CS440 Project 4: Colorization

Parth Patel (psp116)

May 2021

Introduction

Implementation of basic agent and advanced agent to colorize a given gray scale image based on a model.

Language used: *python*
Libraries required: *random*, *numpy*, *math*, *matplotlib*

Basic Agent

- **Input Space**

  Both the agents follow the same idea of taking the image and dividing it into two parts; The left hand side is used as training data and the right is used as testing data. These images are basically a matrix of 3-component vectors, where:

  ```
  Image[x][y] = (r, g, b)
  ```

  The next thing we do is convert both the sides into its gray scale equivalent using this formula:

  ```
  Grayscale(r, g, b) = 0.21r + 0.72g + 0.07b
  ```

Having generated these two matrices, the input that the basic agents accepts is as follows: for every pixel in the gray scale testing data that is to be colorized, a patch of 9 pixels (which includes the 8 neighboring pixels around it and the concerning middle pixel) is provided and the agent yields a respective (r, g, b) equivalent. This provides the agent with more information in order to make smarter choices. Each input vector can visualized as such:

```
[g1, g2, g3, g4, ... , g9]
```

- **k-means Clustering**

  In order to make the whole process of colorizing easier, k-means clustering is used to yield the 5 most representative colors in the image. The following method is used:

  ```
  def kmeans(image, clusters=5)
  ```

  This methods works by randomly choosing 5 colors and then each point in the dataset is clusterd depending on whichever representative color is *closest* to that data point/pixel. After that the clusters averages are generated again by averaging each cluster and the process repeats. This algorithm ends once there is not much difference between the old and the new cluster averages. Once we have the rgb values of the 5 most representative colors(cluster averages), the image is manipulated to be represented in terms of those 5 most representative colors. This is achieved by checking which representative color is closest to a given pixel. Mathematically, each representative color is subtracted from the pixel and the one with the smallest norm is chosen.

- **Colorizing**

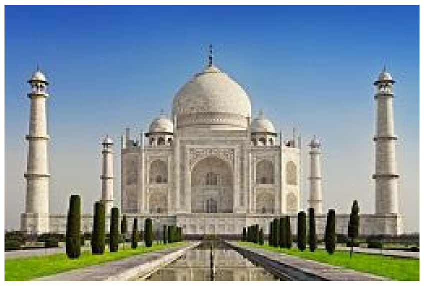
  Once the agent receives the 5-color representative version and the grayscale version of the image, it begins the colorizing process. For each pixel and its associated 9 pixel grayscale vector/patch in the test set, it find the 6 closest vectors/patches in the training set based on the norm of the difference. This is accomplished by first creating the 9 pixel vector for each pixel in the image and followed by creating another array of distances from the patch using *np.linalg.norm*. The last thing it does is sort the distance vector and returns the indices of the six pixels that had the smallest norm. The following method is used for finding patches:

```
def find_patch(patch, img)
```

For each of these patches, get the color representative of the middle pixel. If there is a majority representative color, it is selected. Otherwise, the representative color of the closest patch is selected. The selected color is used to recolor. This technique leaves us with a 1-pixel wide band since the edge pixels have less than eight pixels surrounding it.

- **Output and Results**

  Following this procedure, we get a picture whose left part has been colored in its 5 most representative colors and the right has been colored based off the left. The basic agent performs adequately when the picture is symmetrical since it relies on comparison of patches in order to decide upon a color. Here are some results:


(a) Original image one

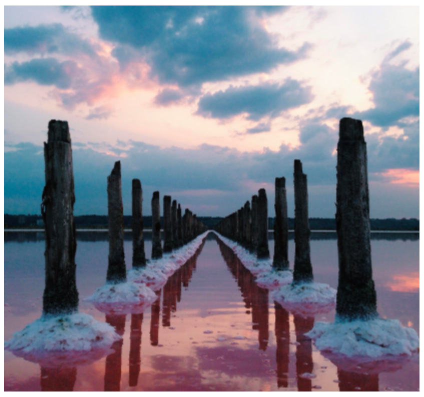
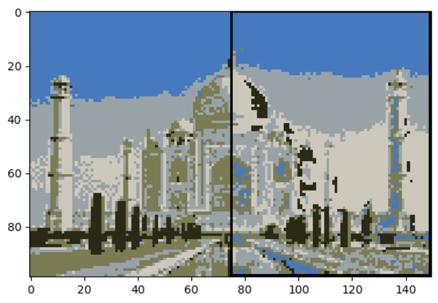
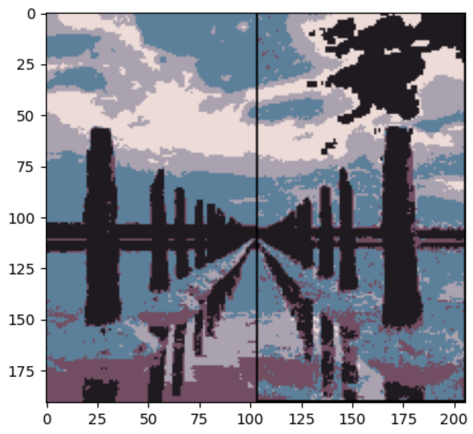

(b) Original image 2

Figure 1: Original images



(a) Result image one



(b) Result image 2

Figure 2: Result images

Visually speaking, it does a fair job in recoloring the training data. It is almost able to make every object distinguishable. Colors can be inconsistent due to reasons such as there was a shadow on the object. Since this agent "puts" each pixel into one of the five buckets of representative colors, I have calculated its error percentage by comparing the agent re-colorized training data and actual re-colorized training data. For original image 2, mathematically, it wrongly classified around $25,000$ pixels out of around $80,000$ which gives it a 70% accuracy rate.

Advanced Agent

- **Acronym**

  The acronym that I have chosen for this agent is **RGB** (*R*egression and *G*radient *B*ased agent).

- **Input Space**

  Much like the basic agent, the advanced agent utilizes patches as its input. A given image is split into two parts: the left is used as the training set and the right is used as the testing set. The training set, as the name suggests, is used to train and calibrate the model. Firstly, the training image's grayscale counterpart is created using the same equation that was used in the basic agent. Now, the grayscale matrix has the same shape as the image (lets say $m * n$). In order to make handling the code easier, the grayscale is reshaped into a linear matrix. The next thing we do is create the patches vector for each grayscale pixel just like we did in the basic agent. This is accomplished by the *neighbors* method. A visualization the input space is as follows:

```
[[g11, g12, g13, ... , g19]
 [g21, g22, g23, ... , g29]
 ...
 [gn1, gn2, gn3, ... , gn9]]
```

  The another part of the input space is the actual value of one of the 3 color components i.e. Red, Green and Blue, which would be used for training. Also, each component has its own model where the weights differ but the structure is identical. So for a given Red model and input, our goal is:

```
[g1, g2, g3, g4, ... , g9] ->  r
```

- **Preprocessing data**

  In order to make my life easier and increase the accuracy of the model, the input space is preprocessed. Firstly, all the elements of the input data are divided by 255 since grayscale has range of 0 to 255. This allows us to avoid overflow errors. Secondly, the data is re-centered meaning the average/mean of the input data is subtracted from each element. Lastly, we add "features" to our data. Given the fact that RGB is a quadratic sigmoid model, there are going to be 55 features for each 9-dimensional input vector. To explain this better, this is an example of fitting quadratic features on a 3-dimensional vector:

  ```
  vector: [a, b ,c]
  features: [1, a^2, b^2, c^2, ab, ac, bc]
  ```

  Moreover, we preprocess the output vectors too. This is not to be misunderstood with the resultant scalar from the model but, rather, the actual output vector derived from the image that we use for our loss function. We divide the entire output data with 255. We do this because the resultant scalar would be between 0 and 1.

- **Model**

  RGB utilizes a sigmoid quadratic model where the sigmoid function can be defined as follows:

  $$\sigma(x) = \frac{1}{1 + e^{-x}}$$

  The sigmoid function has a range of [0, 1]. The idea is to add weights to our input vector and pass it to the sigmoid function to produce a value between 0 and 1. So our model is:

  $$R(\vec{x}) = \sigma(\vec{w}.\vec{x}) = y$$

- **Loss Function**

  The loss function that RGB utilizes is as follows:

  $$Loss(\vec{x}, y) = (R(\vec{x}) - y)^2$$

  where x is the input vector of a given pixel and y is the red, green or blue component associated with that pixel.

- **Gradient**

  Now that we have all the components for our model together, RGB needs a loss gradient to run the stochastic gradient descent.

  $$d(Loss(\vec{x}, y))/d(w_j) \tag{1}$$

  $$\begin{aligned} &d((y - \sigma(R(\vec{x})))^2)/d(w_j) \\ &= 2.(y - \sigma(R(\vec{x}))).d(y - \sigma(R(\vec{x})))/d(w_j) \\ &= 2.(y - \sigma(R(\vec{x}))). - \sigma'(R(\vec{x})).R'(\vec{x}) \\ &2 * (\sigma(R(\vec{x})) - y) * R(\vec{x}) * (1 - R(\vec{x})) * \vec{x_j} \\ &= 2 * (\sigma(R(\vec{x})) - y) * R(\vec{x}) * (1 - R(\vec{x})) * \vec{x} \end{aligned} \tag{2}$$

  Note that (2) has been extended to all components of the vector. This is what we use for updating our weights along with a learning vector α and random input vector x_i:

  $$\vec{w_{t+1}} = \vec{w_t} - \alpha * gradient(Loss(\vec{x_i}, y))$$

- **The Process**

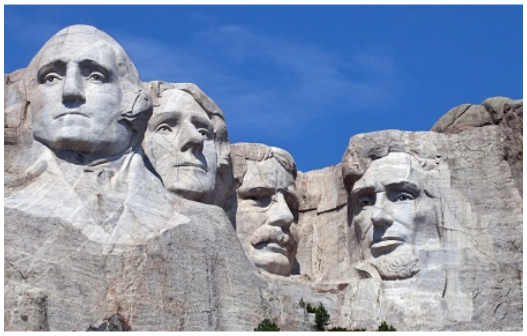
  In RGB, the gradient descent algorithm keeps running until the square root of the Mean Squared Error between the model generated output and the actual output hits a certain threshold. This means that the threshold value has to be tweaked for each model and each image. This greatly depends on the distribution of image as well. I will discuss this more in the next section but the idea is that more the variance, the lesser room for generalization. To actually figure out the threshold, I run the agent once with a threshold of 0.01 and notice the value for which the convergence starts happening which translates to the square root of the mean squared error decreasing very slowly. This also indicates that we have started approaching the minimum.

Once RGB has produced a model for each color component, it moves on to colorizing the testing data which is the right side of the image. The agent does not consider the pixels on the edges to train but for testing, it uses padding to create a patch. So, for the testing dataa:

1. The grayscale value is calculated.
2. The grayscale values are centered on mean and normalized on 255.
3. The patch vector is created. If on the edge, the grayscale value of the pixel is used to fill in the missing values.
4. For each component, the patch vector is multiplied by the weight and passed into the sigmoid function (as per RGB's model).
5. The resultant value multiplied by 255 yields the component color.

- **Output and Results**

  RGB is able to produce fair results. The model is able to make each object in the picture distinguishable but the colors are not as accurate. Here are some results:


(a) Original image one

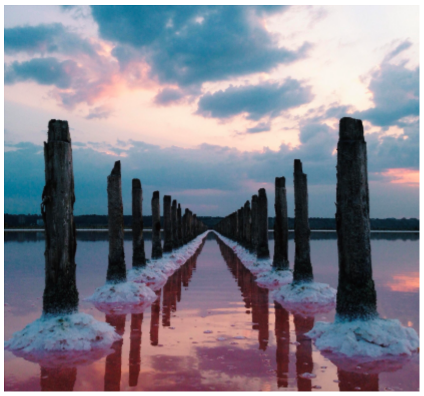
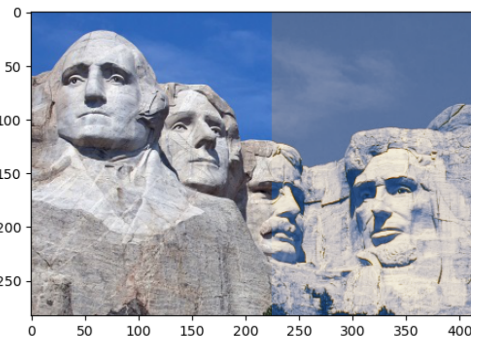
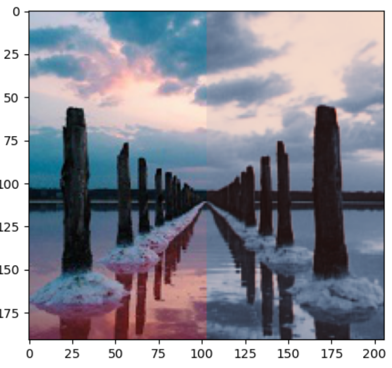

(b) Original image 2

Figure 3: Original images



(a) Result image one



(b) Result image 2

Figure 4: Result images

The accuracy of colors depended on the distribution of the component colors. Let's talk about original image 2. You can observe in the original picture that red has the largest variance. Naturally, in the resultant picture, the sky has almost the right concentration of red but the water seems dull. Similarly, result image one suffers from not having the proper concentration of blue. Mathematically, I took the square root of the mean squared error of each color component to see the average value that the agent is off by. For result image one, the red was off by around 24 while the green was off by about 13 and the blue was off by around 30. These values are in line with my earlier claim since blue has the biggest mean error.

Questions and Answers

- **How did I choose the parameters for my model?**

  I started with a linear sigmoid model but my results were a little blurry and in order to make object detection better, I upgraded to a quadratic sigmoid model. This did not make a considerable difference but was it was definitely better than the linear sigmoid model at edge detection. When it came to the weights, they were random values but I observed that the closer they were to zero, the better results I got. This made sense since having high initial weights would be harder to scale. Naturally, I chose the weights to range between -0.001 and 0.001.

- **How did I handle training while avoiding overfitting?**

  This was the hardest part of the project. Generalizing the alphas and the terminating condition was very specific to a given training dataset. For example, in original image 2 the distribution of red is the most variable in comparison to blue and green. Consequently, training the red model was the hardest since more variance equals to less room for generalization. While training my models for this image, I was able to push the mean squared error to be less than 0.05 for green and blue model but for red, it was tough pushing past 0.09 even after 2 days of training. Consequently, the image produced by the agent lacked the proper shades of color red. To avoid overfitting, as explained in the section The Process, I do a sample run of the agent and notice at what value does the error starts converging.

- **How does RGB hold against the basic agent?**

  It is difficult to directly compare RGB with the basic agent since the former is a regression problem while the latter is a classification problem. Nonetheless, an evaluation of each agent in its respective principle should give us an intuitive sense of their performance. I would be using examples I used in both the agent's Results and Output section.
  For the basic agent, I said that the agent classified the pixel correctly 70% of the time for original image 2 but if we quantify the actual difference(square root of the mean squared error) between the individual color component values, this is what we get: 96.25 for red, 104.09 for green and 102.81 for blue.
  Now, with RGB on the same image(original image 2) with the same error standards: 25.61 for red, 15.35 for green and 16.22 for blue. Clearly, RGB stands out better, but, they are both very different problems, doing very different things.

- **How would I improve my model with sufficient time, energy and resources?**

  For each picture that I generated through the advanced agent, it took an average of 2 days to compute. If given sufficient resources, I would be able to train my model better in less amount of time. As mentioned before, the colors which have a big variance would not be as big of a problem with such resources.

Academic integrity

- I have read and abided by the rules laid out in the assignment prompt.
- I have not used anyone else's work for my project, my work is only mine.

  Signed by: Parth Patel