this matrix. The Y channel of the output image is always the Y channel of our original composite image.

This optimized image is then converted back to RGB using the inverse of the YIQ conversion matrix (equation 1), and the RGB image is output as the final result.

4 Experimental Results

Our first experimental task was to test our implementation on the images and markings given in the original paper, and see whether or not our results matched. As seen in figures 2 and 3, our results did, in fact, match the paper's, as desired.

The next step was to find images other than those supplied in [7], convert to grayscale, add color scribbles, and compare our implementation's colorization to the true colors. Our initial hypothesis was that high-contrast images would preform better, so we took the image seen in Figure 1 with a smart phone, and attempted to run our algorithm on it directly using the markings found in Figure 4a. This proved to be extremely time consuming, as the initial image was in 4K resolution. This brought to light one of the main limitations of the algorithm, which is that it works best for relatively low-resolution images due to the fact that it iterates over each pixel individually. We then scaled the image and markings down to 160×213 (maintaining the aspect ratio) and re-ran the algorithm, again using the markings found in 4a. To this author's surprise, the result (Figure 5a) was extremely poor. An attempt was made to remedy this fact by adding more scribbles (Figure 4b) and then running our algorithm with a window radius of 3 (Figure 5b). This output was, once again, undesirable, and we got the same result when increasing window radius to 5 (Figure 5c). This lead to the realization that the image works best when every isolated surface has a marking, and essentially does nothing but output the markings image otherwise.

We tested this hypothesis on another image (Figure 6a), this time with many more markings (Figure 6b). We chose to go with a headshot-style image as we knew the algorithm had preformed well on similar input. Our hypothesis was mostly confirmed, as the result (Figure 7a) was much closer to the true colors than the backpack trials were. This time, increasing window radius to 3 lead to an even better distribution of colors (Figure 7b) while simultaneously decreasing the opacity of the scribble lines. Increasing window size, however, came with the trade-off of increased

color bleed between regions, which can clearly be seen by the large erroneous blue patch on the right shoulder of Figure 7b, as well as greater processing time. The colors had an overall lower saturation than the true image's, but this is to be expected with methods that do not utilize CNNs.

Our final experiment was to use a simple image with hard borders and clearly defined color regions, in order to further confirm our hypothesis that the best performance is achieved when each and every color region has a corresponding marking. We picked a stock image of a pack of Trident gum for this test (Figure 8a) and gave the algorithm a very detailed marking image (Figure 8b). We initially used a window radius of 1, which showed very promising but sub-optimal results (Figure 9a). Increasing the window radius to 3 gave us the best results out of any of our experiments using independently-gathered images: the colors on the pack of gum were nearly perfectly replicated, with the only difference being the expected desaturation and color-bleed into the whitespace around the pack of gum (Figure 9b).

As an aside, an attempt was made to implement the Lucas-Kanade algorithm to compute optical flow in order to propagate markings through time (in the case of a GIF or sequence of images). The implementation was unsuccessful due to the limitations of Lucas-Kanade. This author hypothesizes that an implementation of the Gunner Farneback dense optical flow algorithm may be more suitable, especially due to the existence of efficient computer vision libraries such as OpenCV.

Figure 1: Smartphone image of backpack and water bottle against simple background, used as the first independently-sourced image to test colorization on. Revealed shortcomings in the algorithm stemming from image resolution and marking comprehensiveness

5 Conclusion

Despite the confidence Levin et al. clearly display within their paper, the algorithm shows its age when presented with very high-resolution images or when compared to contemporary processes utilizing convolutional neural networks. The authors of the original paper assert that the algorithm produces excellent colorizations from "a surprisingly small amount of user effort," [7]. To an outside observer using the algorithm for the first time, however, the amount of markings required for a proper colorization is much greater than originally expected (as can be seen in the backpack example).

These shortcomings aside, the algorithm does, in fact, do an exceptional job of colorizing images when supplied with proper combination of image resolution, color markings, and window radius. We observed this directly when testing the algorithm on the original paper's images/colorings and on the independently-sourced headshot and trident gum images. The high requirement for proper output is what has resulted in the further development of colorization technology, which has lead to the creation of fully-automatic methods that replicate colors more accurately with zero user input. Due to the fact that algorithms in the same "generation" as the one described in this paper have been essentially entirely phased out by convolutional neural networks, it can not be said that our algorithm entirely solves the problem that users in image editing and film fields face when tasked with coloring an image.

Acknowledgments

We would like to thank the developers of NumPy and SciPy [4], [6] for their work in creating extremely powerful tools for mathematical programming within Python. We would also like to thank A. Peshin and S. Pigeon [3], [5] for their articles describing colorization in movies and YUV/YIQ colorspaces, respectively.

References

[1] T. Horiuchi, "Colorization algorithm using probabilistic relaxation," *Image and Vision Computing*,

(a) Grayscale image (b) Grayscale image with user scribbles (c) Our implementation's output

Figure 2: First test on data from [7]. The colorized output seen in Figure 2c matches the original paper's output exactly.

(a) Grayscale image (b) Grayscale image with user scribbles (c) Our implementation's output

Figure 3: Second test on data from [7]. The colorized output seen in Figure 3c matches the original paper's output exactly.

vol. 22, pp. 197–202, Mar. 2004. DOI: 10.1016/j.imavis.2003.08.004.

[2] R. Zhang, P. Isola, and A. A. Efros, *Colorful image colorization*, 2016. [Online]. Available: https://richzhang.github.io/colorization/.

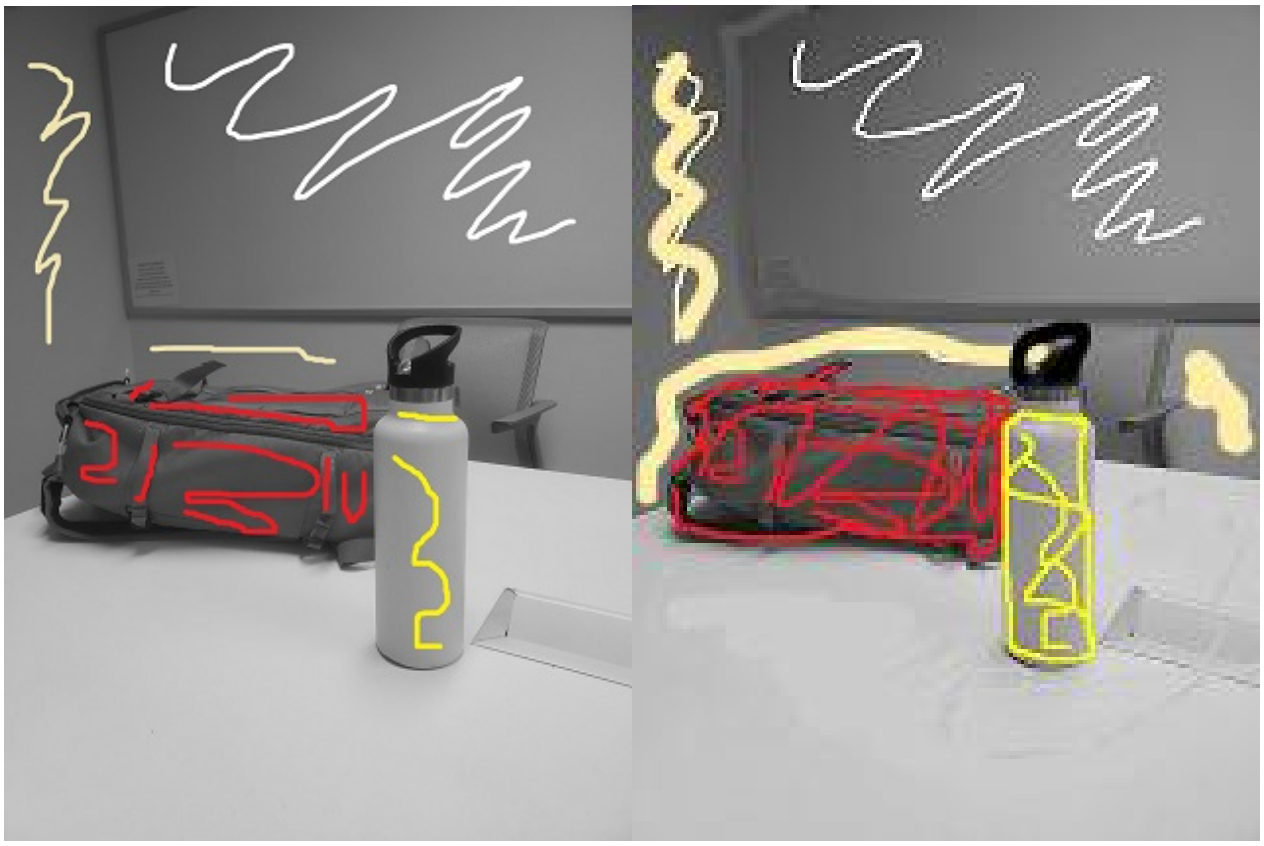
[3] S. Pigeon, *Yuv and yiq (colorspaces v)*, 2018. [Online]. Available: https://hbfs.wordpress.com/2018/05/08/yuv-and-yiq-colorspaces-v/.

[4] *Numpy reference 2019*, 2019. [Online]. Available: https://docs.scipy.org/doc/numpy/reference/.

[5] A. Peshin, *How are black and white films colorized?* 2019. [Online]. Available: https://www.scienceabc.com/innovation/how-are-black-and-white-films-colorized.html.

[6] *Scipy reference 2019*, 2019. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/.

[7] A. Levin, D. Lischinski, and Y. Weiss, *Colorization using optimization*. [Online]. Available: https://www.cse.huji.ac.il/~yweiss/Colorization/index.html#still.



(a) First attempt markings

(b) Second attempt markings

Figure 4: All marking trials for backpack image

(a) First attempt output. Markings from 4a, window radius 1

(b) Second attempt output. Markings from 4b, window radius 3

(c) Third attempt output. Markings from 4b, window radius 5

Figure 5: All output trials for backpack image, none of which turned out as desired due to ambiguity of background colors and lack of sufficient overall markings

(a) Headshot true colors

(b) Headshot markings

Figure 6: Headshot trial image inputs

(a) Headshot first attempt output. Markings from 6b, window radius 1

(b) Headshot second attempt output. Markings from 6b, window radius 3

Figure 7: Headshot trial image outputs. Figure 7b turned out substantially better than Figure 7a due to the increased window radius

(a) Trident gum true colors, image obtained through the Amazon listing for a pack of Trident sugar-free gum

(b) Trident gum markings

Figure 8: Trident gum trial image inputs

(a) Trident gum first attempt output. Markings from 8b, window radius 1

(b) Trident gum second attempt output. Markings from 8b, window radius 3

Figure 9: Trident gum experiment outputs. Increasing window size from 1 (Fig. 9a) to 3 (Fig. 9b) resulted in a very satisfactory output.