CS 520 - Colorization Report

Srija Gottiparthi (sg1416)

December 11th, 2020

Abstract

The following paper discusses my findings and analysis of the Colorization problem. I will briefly discuss my implementations for the Basic and Improved Agents, and finally showcase and discuss the results in detail. For this implementation, I coded in Python 3.

1 Introduction

The objective of this project was to recolor the right half of a grayscale image, using the left half of the image as training data. There were many approaches to this problem, but the given algorithm for a Basic Agent involved using k-means clustering to find the five most representative colors in the photo, and then recoloring the right half of the image patch by patch. The overall goal was to learn more about AI, machine learning, and computer vision, and to improve upon the Basic Agent's performance by implementing basic techniques for supervised learning.

2 Agents

This section will discuss the implementation of each agent, the results that each agent came up with, and analysis of these results, including details about parameters, data pre-processing, training, and quality evaluation.

2.1 Basic Agent (`basic_agent.py`)

The basic agent's algorithm was outlined in the project description: recoloring the photo using the 5 most represented colors from k-means, then determining similarity between 3x3 pixel patches in the test data and training data to determine what color should be assigned to the pixel.

The libraries I used were NumPy to make array and math operations easier, Scikit-Image for image manipulation, Matplotlib to display the final results, and the KDTree functions from Scikit-Learn to do nearest neighbor search. Below, I've outlined the Basic Agent, and its helper functions.

- **Basic Agent** (`agent`)
  I decided what color a pixel should be given by looking at its surrounding 3x3 patch, and taking average patch color & distance from the target pixel into account when comparing to the training data.

  First, I generated a list of average colors from all the patches in the grayscale training data, and extracted just the pixel coordinates of each patch as a NumPy array to make processing easier. At this point, I initialized the KDTree to be used later on as well. Then, I started looking at every pixel in the test data.

For each pixel, I compiled a list of pixels whose patch's color distance was less than a specified value (I experimented with this, and included a few lines of code ensuring that if there weren't any pixels with a close enough color distance, the agent would look in a larger "radius" for similar patches). The initialized KDTree was then fit to this list of patches, and returned the 6 closest patches by distance. My reasoning for this was that if two patches were already of a similar color, then if they were also closer together it was more likely that they would be the same color. With this list of 6 similar patches (both by color and by location), I extracted the color from the center pixel in each patch, and assigned the majority color to the target pixel in the test data.

- **Getting Average Patch Colors (`get_avg_color` and `get_avg_bw`)**
  These functions look at all of a pixel's surrounding pixels, and average their RGB values. There are two different ones for color and grayscale because SK-Image's rgb2gray function converts RGB values into single numbers on a scale from 0 to 1 representing the amount of gray, so `get_avg_bw` is slightly modified to account for this.

- **Recoloring the Photo (`five_color`)**
  This function recolors each pixel using one of the five representative colors that is closest to it.

- **Getting Color Distance (`color_distance`):**
  This function gets the distance between two RGB colors. I had to research this a bit; the formula I ultimately ended up using is credited in my code comments.

- **K-Means Algorithm (`k_means.py`)**
  For the purpose of this project, I wrote a K-Means clustering algorithm from scratch in `k_means.py`, specifically to be used on image files. My implementation made use of the Python Imaging Library (PIL) to open and read images. Since I was not familiar with image manipulation in Python at the start of this project, I ended up creating `Point` and `Cluster` classes. Points store the coordinates of each pixel, and Clusters store the center of a cluster & the points contained in that cluster. I also wrote two helper functions: `convert_to_points` converts every pixel in an image to a Point, and `euclidean_dist` finds the Euclidean distance between two Point objects.

  The main `KMeans` class is where all the action happens. The `find_center` function finds the center of a set of Points by adding up all their values and dividing by the number of Points. `assign_to_cluster` and `fit` do the work of putting Points in Clusters, and then re-assigning these Points to new Clusters until the Cluster centers stop changing; this indicates that the data is sufficiently clustered.

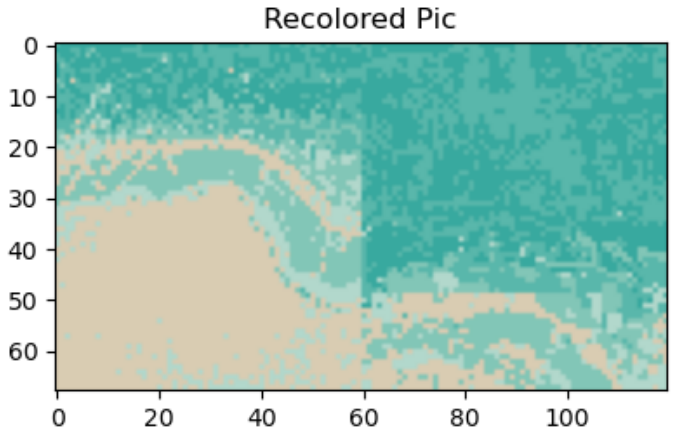
  Within the `get_colors` function, which acts like the "main" function, the image is opened, and all its pixels are converted to Point objects so that they are more easily stored in Clusters. Then, the K-Means fit function is run, given the number of desired clusters (5 for this project). Finally, after some cleaning up of the output from the fit function, a list of the 5 most representative colors in the given image is returned.

- **Quantitative Difference (`difference`)**
  I implemented a function that calculates the difference between two sets of pixels. It simply returns the ratio of incorrect pixels (as determined by the recolored photo) to total pixels in the photo. A higher value means that there is more error.

Basic Agent Results

All agents were tested with this photo:

Because the Basic Agent iterates through every pixel more than once, it is too slow to run on images of usual dimensions. I ended up running it on an extremely compressed version of my chosen image (120 x 68 vs. the 800 x 450 of the original) and it took around 3-5 minutes to run. Here is the result of one run, which took **3 minutes**.



While the basic agent did pick up some of the wave shape from the picture, and color in most of the ocean correctly, it incorrectly predicts the color of the sand and brings some of the ocean green into the bottom half of the photo. I attribute this to the average color calculations; since green is a more prevalent color in the picture, patches are more likely to be mistaken as green. This prediction had **80.8% error**.

A Note about my Improved Agent(s)

I decided to try writing a neural network for my improved agent to practice the concepts discussed in lecture. As I was experimenting with the different data I could give the generic Neural Network class I wrote, I ended up with two improved agents. Each performs well, but has its drawbacks, so I decided to include both in this report.

The libraries I used were NumPy to make array and math operations easier, Scikit-Image for image manipulation, and Matplotlib to display the final results. Below, I outline the implementation and results of both Improved Agents.

2.2 Improved Agent 1: Color Bins (`improved_agent_1.py`)

This improved agent doesn't use the five representative colors from the basic agent. Instead, I decided to expand the color range by letting RGB values be one of four values: 0, 64, 128, or 192. This made recoloring the photo much quicker since it just involved doing simple division on each pixel's color values. To simplify further, I decided to represent these values as one-hot codes. Then, I generated 3 neural networks (one for each color) and data about each training pixel's red, green, and blue value was stored for therespective neural network. These neural networks were then trained and used to predict the recolored photo. This was just a quick summary; more details about each function and the implementation are discussed below.

- **Neural Network (`neural_network.py`)**
  This is the generic Neural Network class I wrote. It is initialized with a list of layer sizes, and an activation function (for both agents, I chose to use tanh as the activation function, but this neural network could work with the logistic activation function). Weights are initialized to very small random values based on how many layers are provided. The `fit` function accepts input data, training data, a learning rate, and the number of epochs to run, and then performs a forward pass, backpropagation, and updates weights for each epoch. The `predict` function takes in input data and uses the chosen activation function and weights to output a prediction based on what the neural network has already processed.

- **Recoloring the Photo (`recolor`)**
  This function recolors the photo by iterating over each pixel, performing floor division by 64 on each of its R, G, and B values, and then multiplying those results by 64 to get the closest multiple of 64 for the new colors.

- **One-Hot Codes (`one_hot` and `oh_to_reg`)**
  I chose to represent the values 0, 64, 128, 192, simplified to 0, 1, 2, 3, as one-hot codes:

0 (0)	[1, 0, 0, 0]
64 (1)	[0, 1, 0, 0]
128 (2)	[0, 0, 1, 0]
192 (3)	[0, 0, 0, 1]

The `one_hot` function converts a number from 0-3 into a one-hot code to be used for the neural network's training data, and the `oh_to_reg` function converts a one-hot code back into an integer when making final predictions.

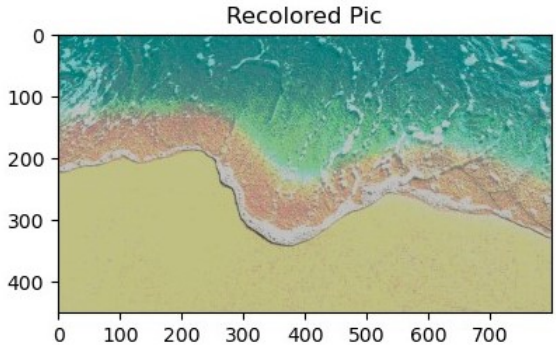
- **Improved Agent 1 (bottom of `improved_agent_1.py`)**
  After the image is recolored using discrete RGB values, I first declare the number and size of the layers to be used in this agent's neural networks. I went with an input layer of size 9 since 9 pixels are processed in each patch, a hidden layer of 9 that uses the tanh activation function, and an output layer of 4 since I intended the output to be a one-hot code (which are of size 4). I also initialized a learning rate (0.1) and number of epochs (20,000).

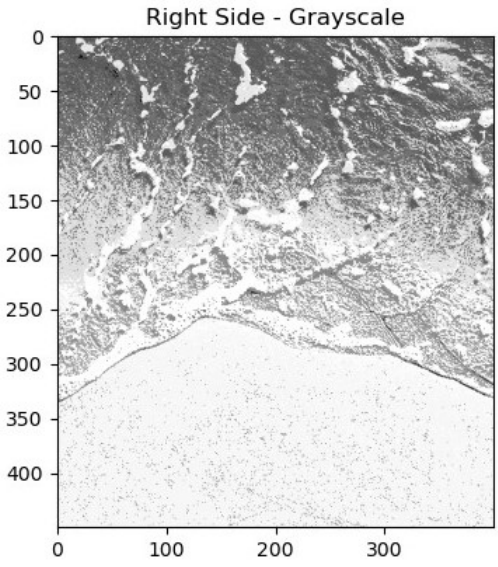
  Then, I initialized three neural networks: one for red, one for blue, and one for green, and the training data sets associated with each neural network. Iterating through each pixel in the training data, I converted each R, G, and B value to a one-hot code and added it to its respective training set. After fitting the data sets to each neural network using the learning rate & number of epochs declared beforehand, I then made a prediction for every pixel, converted the resulting one-hot codes to regular integers between 0-3, and then multiplied those numbers by 64 to yield the final color prediction for that pixel.

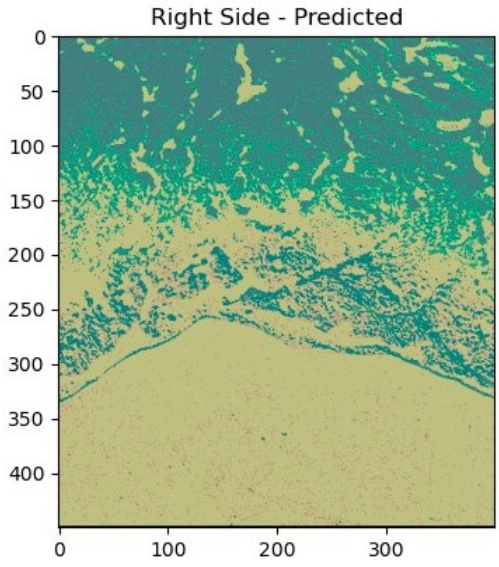
- **Quantitative Difference (`difference`)**
  See Basic Agent. This agent calculates quantitative difference in the same way.

Improved Agent 1 Results

Since neural networks require larger bodies of data to make good predictions, running both improved agents on the compressed photo from the basic agent unsurprisingly yielded subpar results. I decided to use the original, full-size 800 x 450 version of the image to test both improved agents. Since this agent didn't need to find the 5 most popular colors and just did simple division on each pixel to recolor it, that combined with the neural network being faster resulted in this agent's runtime being under 2 minutes, if not under 1 minute, every single time I tested it. Here are the results of a run lasting **20.75 seconds**:







This prediction has **52.632% error**, which means that it is closer to the recolored photo than the Basic Agent was. Qualitatively, it also managed to capture the texture of the waves as evidenced by the grayscale version I displayed next to it.

However, the colors aren't too close to the original image's colors. When the photo was recolored using discrete RGB values, it introduced shades of red into the picture that weren't originally there before, and also tinted the entire photo yellow, possibly due to the RGB values I chose: equally spaced values between 0 and 255, without taking into account that this photo has more green and blue. The basic agent, while definitely inferior to this faster, more detailed agent, had the colors down accurately.

This made me wonder if I could combine the accurate representative colors from k-means with the speed and accuracy of this neural network, and in experimenting with that thought, I ended up with another improved agent.

2.3 Improved Agent 2: Representative Colors

This agent uses the Neural Network class from the first agent, but instead of recoloring the photo with discrete RGB values, it recolors the photo with the five representative colors chosen by my k-means algorithm. It then assigns each of these colors a one-hot code, and then goes through training, fitting, and prediction using a single neural network with a slightly different structure. More details about this implementation are below.

- **Neural Network** (`neural_network.py`)
  See Improved Agent 1. This improved agent uses the same Neural Network class with a slightly different initialization, which is talked about in the forthcoming sections.

- **Recoloring the Photo** (`five_color`)
  This function recolors the photo by using the five representative colors determined by k-means.

- **One-Hot Codes** (`one_hot` **and** `oh_to_reg`)
  My k-means function returns a list of five colors. The presence of a color in a pixel translates to a one-hot code that corresponds with the position of that color in the returned list.

  Given a list of representative colors [A, B, C, D, E]:

A	[1, 0, 0, 0, 0]
B	[0, 1, 0, 0, 0]
C	[0, 0, 1, 0, 0]
D	[0, 0, 0, 1, 0]
E	[0, 0, 0, 0, 1]

  The `one_hot` function converts the presence of a specific color into a one-hot code to be used for the neural network's training data, and the `oh_to_reg` function reads a one-hot code, determines the largest value in the code, and returns the corresponding index to a color from the list of five representative colors.

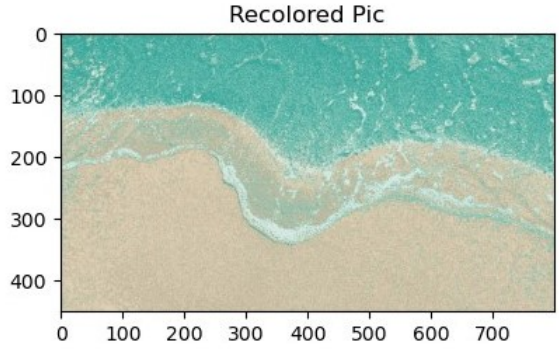
- **Improved Agent 2 (bottom of** `improved_agent_2.py`**)**
  After the image is recolored using the five representative colors, I first declare the number and size of the layers to be used in this agent's neural networks. I went with an input layer of size 9 since 9 pixels are processed in each patch, a hidden layer of 9 that uses the tanh activation function, and an output layer of 5 since I intended the output to be a one-hot code (which are of length 5, unlike in the first improved agent where one-hot codes were only of length 4). I also initialized a learning rate (0.1) and number of epochs (20,000). Both of these values are the same as improved agent 1.

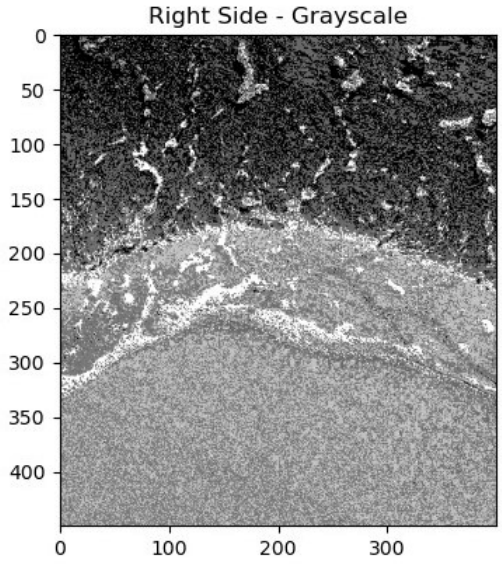
  Then, I initialized only one neural network to work with the RGB colors in their entirety, and a training set associated with it. Iterating through each pixel in the training data, I converted each pixel's color to a one-hot code depending on what color it was, and added it to the training set. After fitting the data set to the neural network using the learning rate & number of epochs declared beforehand, I then made a prediction for every pixel, converted the resulting one-hot codes to regular integers between 0-4, and then used the corresponding color at that index from the list of five colors for the final color prediction of that pixel.

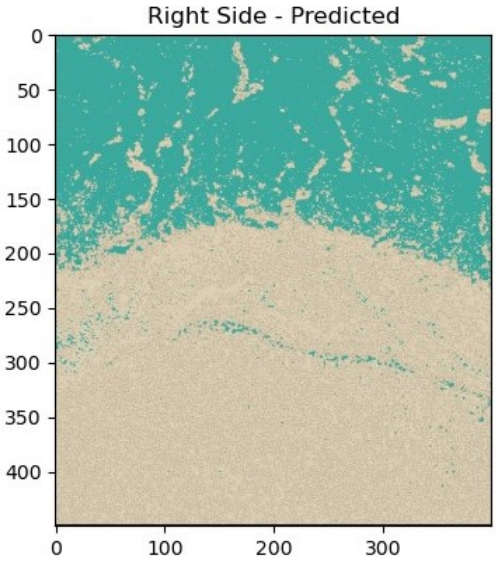
- **Quantitative Difference** (`difference`)
  See Basic Agent. This agent calculates quantitative difference in the same way.

Improved Agent 2 Results

As I explained in the results from Improved Agent 1, I used the full image to test this improved agent. Since this agent had to use my k-means algorithm to find the 5 most representative colors before even beginning to train the neural network, this agent's runtime is around 1-2 minutes. Here are the results of a run lasting **57.8 seconds**:







This prediction has **31.017% error**, so it is more accurate than Improved Agent 1, but not as bad as the Basic Agent. Qualitatively, the colors are pretty much spot-on compared to the original photo. A lot of texture that Improved Agent 1 captured was lost in this result; for example, the white in the waves is nonexistent. However, when compared to the grayscale version of the right side, clearly the shapes of the texture were preserved, it's just the color that is missing.

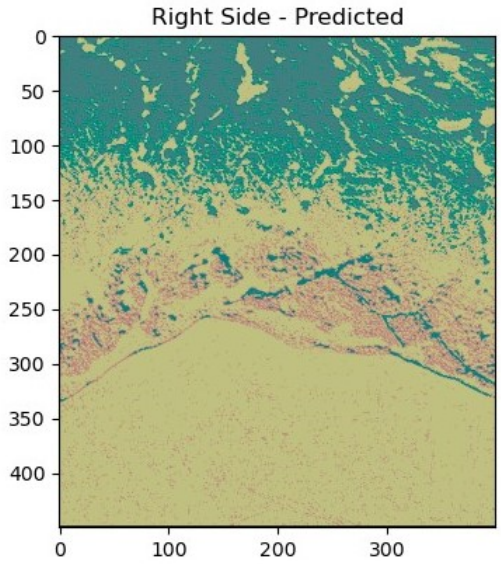
3 Overall Analysis

To summarize, here are the quantitative and qualitative observations of the basic agent, Improved Agent 1, and Improved Agent 2:

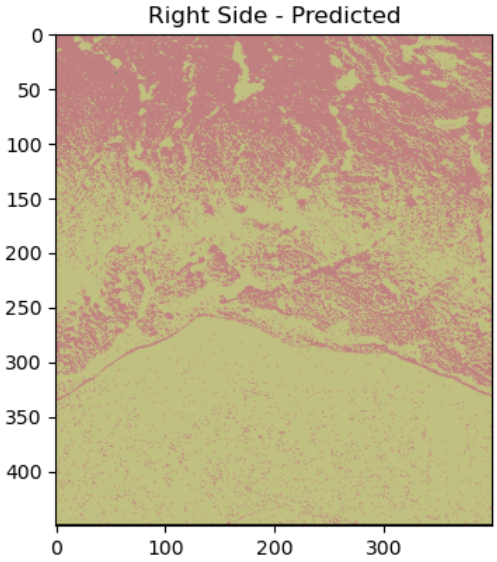
Agent	Runtime (in min)	Error	Observations
Basic	3:00 (compressed)	80.833%	More green & blue than original
Improved 1	0:20.75	52.632%	Detail preserved, inaccurate color
Improved 2	0:57.8	31.017%	Color mostly preserved, somewhat inaccurate detail

A prediction being quantitatively closer to the original does not always mean that it will look aesthetically pleasing, as I experienced with Improved Agent 1. Although there was definitely improvement over the Basic Agent & the error percentage was very low, the yellow tint and newly-colored shades of red in the result made me start trying to find an alternative that had better coloring.

I attempted to get Improved Agent 1's result to become less yellow, but as I increased learning rates on the R, G, and B neural networks, more and more red started becoming visible in the picture. As an example, here is Improved Agent 1's result with learning rates of 0.6 for R, G, and B:



And, just for fun, here is an extremely overfitted Improved Agent 1, with a learning rate of 0.8:



At this point, I decided to leave the learning rate at 0.1 to prevent overfitting.

On the other hand, Improved Agent 2 looked way more accurate than Improved Agent 1 at first glance thanks to the coloring technique, and was also quantitatively more accurate thanks to the color comparison used by my difference-finding function, but in reality there was some loss of detail and it also wasn't as similar to the original as Improved Agent 1 was.

Since my curiosity about quantitative versus qualitative accuracy was peaked at this point, I decided to ask some other observers (my family and some friends who are not Computer Science majors) what they thought of the two results. Although me asking them was not a formal investigation in any way, they overwhelmingly said that while Improved Agent 2's result looked closer to the original, Improved Agent 1 had more detail, which supports the data and my own conclusions.

Of course, both Improved Agents were much, much better than the Basic Agent in results and runtime.

4 Future Improvements

The ideal coloring agent would combine Improved Agent 1's detail accuracy with Improved Agent 2's color accuracy. I believe the following ideas would result in improvements if I had the extra time and resources to spend on this project.

- **Modifying the discrete RGB values in Improved Agent 1 to be more biased towards blue and green.** While my even approach to binning RGB values would be fair for most photos, it does this photo some injustice since there is way more blue and green than red in it.
- Following up on the last point, perhaps **using the five representative colors to determine bin values** could have yielded better results. This could be done by finding the five representative colors, then finding the average of each R, G, and B from those colors, and evenly spacing bin values around that average. That way, the color space would still be small enough to process the photo quickly, the recoloring would make sense against the original photo, and this approach is adaptable to any image no matter what its dominant colors are.

- I calculated the error between the intended outcome and the predicted outcome by using solely pixel color: if the RGB values were off by even one, the pixel was considered "wrong". However, when it comes to color and the bigger picture (no pun intended), a very small difference in RGB values isn't apparent to the human eye. I suspect this is why the error rates were higher to begin with. If I had more time I would have **researched an acceptable range of RGB values to check against** so that the error rate could be a bit qualitative as well, and determine whether a color difference was large enough to be detected by the human eye.

- I also didn't experiment much with the learning rate, number of epochs, or activation functions for my neural network. After my initial experimentation on Improved Agent 1, I kept them standard across both agents (learning rate of 0.1, 20,000 epochs, and tanh activation). **Trying different values and activation functions** could have potentially resulted in a better prediction, especially in Improved Agent 1 if I had given each neural network (R, G, and B) a different learning rate.

- Both Improved Agents rely on discrete classification. **Implementing a regression model** could improve accuracy, if I had the time.

- Finally, this was a very basic neural network written by myself for the purpose of this project. Many machine learning libraries already exist, and **using a library like scikit-learn or TensorFlow** would have resulted in faster and much more accurate prediction. This was the group portion of the project, but as I am working solo it would have taken more time to implement.