University of California, Irvine

EECS 195: Network Science

Colorization Utilizing Unsupervised Methods

Timothy Do, Matthew Prata, Jorge Ragde, and Alex Wang

March 15th 2022

1 Introduction

In this work we address the task of colorization using deep learning. In particular, we achieve colorization using generative models. Our motivation is that colorizing images can be an extremely time-consuming process. Machine learning can provide simple and accurate solutions to this problem.

One popular and successful method is semantic segmentation, where the user supplies a plausible or desired color scheme to the gray-scale image in several places and the model then fills in those colors to match the partitioned image. In contrast, this project will attack the problem by producing a *plausible* color version of the photograph in a completely automated fashion. This is clearly an under-constrained problem without user intervention, but we will be embracing the uncertainty and primarily utilizing the lightness channel of the image to assign a color.

1.1 Contemporary Approaches

The paper by Zhang et al. [1] approached the problem by implementing a feed-forward pass in a Convolutional Neural Network (CNN). The paper tries to achieve a "plausible colorization" to potentially fool a human observer. It utilizes a Lab colorspace, 'L' being lightness channel, 'a' and 'b' are color channels (Fig 1). They utilize a loss tailored to colorization problems, which results in a less desaturated image.

Figure 1: Results from Zhang's Colorizer Model

The paper by Levin et al.[2] is based on the principle of neighboring pixels that have similar intensities, should have similar colors. In their approach the user only needs to annotate the image with a few color scribbles, and the indicated colors are automatically applied to produce a fully colorized image or sequence. The paper uses the YUV color space. Y is the monochromatic luminance channel, referred to as intensity, while U and V are the chrominance channels, encoding the color.

Figure 2: Colorization using User Annotations

2 Approach

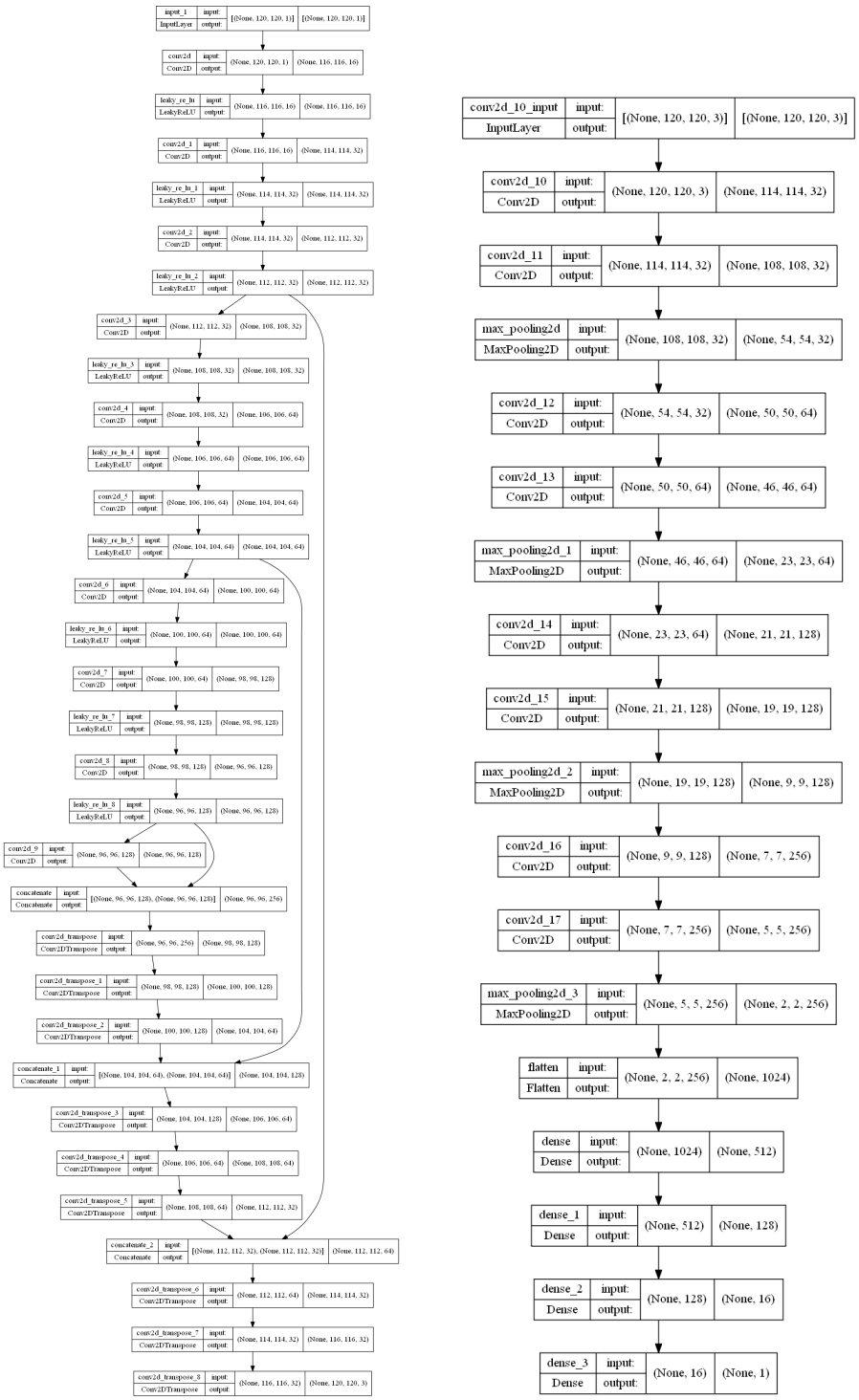
We trained both a Generative Adversarial Network and a Variational Autoencoder as both are unsupervised and allowed us to avoid performing classification. We formatted the data into numpy arrays and fed them into each individual model, splitting the training set into halves to consider validation data in adjusting hyperparameters of the model. The advantage by both of these approaches is that we did not have to do significant preprocessing.

2.1 Generative Adversarial Network (GAN)

GANs are composed of two neural networks, a *generator* and a *discriminator*. The former attempts to generate data 'similar' to the training data, and the latter attempts to discern the training data from generated data. It is best to define similarity in the sense of probability distributions, where the objective of the GAN is then to use $X \subset \mathbb{R}^n$ which has distribution η and would like to find ν which is similar to η. We want to find a mapping $G : \mathbb{R}^d \to \mathbb{R}^n$ such that if a random variable $Z \in \mathbb{R}^d$ has distribution η, then $G(Z)$ has distribution ν. In particular, we want a mapping G such that G^{-1} maps subsets of $\mathbb{R}^n$ to subsets of $\mathbb{R}^d$, that is, a continuous mapping, and $\nu \circ G^{-1} = \eta$. For our particular problem, we have treated the distribution of colors of our images as a random variable in $\mathbb{Z}^3$, since we are using CIELAB to represent colors. For our loss functions, we utilized a binary cross entropy loss function (also known as log loss) for our discriminator, and a mean square error loss for our generator.

$$L_{BCE} = -\frac{1}{N}\sum_{i \leq N} y_i \cdot \log \hat{y}_i + (1 - y_i) \cdot \log (1 - \hat{y}_i)$$

$$L_{MSE} = \frac{1}{N}\sum_{i \leq N} (y_i - \hat{y}_i)^2$$



(a) Generator **(b)** Discriminator

Figure 3: Overview of GAN architecture, note the large number of convolutional layers

2.2 Variational Autoencoder (VAE)

Variational Autoencoders are alternative generative models to GANs. Instead of a normal autoencoder, which produces a coding for a given input, our encoder layer produces a *mean coding* μ and standard deviation σ.The actual coding is then sampled randomly from a Gaussian distribution with mean μ and standard deviation σ. After that, our decoder functions normally. Our latent loss function is (derived in [3]):

$$L = -\frac{1}{2}\sum_{i \leq N} 1 + \log \sigma_i^2 - \sigma_i^2 - \mu_i^2$$

where σ_i and μ_i represent the mean and standard deviation of the ith component of the codings.

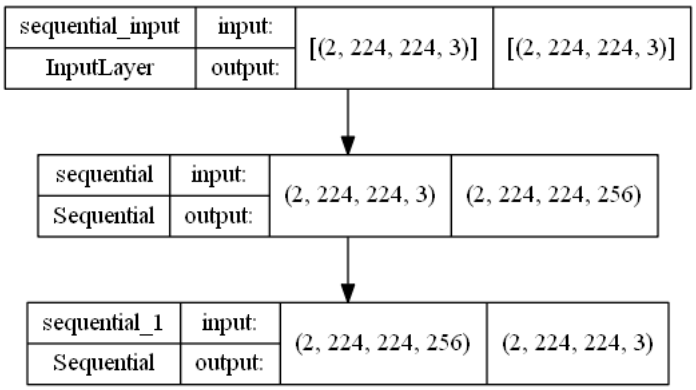


Figure 4: Architecture of our autoencoder, each sequential layer is composed of convolutional layers, batch normalization layers, and transpose layers. First layer is input, second is encoder, last is decoder.

3 Experiments

3.1 Overview of Data

The image colorization dataset from Kaggle [4] contains 25000 images of assorted landscapes along with their grayscale counterparts. The dataset contained generalized images of everyday objects, such as humans, lakes, and skyscrapers. The input resolution of all of the images is 224x224.The Kaggle dataset came in two types of numpy arrays, l.npy, the array for lightness channel for the images, and ab.npy, the array for the corresponding a and b color planes. The Kaggle dataset can be fed directly into each model since it was already in numpy arrays.

Figure 5: Sample Image from Kaggle Dataset

We also collected images to augment our data. The first custom dataset that we compiled was a set of 500 high resolution images of flowers, of varying colors, shapes and sizes taken on an iPhone 8 Plus. The colors of the flowers were mainly red, yellow and white, with a green background of leaves. The second dataset contains 700 high resolution images of the Beach and the Sun for the dataset. The images are extremely diverse in color composition, including the sun, beach sand, ocean, and other characteristics. Both datasets needed to be further preprocessed before being fed into the model.

Figure 6: Sample Images from Beach & Flower Dataset

3.2 Data Preprocessing

The first step into training our models was to preprocess the image sets into numpy arrays (if they were not already). Our input data was fed in as high-resolution images, such as the images in the custom *Beach* dataset shown in the figure below.

Figure 7: Original Unmodified Beach Image (4032x3024 px)

Using OpenCV's resize function in python, all the images are first down scaled to a resolution of 224x224 pixels. The images are downscaled first is because this significantly reduces computation time, since the amount of pixels to compute over is reduced by N^2 (N denoting the difference between original and modified width). This downscaling of the set would be useful later as it goes through the next set of preprocessing steps.

Figure 8: Downscaled Beach Image (224x224 px)

To follow up, the image set is then feed through Image Segmentation using Matlab's superpixels algorithm [5]. An image can be visualized as a 2-Dimensional Array of pixels, each containing 3 8-bit color channels in the RGB plane. The segmentation in Matlab uses a processes that is called 'Simple Linear Iterative Clustering' (also known as SLIC) to group the pixels with similar RGB values to a larger subgroup known as a superpixel. A visualization of how groups of pixels are clustered are shown in the figure below (where the white border is the boundary mask of the superpixel).

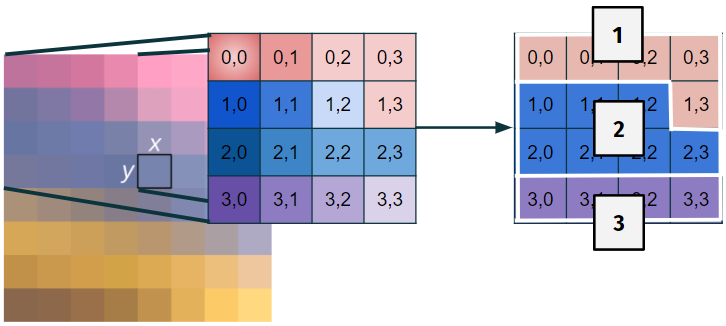


Figure 9: Overview of Superpixel Clustering

In terms of graph representation, each individual pixel is a node of a superpixel; a subgraph clustered with nodes of similar RGB values. On a higher level, the collection of superpixels forms the entire graph which is the input image. The superpixels is then colored with their individual average RGB value to supplement as uniform weights when it comes to setting up the neural network for our training. This method has shown to be quite successful in recolorizing images [6].

Figure 10: Segmented Beach Image (224x224 px)

After the image set has been segmented into superpixels, it needs to be converted to a format where we can properly feed it in as weights to our training model. To do this, we must convert the segmented images to the LAB color format, where L is the lightness channel of the image (denoting intensity), and ab denoting the green-magenta and blue-yellow color balance planes respectively. The L channel would be used to be fed it as the training set to the neural network, and the ab channel would be used as validation for the neural network. For the conversion, we use OpenCV's *rgb2lab* function to break down the image to the L and ab channels and then stored them into individual numpy arrays to feed into the training model (as visualized in the figure below).

Figure 11: The L and ab channels of the Beach Image

3.3 Training

We utilized the Keras library of Tensorflow in order to compile our model in python. We split the training data into halves to generate a validation set, to better adjust hyperparameters of the model. Both models were trained using an Nvidia RTX 3080 with 10GB of VRAM. ADAM was utilized for both the GAN and the VAE as the optimizer. Our models and test scripts can be accessed at [7].

3.4 Grading Algorithms

For this project we need a standard of accuracy to depict whether the colorization is somewhat probable. There are two main metrics on the success of our models' predictions which are quantitative and qualitative predictions. For a grading algorithm we mainly focused on the quantitative aspects such as direct accurate depictions rather than just plausible color. By doing this, we will compare the predicted color image output with the actual colored image. For this approach we constructed 3 grading algorithms, Max Color Channel Accuracy, Backtrack Gray Scale accuracy, and Backtrack OpenCV Gray Scale accuracy.

Max Color Channel

Quantifying based on Max Color Channel is fairly straight forward. This first step for all the algorithms was to segment the image creating super pixels and storing the rgb value as the average from all the surrounding pixels. With this we can iterate pixel by pixel and enumerate the amount of Red, Green, and Blue pixels based on the max color channel. We do the same on a 1 to 1 scale with the predicted image and compare if the max channels were the same. This can then give us an accuracy metric on the predicted max color channel.

Grayscale and OpenCV Grayscale Algorithm

These two algorithms use the same procedure but with two different Gray Scale Formulas. This algorithm reverts both photos using 2 gray scale formulas and then compares the pixel value with a given threshold. The first gray scale formula is the average of each pixel shown below.

$$GrayPixel = (Channel[R] + Channel[G] + Channel[B])/3$$

With this formula we calculate each Gray Pixel and compare it with the predicted Gray Pixels within the given threshold range.

$$(GrayPixel - Threshold \leq Predicted \leq GrayPixel + Threshold)$$

With this we can also get a range of accurate backtracked pixels. The process is the same for the OpenCV grayscale Algorithm yet the formula changes to the one shown below.

$$GrayPixel = .30 * Channel[R] + .59 * Channel[G] + .11 * Channel[B]$$

By quantifying max color channel it gives an estimation on the accuracy which the model can predict color channels. With this information we can see which channel the model has the highest accuracy and which channel prediction needs improvement. Then with this information we can train the model with more needed pixels types. Due to our models downscale, rotations, and upscale it is important that the photo is not staying with in the probable range of a backtracked gray scale image. As in the original fed image is similar to the produced output.

3.5 Results

Using the grading algorithm we can compare the accuracy of all three different models. To start off the Richard Zhang model had the greatest form of accuracy. This model trained by their own data set was able to generate our test model with 91% max color accuracy. It also had an 89% OpenCV grayscale comparison. We then experienced with the GAN model and different epochs steps. For the time being we got a 91% grayscale OpenCV accuracy with threshold at 25 epochs. Unfortunately, the more epochs we would try, the more noise would be generated. Lastly, the VAE did not have a lot of success with Max Color Probability. This was due to the desaturated effect generated from the convolutions and up scaling layers. The image generated still on the bright side had a 71% OpenCV to grayscale accuracy.

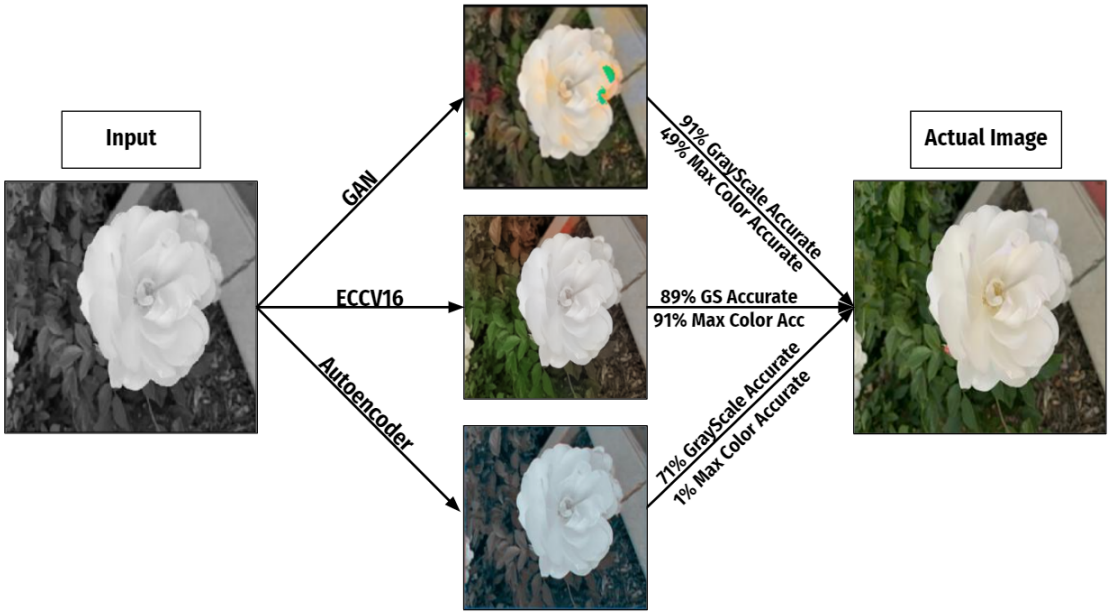


Figure 12: Grading Probabilities on a White Rose

4 Conclusion

For Image Colorization, we found that our approach (the GAN in particular) provided significant advantages over other contemporary solutions to the problem. A significant advantage in our method was the extreme accuracy and plausibility achieved with relatively small training data, without the need to modify another model for feature extraction. In addition, our preprocessing was relatively simple, and we did not need to partition the data in a sophisticated way as in approaches such as [1]. The VAE model produces outputs that look more aged and desaturated, giving them less plausibility to be coinciding with what is believed to be a modern-day colorized image.

Some of the possible future directions this project would be in the works of colorizing videos, improving the GAN model with denoising images via a denoising autoencoder (where pure Gaussian noise is added to the inputs of the autoencoder) , and utilizing different parameters such as 3D cloud point data to colorize LIDAR images for light detection. A video is a collection of images (each is a 2D array of pixels) that are played in sequence. An additional weighting factor for a GAN could be color gradients with respect to time between each individual pixel. This time dependence in correlating similar frames can produce more uniform coloring across a static scene. There is an ongoing open source project called *DeOldify* [8] which further explores this work of video colorization with GANs giving extremely promising results.

References

[1] Richard Zhang, Phillip Isola, and Alexei A. Efros. *Colorful Image Colorization*. 2016. arXiv: 1603.08511 [cs.CV].

[2] Anat Levin, Dani Lischinski, and Yair Weiss. "Colorization Using Optimization". In: *ACM Trans. Graph.* 23.3 (Aug. 2004), pp. 689–694. ISSN: 0730-0301. DOI: 10.1145/1015706.1015780. URL: https://doi.org/10.1145/1015706.1015780.

[3] Aurelien Geron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. 2nd. O'Reilly Media, Inc., 2019. ISBN: 1492032646.

[4] Shravankumar Shetty. *Image colorization*. Oct. 2018. URL: https://www.kaggle.com/shravankumar9892/image-colorization.

[5] *Superpixels*. URL: https://www.mathworks.com/help/images/ref/superpixels.html.

[6] Radhakrishna Achanta et al. "SLIC Superpixels Compared to State-of-the-Art Superpixel Methods". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 34.11 (2012), pp. 2274–2282. DOI: 10.1109/TPAMI.2012.120.

[7] Timothy Do et al. *Colorization Using Unsupervised Methods*. URL: https://github.com/dotimothy/EECS195-Colorization.

[8] URL: https://deoldify.ai/.