From Grayscale to Color: Digital Image Colorization using Machine Learning

Cris Zanoci and Jim Andress

December 11, 2015

1 Introduction

Image colorization is the process of adding colors to a grayscale picture using a colored image with similar content as a source. Colorization techniques are widely used is astronomy, MRI scans, and black-and-white image restoration. However, the colorization problem is challenging since, given a grayscale image, there is no unique correct colorization in the absence of any information from the user. Even though many implementations require the user to specify initial colors in certain regions of the picture, our project focuses on automatic image colorization without any additional input from the user.

In this paper we describe our attempt to solve this challenging problem. We take as an input a single colored image to train on and a grayscale image of similar content which we will colorize. The algorithm begins by creating a high dimensional representation of the textures found within the colored image. After training a collection of SVMs on this input image, we then phrase the color transfer process as an energy minimization problem. Our approach strives to reconcile the two competing goals of predicting the best color given the local texture and maintaining spatial consistency in our predictions.

2 Related Work

The idea of using machine learning to colorize images first became popular in the early 2000s. Welsh et al. began with an algorithm which transferred colors simply based on the value and standard deviation of luminance at each pixel [1]. Although the algorithm runs quickly, the results we obtained by implementing their method are far from optimal in that each prediction is purely local and that the feature space they are searching over has only two dimensions. Soon after, Levin et al. took colorization in a completely different direction with their reformulation as a global optimization problem over the image [2]. Unlike Welsh's technique, which accepted a color image as training input, Levin's optimization algorithm accepted hand-drawn "scribbles" of color from the user which are then used as linear constraints for the optimization of a quadratic cost function. This algorithm sought to label pixels the same color if they had similar luminance values, which is one of the first examples of automatic colorization algorithms explicitly incorporating spatial coherency as part of their classification.

Irony et al. then improved the technique by incorporating a two-phase voting procedure [3]. In the first step, pixels are independently colored based on their individual texture features. In the second, each pixel's color is replaced by a weighted majority vote over its neighboring pixels. The most confidently labelled pixels are then fed into Levin's original algorithm as "micro-scribbles." Noda et al. formulated the problem as Bayesian inference using a Markov random field model of an image [4]. This formulation naturally includes spacial coherency considerations while computing the maximum a posteriori (MAP) estimate of the colors. Finally, one of the more recent takes on the colorization problem was presented by Charpiat et al., on whose algorithm ours is based [5]. This technique involves solving an energy minimization problem, the details of which will be laid out in Section 4.

3 Dataset and Features

One key feature of our project is that it does not require a large corpus of training examples; rather, every execution of our algorithm takes only one color image and one grayscale image as input. We used publicly available pictures with a typical resolution of 600×400 pixels, although there is no constraint on the size of our

input. Because our algorithm will ultimately be applied to grayscale testing images, the features we extract and learn upon must all be derivable from this type of image. In particular, none of the features can be based on information from the color channels of the training input. Thus, we attempt to match regions with similar content in both images based on the similarity of the underlying textures in the luminance channel. In our implementation, we use SURF and FFT descriptors to extract information about textures.

SURF stands for *Speeded Up Robust Features* and was first introduced by H. Bay et al. in 2006 [6]. First, it uses a blob detector based on an approximation of the Hessian matrix to locate changes in the image's texture. These changes are associated with interest points, which identify the distinctive patters of the picture. For every interest point, horizontal and vertical Haar wavelet responses are computed in the sub-regions of a square centered at that point. These responses are then weighted and reduced to a lower dimensional space to produce a 128-dimensional local descriptor. One important criterion for choosing SURF is its invariance under geometrical transformations, like scaling and rotation. Thus the descriptor is consistent across different parts of the picture and even across different pictures. Moreover, computing SURF features is relatively fast, which allows us to apply it to all the pixels.

We also compute the Fast Fourier Transform (FFT) as part of our feature vector. FFT represents the image as a sum of exponentials of different magnitudes and frequencies and can therefore be used to gain information about the frequency domain of our image. It is also useful for finding correlations between images because the convolution function is similar to the correlation function [7].

For every pixel p in the training and testing images, we consider a 15×15 pixel window centered at p. The dimension of the window was chosen so that it captures enough information about the neighborhoods of p but doesn't slow down our implementation. By applying SURF to this window and two blurred version of it, we extract 3 SURF vectors, which we then concatenate to form a 384-dimensional feature vector. To that vector we add the 225 FFT features as well as the mean and standard deviation of the luminance around p. We end up with a 611-dimensional feature vector, which we further reduce to only 30 dimensions by applying PCA. Although the dimensionality of the data has been significantly reduced, we found that these top 30 principle components retain around 70% of the variance in the data, meaning that we are still able to richly describe the image textures using this feature space.

4 Methods

Our colorization algorithm is as follows. We begin by discretizing the colors of the training image using k-means clustering. Rather than running the k-means over the three dimensional RGB coordinates of each pixel, we instead convert the input images to the $L\alpha\beta$ color space, where L represents the luminance of a pixel and α, β are the two components of its color. This space is specifically designed to match a human's perception of color, which means that the centroids produced through clustering will represent a natural division of the colors. For k-means, we typically use 32 colors. By running our algorithm with various values of k, we have found that using more than 32 colors does not improve the results.

Next, we extract a feature vector at every pixel of the training image and use PCA to reduce its dimension as described in Section 3. Then we train a set of SVMs, one per discretized color. We use a one vs. all classification procedure, which means that each SVM predicts whether or not a pixel has its corresponding color. To be less sensitive to outlier pixels, we employ ℓ_1 regularization:

$$\min_{b,w} \frac{1}{2}\|w\|^2 + C\sum_i \xi_i \tag{1}$$

$$\text{s.t. } y^{(i)}(w^T x^{(i)} + b) \geq 1 - \xi_i, i = 1, 2, ..., m \tag{2}$$

$$\xi_i \geq 0, i = 1, 2, ..., m \tag{3}$$

where C is a parameter that we can vary. We tried using two types of kernels, RBF and linear, and chose the latter as it yields comparable results while also being significantly faster.

For testing, we read in the grayscale image, compute the feature vector at every pixel and project it into the subspace determined by PCA in the training step. At this point, it is tempting to simply run the SVMs on each pixel and assign the color of the maximum-margin SVM. However, this would completely disregard

the spatial coherency element of image colorization. Instead, we make the reasonable assumption that colors are most likely to change where there are edges in the luminance channel of the image. We therefore use the Sobel operator to approximate the norm of the gradient of the luminance at each pixel. If we treat the image as a graph with edges connecting pixels, then values computed by the Sobel operator indicate how weakly a pixel couples to its neighbors. After computing the SVM margins and Sobel edge weights at each pixel in the training image, we feed all of this information into the energy minimization problem with cost function

$$\alpha \sum_{p} -s_{c(p)}(p) + \sum_{p,q \in \mathcal{N}(p)} \frac{\|c(p) - c(q)\|}{2(w(p)^{-1} + w(q)^{-1})^{-1}} \tag{4}$$

Here p is a pixel, $\mathcal{N}(\cdot)$ represents a pixel's eight neighbors, $c(\cdot)$ is the color chosen for a particular pixel, $s_c(\cdot)$ is the margin when the SVM associated with color c is used to classify a pixel, and $w(\cdot)$ is the Sobel edge weight at a pixel.

Although it seems like finding the color assignment which minimizes Equation 4 would be difficult (in many cases this problem is, in fact, NP-hard), we can use the Boykov-Kolmogorov algorithm to efficiently compute approximate solutions [8, 9]. This algorithm works by iteratively expanding the set of pixels given a particular color by constructing a graph in which each pixel is connected to its neighbors, a node n_1 representing the color being expanded, and a node n_2 representing all other colors, each with appropriate weights. By computing a minimum cut of this graph, we are left with some pixel nodes connected to n_1 and some to n_2, but none connected to both. We then expand the labelling so that the nodes connected to n_1 are given the color associated with it and the others remain the same. This procedure is proven to converge to within a known factor of the global optimum, and typically finishes in tens of seconds for images with 600×400 pixels.

5 Results

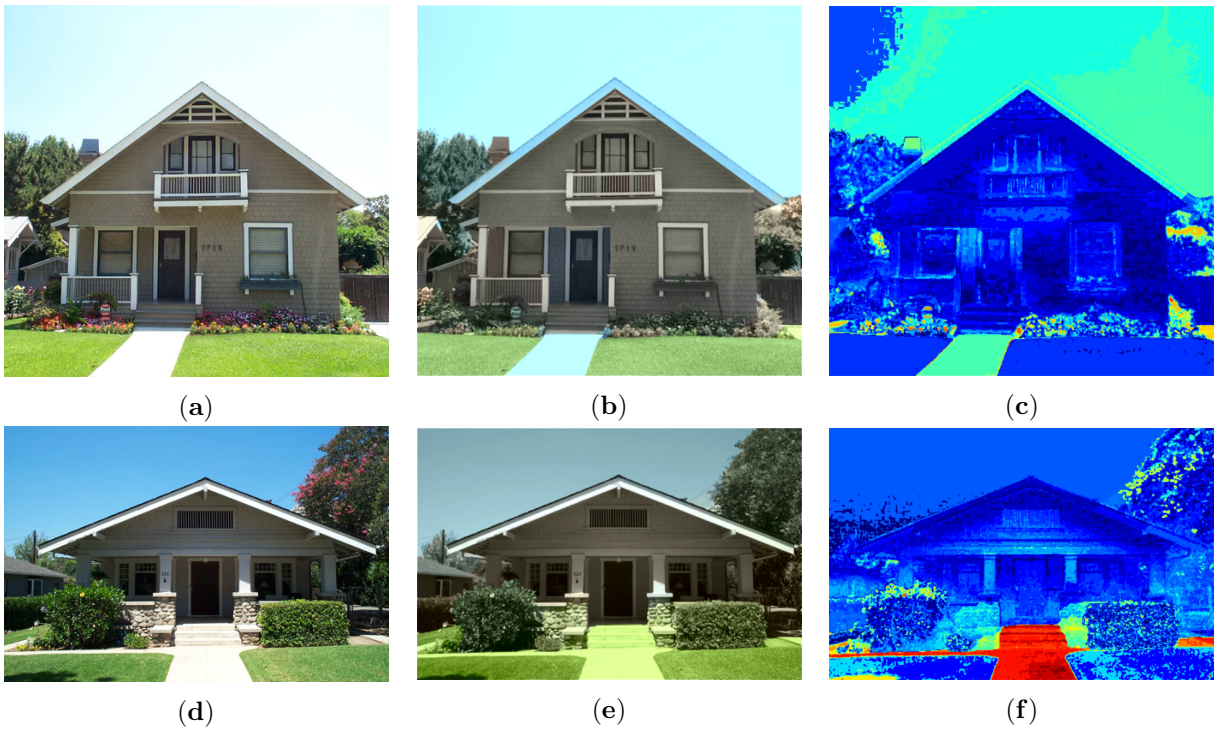
One challenging aspect of this project is the fact that without additional constraints, the image colorization problem does not have a unique correct solution, making it difficult to quantify the accuracy of our colorization. This means that defining a metric to evaluate colorization results is of the utmost importance. We therefore now describe the motivation behind our custom scoring function. First, note that in order to determine the accuracy of a colorization, we must have access to the full color version of the testing image. Because the colors in the original test image are not discretized, it is impossible for our algorithm to reconstruct the image with 100% accuracy. However, we can compute a "best-case" colorization, in which the α, β channels of each pixel in the full color test image are replaced by the values from the nearest of the 32 colors selected from the training image. For each pixel p_i, we then compute the norm of the difference between our colorization $c(p_i)$ and this best-case colorization $c_0(p_i)$ in $L\alpha\beta$ space. The final score of a colorized image is the sum of these norms divided by the number of pixels m, so that we can compare the scores across images of different sizes.

$$Score = \frac{1}{m} \sum_{i=1}^{m} \|c_0(p_i) - c(p_i)\| \tag{5}$$

A score of zero would indicate that our colorization is as accurate as possible with respect to the original image, whereas a score of $256 \cdot \sqrt{2} \approx 362$ is the worst possible score. We choose to define our score function in this way, in terms of the best case colorization, since it involves measuring our performance against an obtainable goal.

Having thus defined our score function, we are able to quantize the results of our colorization algorithm. Figure 1 shows our colorization results on two images drawn from the Pasadena Houses dataset. The colorization shown in 1(b) received a score of 14.4 and the one in 1(e) received 13.6. Figure 2(a) shows how the α parameter in Equation 4 impacts the score of the colorized image. If α is too low, the dominant term in Equation 4 is the right hand spatial coherency term, meaning that the algorithm tends to assign the entire image the same color. If α is too high then the left term dominates, meaning that we lose spatial coherency and will have random patches of color. We see that an α value around 32 seems to produce optimal results.

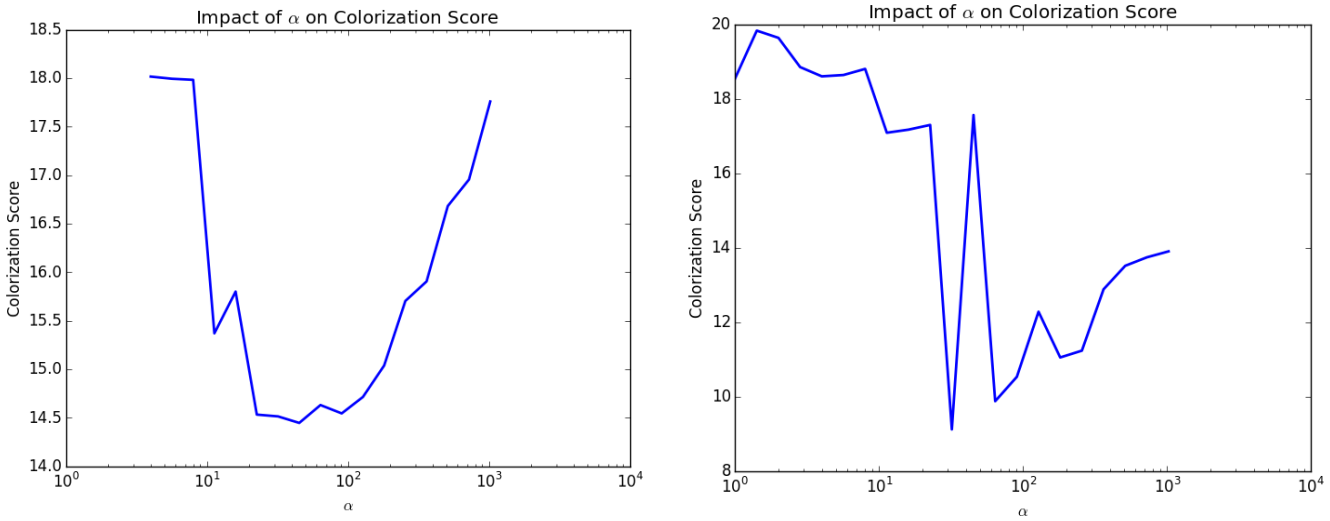
Figure 3 shows another set of results from our algorithm. Note that 3(c) demonstrates a colorization of an image using the same image as the training data. Understandably, this test achieves the lowest score we've seen



(**a**) (**b**) (**c**)

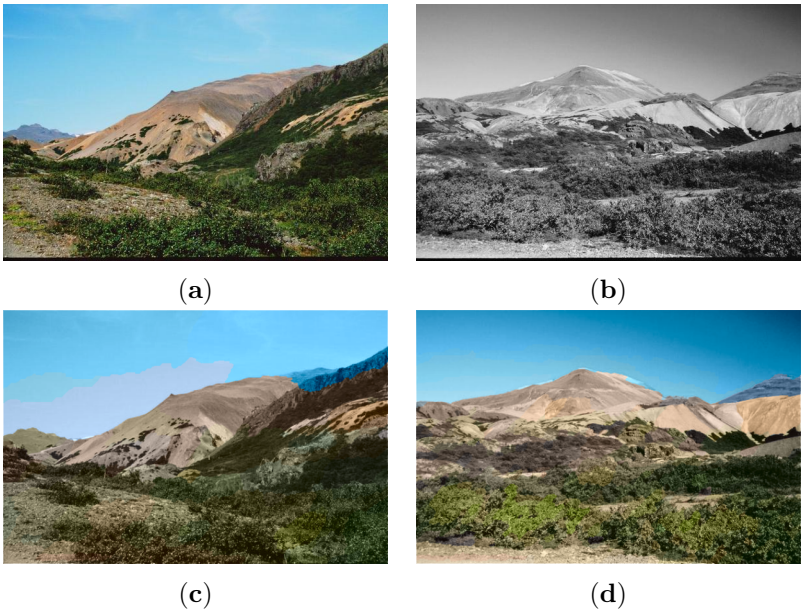
(**d**) (**e**) (**f**)

Figure 1: Colorizations of house images. (**a**) Original image of first house. (**b**) Results from training on **d** and testing on **a**. (**c**) The accuracy of the predicted colors in **b**. (**d**) Original image of second house. (**e**) Results from training on **a** and testing on **d**. (**f**) The accuracy of the predicted colors in **e**.



(**a**) (**b**)

Figure 2: Graphs demonstrating the effect that the α parameter in Equation 4 has on the score of a colorization. Note that both graphs show that a value from 30 to 60 is optimal. **(a)** The impact of α on the score when colorizing Figure 1(a). **(b)** The impact of α on the score when colorizing Figure 3(a).



(**a**) (**b**)

(**c**) (**d**)

Figure 3: Colorizations of landscape images. (**a**) Original colored training landscape. (**b**) Original grayscale testing image. (**c**) Results from training and testing on **a**. (**d**) Results from training on **a** and testing on **b**.

on any input, 9.1. However, we can see that the algorithm performs almost equally well on novel testing data. We unfortunately do not have access to the original color version of Figure 3(b) and therefore cannot compute a score for this colorization. Nevertheless, the algorithm's similar behavior on both testing and training data suggests that we have been able to avoid overfitting while constructing our model. Figure 2(b) shows the effect of α on the score of the colorization in Figure 3(c).

6 Conclusions

In this paper, we have presented and implemented a method for automated image colorization that only requires the user to specify an input training and test image. The proposed algorithm is based on minimizing the cost function in Equation 4. Our approach takes into account both the best color assignment for individual pixels, as given by the SVM margins, and the color consistency within a neighborhood of each pixel, as indicated by the Sobel edge weight term. Because it solves the colorization problem from a global perspective, our method is more immune to outliers and local prediction errors. We have also shown that it is possible to make accurate predictions about an image's color solely based on the features extracted from its luminance channel.

As seen by our resulting images, our framework is applicable to a variety of inputs. The algorithm performs well as long as the training and testing images have well defined, rich textures. However, our algorithm does poorly on images with smooth textures, like those of human faces and cloth, in which it often attributes one color to the entire image.

Our approach can be further extended to accept multiple training images by taking the union of the discretized colors and the union of the feature spaces of those pictures. It would be interesting to apply state-of-the-art computer vision techniques, like convolutional neural networks, to our colorization problem and see whether the addition would further improve our results.

Our algorithm was implemented in Python 2.7 using the OpenCV [10], NumPy [11], scikit-learn [12], and Pygco [13] libraries. It generally takes around 30 minutes to train and test our algorithm on two 600×400 pixel images. Our code is available at https://github.com/jandress94/cs229-cris-jim

References

[1] T. Welsh, M. Ashikhmin, and K. Mueller, "Transferring color to greyscale images," *ACM Transactions on Graphics (TOG)*, vol. 21, no. 3, pp. 277–280, 2002.

[2] A. Levin, D. Lischinski, and Y. Weiss, "Colorization using optimization," in *ACM Transactions on Graphics (TOG)*, vol. 23, pp. 689–694, ACM, 2004.

[3] R. Irony, D. Cohen-Or, and D. Lischinski, "Colorization by example," in *Eurographics Symp. on Rendering*, vol. 2, Citeseer, 2005.

[4] H. Noda, H. Korekuni, N. Takao, and M. Niimi, "Bayesian colorization using mrf color image modeling," in *Advances in Multimedia Information Processing-PCM 2005*, pp. 889–899, Springer, 2005.

[5] G. Charpiat, I. Bezrukov, Y. Altun, M. Hofmann, and B. SCH, "Machine learning methods for automatic image colorization," *Computational photography: methods and applications. CRC Press, Boca Raton*, pp. 395–418, 2010.

[6] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool, "Speeded-up robust features (surf)," *Computer vision and image understanding*, vol. 110, no. 3, pp. 346–359, 2008.

[7] R. C. Gonzalez and E. Richard, "Woods, digital image processing," *ed: Prentice Hall Press, ISBN 0-201-18075-8*, 2002.

[8] Y. Boykov, O. Veksler, and R. Zabih, "Fast approximate energy minimization via graph cuts," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 23, no. 11, pp. 1222–1239, 2001.

[9] Y. Boykov and V. Kolmogorov, "An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 26, no. 9, pp. 1124–1137, 2004.

[10] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[11] E. Jones, T. Oliphant, P. Peterson, *et al.*, "SciPy: Open source scientific tools for Python," 2001.

[12] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[13] A. Mueller, "Graphcuts for Python: pygco," 2012.