CS 520: Assignment 4 - Colorization

Aditya Vyas
Vedant Choudhary
Nitin Reddy
Siddharth Sundararajan

May 13, 2019

The purpose of this assignment is to demonstrate and explore some basic techniques in supervised learning and computer vision.

1 Representing the Process

As each image is a collection of pixels, in our approach we have extracted the pixels from the image (for each color) and stored it as a separate numpy array. Note that our input is a set of colored images. Therefore, each image will give us three numpy arrays (one array for each color - Red, Green and Blue). The mapping takes place between the gray scale and the RGB scale. Each color has a scale from 0 to 255.

As the input images are colored, we use the following formula to extract the corresponding gray values for each pixel.

$$\text{Gray(r, g, b)} = 0.21r + 0.72g + 0.07b$$

where, r, g and b correspond to the corresponding red, green and blue pixel values.

With the above formula, the input array for one image is stored a numpy array, where each value is the gray pixel value (traversed from left to right and top to bottom). The output is a set of three arrays, one for each of RGB colors. For example, a pixel in the image will have the following input to output mapping.

$$\begin{bmatrix} X & | & & Y & \\ Gray & | & R & G & B \\ 154 & | & 105 & 51 & 67 \end{bmatrix}$$

It is difficult to capture good amount of information with the above mapping, i.e., a single grayscale pixel value to a corresponding color (RGB) on a pixel by pixel basis. To overcome this problem, in our approach the concept of filter is used. Filters help capture the information surrounding a pixel. Note that when using filters it is also important to choose the filter size. Filters help capture more information in the input before mapping it to the output. For example, if a filter size of $3 * 3$ is taken, then there is a $9 : 1$ input to output mapping (from gray to one color(Red Green or Blue)). An instance for one pixel is seen below.

$$\begin{bmatrix} X1 & X2 & X3 & X4 & X5 & X6 & X7 & X8 & X9 & | & Y \\ 121 & 135 & 174 & 221 & 109 & 64 & 52 & 241 & 97 & | & 159 \end{bmatrix}$$

where X1 to X9 is the input vector mapped to Y, the output scalar (which can be the value for Red, Green or Blue). In our case, filter size was chosen based on the size of the image taken. If the size of the image taken was $300 * 300$, then the filter sizes taken were between $5 * 5$ to $9 * 9$. Note that it is important to capture the right amount of information, not too much or too less, as it will help the model being built to learn well.

As filters capture the information around a pixel, the size of the input mapping for corner cases are not the same as other cases. To overcome this, zero padding is done. For example, if the size of the image is $6 * 6$ and stride is 1, after zero padding, the dimension size becomes $8 * 8$. On this new image dimension if a $3 * 3$ filter size is applied, the output is a $6 * 6$ image (which is expected). The formulae used for zero padding is as follows:

$$O = \frac{W - K + 2P}{S} + 1$$

where O is output size, W is the input size, K is filter size, P is padding and S is stride. Note that in this work, stride is taken as 1, output size and input size are the same and zero padding is done based on the image size.

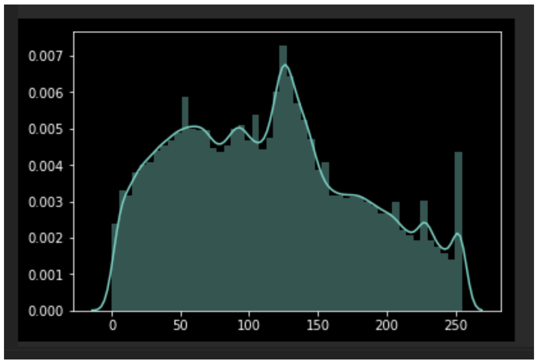
2 Data

For this work we selected ten images from Google Images. In Figure 1 we see that the sizes of the images taken for the training data are different. A padding of black was given to images that were rectangular in dimension. Images that had a square dimension had no padding as it was already uniform. These images were selected as the distribution of the RGB components was uniform. The distribution of the RGB components are seen in Figure 2.

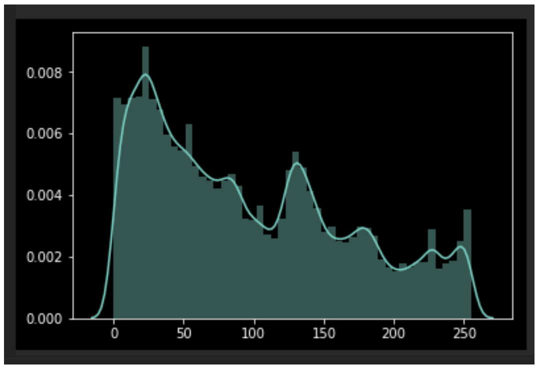
(a) Scene 1

(b) Scene 2

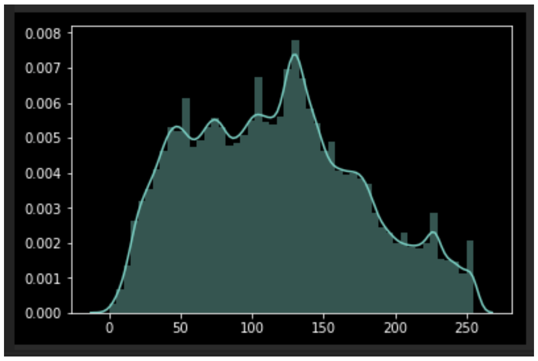
Figure 1: Sample data from training data



(a) Distribution of the color Red



(b) Distribution of the color Green



(c) Distribution of the color Blue

Figure 2: Distribution of RGB components for Training Data

Three pre-processing steps are performed before building the model:

- First, as all the image dimensions are different, the images are shrunk to a constant dimension size. In our work, few dimension sizes that were considered are $100 * 100$, $300 * 300$, $500 * 500$ etc. Note that the sizes of the images were reduced as it would be computationally expensive to consider the original dimension size.

- Second, the training data (understood by the computer - numpy array) was formed by converting the RGB components to Grayscale components (using the formulae mentioned earlier).

- And finally, zero padding was done on the gray scale pixel values as per the process mentioned earlier.

3 Evaluating the Model

In this problem, a model is said to be perfect when it is able to completely convert a gray scale image to a color image. In our work, the model built is able to re-create most of the colors. In Figure 3, we see the original images of the test dataset. In Figure 4, we see the output images of the coloriser model we build. Note that all the images that are given by the model is of size $100 * 100$. When the images from the two figures (3 and 4) are compared, it is evident that our model is able to distinguish colors well. Although the colors may not be perfect, it is able to very well distinguish that there are different colors for each object (grass, train, trees, etc) present in the image. A very good example would be the famous Van Gogh painting (the second image present on the top right corner of figures 3 and 4). Our model is able to almost replicate it well. This is an example for an almost perfect conversion.

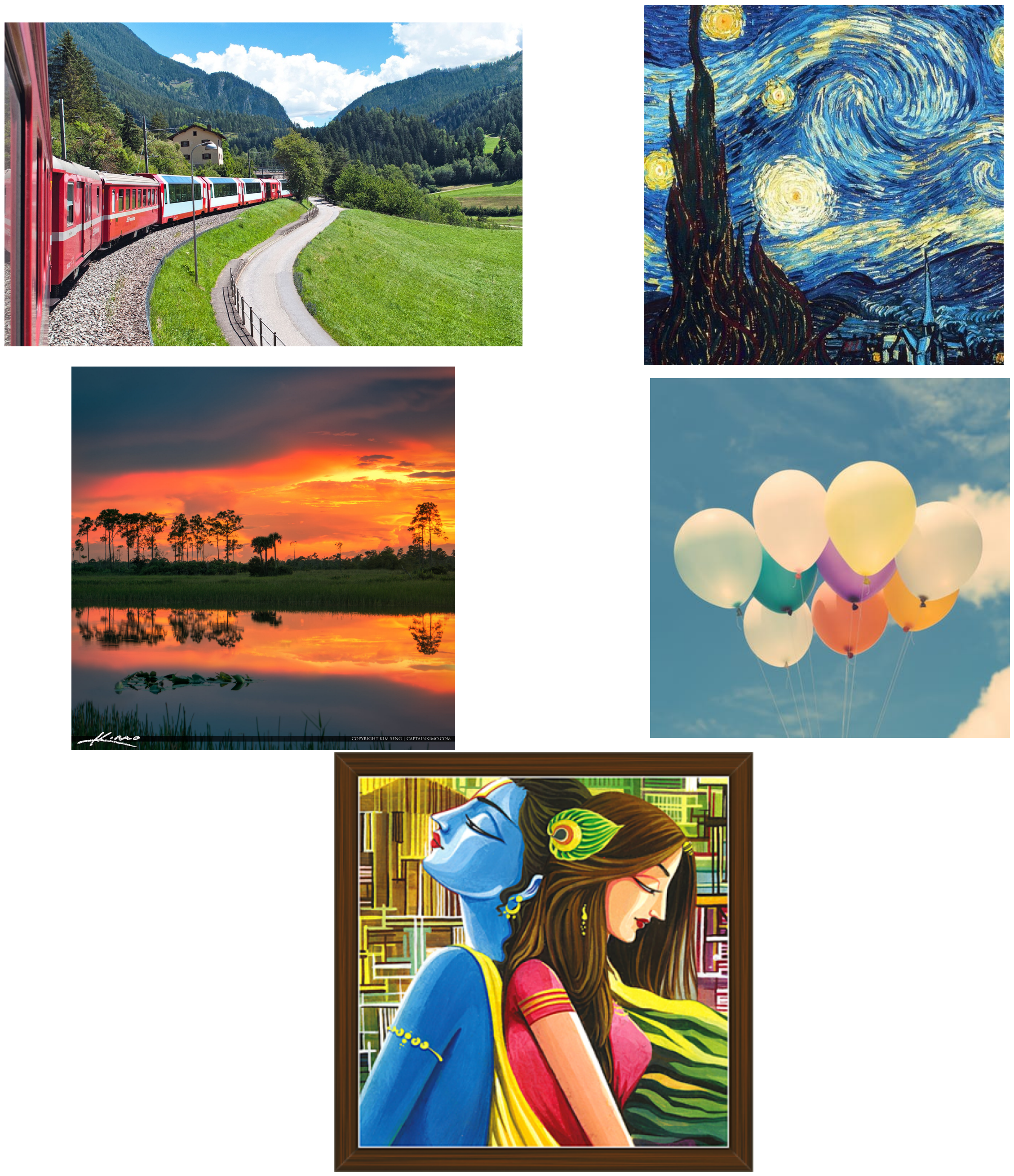


Figure 3: Original Images

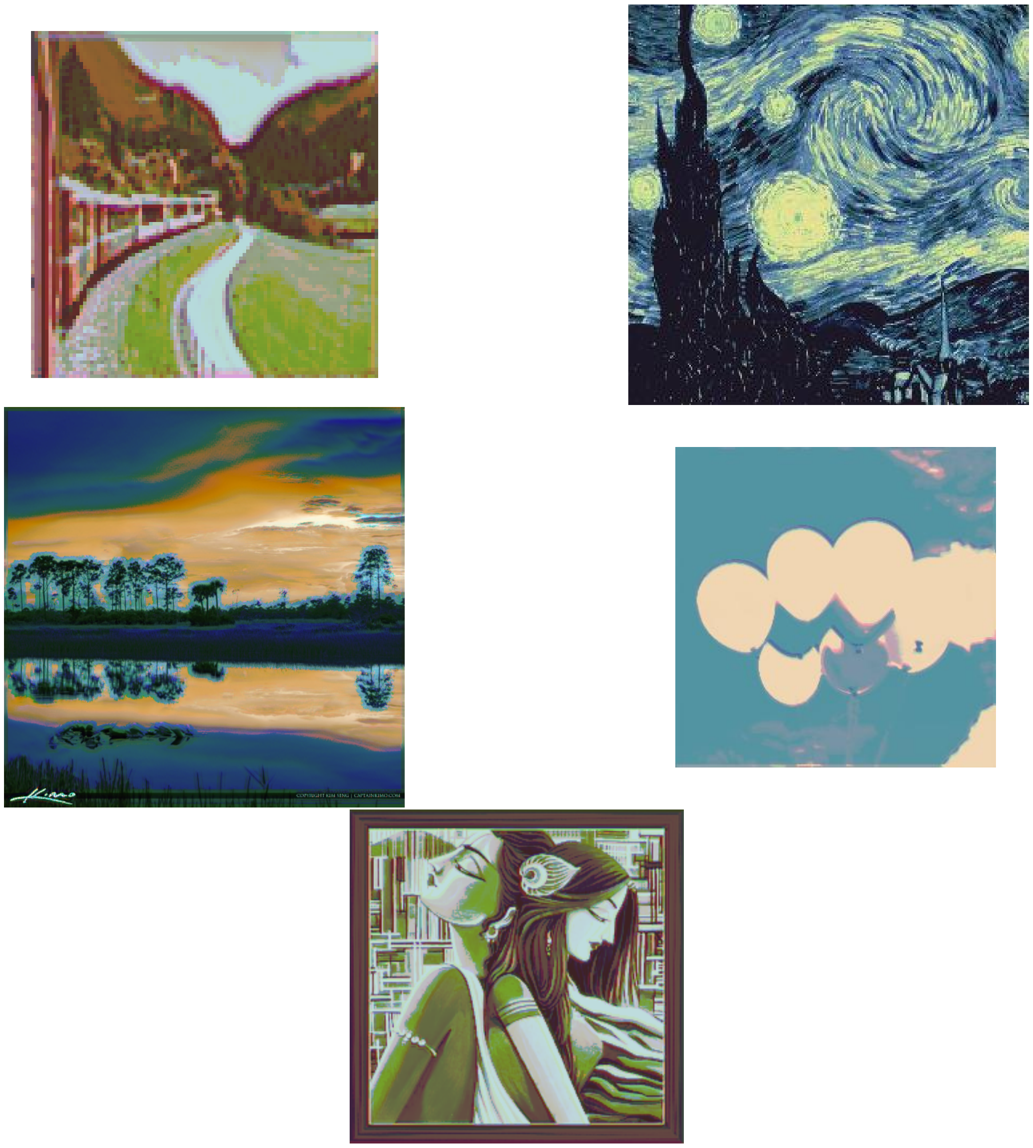


Figure 4: Colorised Image Outputs from the model

To evaluate how well our model is performing, Mean Squared Error or MSE is used. This is a useful evaluation metric as our model learns iteratively over time. Note that MSE is the numerical/quantified way of representing the error. As the problem we are dealing with is based on images, it is good to understand the perceptual error (how good do humans find the result of the program). For example, if our model colored the grass purple, the perceptual error would be high a human knows that the color of grass cannot be purple. Similarly, if our model colored the train blue, the perceptual error will be low as a train can be blue in color. Note that this is with the assumption that the human has not seen the original image. In our case, from figure 4 we see that the perceptual error is minimal.

4 Training the Model

In our work, the model built is a dense (fully connected) feed forward neural network model. Initially, we started with 5 hidden layers with each layer having 20 nodes. This was our starting point because the idea was to capture enough information (different shades of gray on shapes) at every node. After trial and error, it was found that 3 hidden layers with 10-20-10 nodes was more than enough to capture the required information. To train our model in a computationally tractable manner, we used the backpropagation algorithm with stochastic gradient descent method to find the optimum solution. This optimization technique did not give any useful results. Finally, we used backpropagation with mini-batch gradient descent method.

Two methods were used to determine convergence. First, as a starting point, number of epochs were set to 30, which was more than enough to find an optimum solution. Second, the tolerance was set to $1e-5$ to determine convergence. The latter method served better and hence was used.

Along with the train data, a small fraction of data was used in the model as the validation. This data was not used in the training process of determining weights. After every epoch, error on the validation data was determined. If the error on the validation data did not improve over two consecutive epochs by a tolerance factor (as mentioned previously), training would stop. This helped in stopping the model from over-fitting or learning the weights very specific to training data.

5 Assessing the Final Project

- **How good is your final program, and how can you determine that?**

  Our model performs decently - given grayscale images, it is able to identify and color objects to distinguish them from each other. For eg. It will give a shade of blue to skies, shades of green to grass and trees and so on. Being able to distinguish objects by different colors is a metric we used to identify the goodness of our model.

- **How did you validate it?**

  We validated our model by using lots of images downloaded from Google. It was obvious that the model will not produce an exact matching colorisation, so we observed the colors of different objects in the images. If the model assigns same color to multiple objects, it means it is not able to identify boundaries between objects. From the above image examples, we can see that the model is indeed able to colorise the images properly, however, sometimes it is prone to errors because of dominance of a particular color in the training data.

- **What is your program good at?**

  Our model is good at distinguishing different objects and coloring them differently. Further, in many images, it is able to assign a shade/hue of the original color of the object. For example, in the image of the train and mountains above, it assigned white color to the clouds, shade of green to the grass, a shade of red to the train and blue to the sky.

- **What could use improvement?**

  Our model is prone to changes in the training data. The type of colorisation depends a lot on the distribution of colors in our training data. During the initial testing, all the training images had a lot of green color in them due to which our model became very easily biased towards green and started giving a greenish hue to the test images. Hence, a proper selection of training data images with an almost equal distribution of colors can improve our model a lot.

- **Do your program's mistakes 'make sense'?**

  Our models mistakes makes sense. One mistake our model made a lot was to become biased towards a particular color and assigning a shade of that color to the test images. This can be solved by using a properly curated training data with almost similar distribution of colors. Another mistake our model made was to assign same colors to far off objects which are appearing very small in the image. This is because it is not able to distinguish the smaller object pixels.

- **What happens if you try to color images unlike anything the program was trained on?**

  Our model depends a lot on the color distribution of training images. Hence, it will try to color the unseen images on using whatever knowledge it has gained from training. If the images contain sky, grass and other such common items, then it correctly assigns them colors but colors not in the training set are not colorised correctly.

- **What kind of models and approaches, potential improvements and fixes, might you consider if you had more time and resources?**

  We used a very simple neural network to colorise images - a feedforward neural network with just 3 layers. Although, it was able to give decent results, we think that the accuracy can be improved further by using the following

  - Convolutional Neural Networks (CNN): CNNs capture the underlying patterns in an image perfectly and thus are able to identify and learn the color distributions of images more properly than a feedforward network.
  - Variational Autoencoder (VAE): A VAE can be used to learn the underlying distribution of the images and generate more images from this distribution. It is a method of generating synthetic data and hence can be used to generate images with different colors.