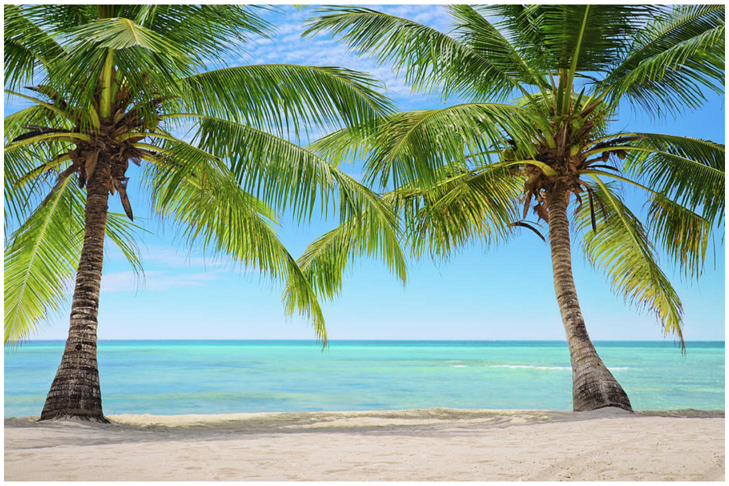
Intro to AI: Project 4 - Colorizer

Ashwin Haridas (ah1058), Ritin Nair (rrn32)

May 2021

1 Introduction

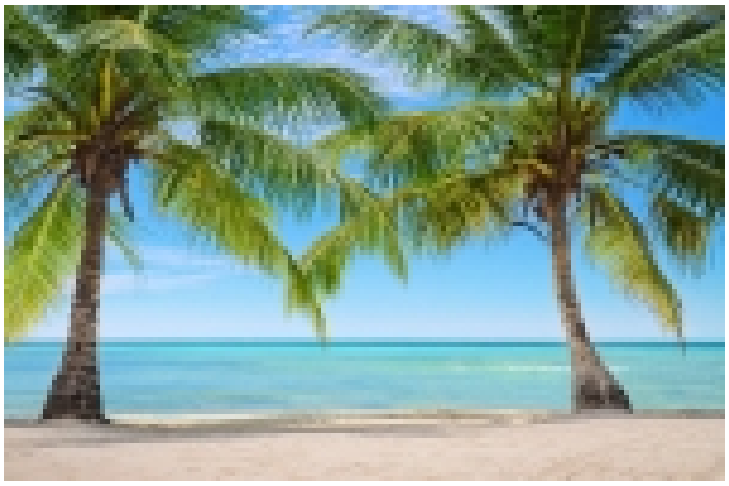
In this assignment, we are tasked with creating a colorizer. The colorizer is meant to use a colored image (in this case, the left side of an image) as training data and then recolor a black and white image (the right side of an image) using some machine learning model. The following image was used to test our agents:



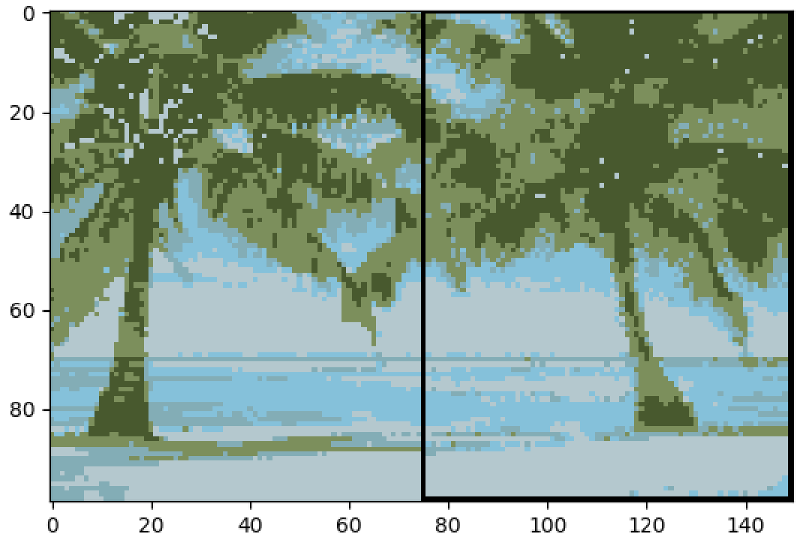
The purpose of this assignment is to showcase our understanding of basic machine learning principles. As like the previous assignments, we first implement a basic agent and then attempt to create and implement an improved agent.

2 Basic Agent

The basic agent works like this: first, we run k-means clustering on the training data (the left colored side) to find the 5 best representative colors ($k = 5$). Then, we recolored the right side by using 3 x 3 gray scale patches and the algorithm provided in the project description. One thing to note: since the recoloring process takes a long time (we have to compare each 3 x 3 gray scale patch on the right side with every single gray scale patch on the left side), we reduced the size of the original image. The reduced image is as follows:



After running the basic agent, we have:



2.1 Question 1

How good is the final result?

Solution:

The final result using the approach of the basic agent has the right side of the image colored by five representative colors. Due to the entire image only having five colors, the visual quality of the original is not able to be entirely captured. However, we are still able to make out the general structures present in the original image. For example, we used a picture of a beach with two trees as a sample test case. In the recolored version of this picture we are able to make out defining characteristics that allow us to get an understanding of what the picture is of. Overall, with the use of the basic agent for recoloring the final result ends up being good enough to tell what the image is of, but not good enough to capture the wide range of colors in the original image.

2.2 Question 2

How could you measure the quality of the final result?

Solution:

To measure if the structure of the original image is preserved we need to look at the recolored image and compare it visually to the original. For example, in the sample image of the beach we used in our trials we can see certain structures such as trees, the sky, and the ocean are all somewhat preserved by the recolored image. However, simply looking at the visuals of the recolored image to measure the quality of the recoloring can end up being relatively subjective and inaccurate. This is heavily dependent on how individuals perceive color and can result in inconsistent conclusions depending on who evaluates the quality of the recoloring. Therefore, it's more beneficial to measure the effectiveness of the recoloring numerically instead. We can do this by iterating through each pixel in the right half of the image which we recolored. For each pixel we then calculate the euclidean distance between the vector of RGB values of the original images pixel and the vector of RGB values of the recolored images corresponding pixel. The euclidean distance between these two vectors acts as a measure of how similar the two colors of these pixels are. If two pixels have the exact same color then the euclidean distance between the vectors representing these colors will be 0. Therefore, the closer the colors are to each other then the closer the euclidean distance will be to 0. The largest euclidean distance possible would be if there is a pixel that is black and it is recolored to be white (or vice-versa). This would result in a euclidean distance given by the following formula:

$$\text{Euclidean Distance} = \sqrt{(255-0)^2 + (255-0)^2 + (255-0)^2} \approx 441.67$$

These euclidean distances are calculated for each pixel on the right hand side of the image (excluding the border pixels which we color black) and then summed up. This is shown by the following formula:

$$\text{Euclidean Distance} = \sum_{i=0}^{N} \sqrt{(r_i - R_i)^2 + (g_i - G_i)^2 + (b_i - B_i)^2}$$

In this formula i represents each pixel to be iterated through (all pixels on the right half of the image which was recolored except for border pixels), N represents the total number of pixels to compare, r_i represents the recolored image's red value for pixel i and R_i represents the original image's red value for pixel i (and so forth for the green and blue pixels with g_i/G_i and b_i/B_i respectively). Finally, this sum is averaged to give an overall measure of how similar the recolored image is to the original one by dividing by N, or the total number of pixels compared.

2.3 Question 3

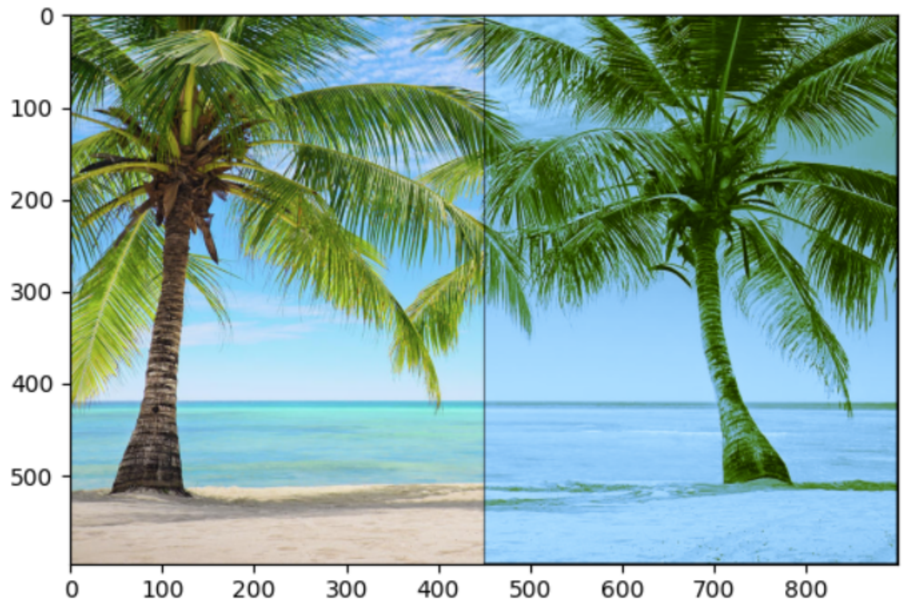
Is the recolored image numerically satisfying, if not visually?

Solution:

Following the answer to the previous question, we found the average euclidean distance between the recolored image and the original image to be 259.46. Since this number is very large and pretty close to the max euclidean distance value of 441.67, we can say that image is not very numerically satisfying. We can say that the recolored image is kind of visually satisfying, since the most representative colors are present and the general structure of the image is still preserved.

3 M.I.N.T - Improved Agent

Our improved agent is called "M.I.N.T", which stands for **M**ore **I**mprovement **N**ecessary with **T**ime. We used the original picture (not rescaled) to test, and with a regression model, we got the following result:



3.1 Agent Overview

In the basic agent for this recoloring problem the strategy used was classification in which each middle pixel of a gray patch was assigned to one of five representative pixel colors. A major disadvantage of this approach is that it is unable to capture how the pixel values of the colors in the original image are not just five discrete values, but are also continuous in nature. This issue can be remedied by applying a regression model that takes in gray-scale patches as an input and outputs the red, green, or blue pixel values of the middle pixel of the patch. These values can fall anywhere in between the range of 0 and 255 and are not necessarily bound to five distinct values like how the output from the classification process in the basic agent was.

The input space for this problem was a 1 x 10 vector built from the black and white pixel values from the image. The first component of this vector was assigned the value 1, and the rest were assigned values from each of the nine pixels of the patch. The reason we make the first value of this vector a 1 is so when we take the dot product of this input vector with a vector of weights we are able to isolate a bias value. This is necessary because our model does not necessary pass through the origin, so the bias accounts for this offset.

The output space for this problem is a singular value from 0 to 255 which represents the pixel value for the color the model is for. The three colors, red, green, and blue, are each trained individually with the same model. The output for all of these models is also the same and always falls in between the range of 0 to 255 to ensure that the pixel value assigned is valid.

The model space for this problem is given by the following equations:

$$R(X) = 255 * \sigma(\underline{w} \cdot \underline{x})$$
$$G(X) = 255 * \sigma(\underline{w} \cdot \underline{x})$$
$$B(X) = 255 * \sigma(\underline{w} \cdot \underline{x})$$

As shown by the above equations, we choose to use the same type of model for each of the three colors, but train the models separately so the weights we get for each one end up being different. This is to account for how depending on the image used to train the models, some colors may be more prominent than others. The model can also only output one value and would not be able to completely capture mapping three separate values and once, thus making it necessary to train each of these three instances of the model separately.

The error / loss function we chose to use for this problem is shown by the equation below:

$$Loss(f_{\underline{w}}(\underline{x}), y) = (f_{\underline{w}}(\underline{x}) - y)^2$$

In other terms, the loss function for each of the three colors would be:

$$L = (R(x) - r)^2$$
$$L = (G(x) - g)^2$$
$$L = (B(x) - b)^2$$

The learning algorithm we used for this problem to train the models and find weights that minimized the loss based on our loss function was stochastic gradient descent. We used the left half of the image for training the model and the right half of the image for testing. The first step of the improved agent algorithm was to initialize a vector of weights. We chose to manually initialize the weights as a low value of 0.0005 to accommodate our sigmoid based model. While these weights were not randomly selected, it was still beneficial to take this approach to ensure that the model did not converge too early, or converge at weights that were too high which would make the sigmoid function in our model always output 1. This is because for σ(z) as z begins to approach 5, σ(z) starts to asymptotically approach 1. Having this value always be 1 would result in a predicted output that is always 255 then. We also chose a low alpha of 0.00005, or learning rate, value to ensure that the updates on the weights were relatively low compared to the values of the weights themselves (which also tend to be low). Choosing a low alpha value means that the algorithm took longer to converge at weights that minimized the loss, but it was less likely that taking a step would result in a minimum of the loss function being missed. The way our stochastic gradient descent learning algorithm works is by first selecting initial weights. Then, at each time t picking one random pixel i is picked and a gradient of the loss function is calculated based on the patch values of this pixel i. These updates are repeated 10000 times until convergence as this algorithm tends to converge after each pixel has been used for the gradient approximately once. After stochastic gradient descent has been run for each of the three colors, the weights gotten are used to predict the output color value of each gray-scale patch on the right half of the black and white image to recolor it.

3.2 Parameters

We chose to to include a sigmoid function in the structure of the model, and multiply the output of the sigmoid function by 255. The reason for this was to bound the predicted outputs from our model between the values of 0 and 255 to ensure that any outputs gotten were valid pixel values. The reason sigmoid was chosen was because the output of a sigmoid function is bound between 0 and 1, and multiplying these values by 255 results in the output being bound between 0 and 255.

We chose to initialize the weights of our algorithm to 0.0005 to ensure that the weights were low enough such that $\sigma(\underline{w}\cdot\underline{x})$ would not always output 1. This would result in our model converging too soon and the updates on the weights in the learning algorithm not resulting in any changes.

We chose the value of alpha, or the learning rate, to be 0.00005 because having a smaller step size decreases the likelihood of the weights that would minimize the loss function being skipped over as a result of the update being too large in magnitude. This helped guarantee convergence at a slightly more consistent value that was closer to weights that actually minimized the loss function.

3.3 Pre-Processing

We pre-processed the input by dividing each component of the gray-scale vector patch by 255 to normalize the values in the input vector between 0 and 1. Doing this helped guarantee that when computing $\sigma(\underline{w} \cdot \underline{x})$ the dot product inside the sigmoid function would not result in too high of a value as a result of the input vector. If this were the case, then the sigmoid function would always return 1 and cause the predicted pixel output value to always be 255. By pre-processing the input in this way we help guarantee that the predicted output can be anywhere in the 0 to 255 range, and not simply always output 255.

3.4 Training

The following calculation is for the derivation of the gradient of the loss function. This example uses the model for red pixel values but it can likewise be applied to the models for green pixel values and blue pixel values to get the same result:

$$\frac{\partial}{\partial \underline{w}}(r - R(X)^2)$$
$$2 * (r - R(X)) * \frac{\partial}{\partial \underline{w}}(r - R(X))$$
$$R(X) = 255 * \sigma(\underline{w} \cdot \underline{x})$$
$$2 * (r - 255 * \sigma(\underline{w} \cdot \underline{x})) * \frac{\partial}{\partial \underline{w}}(r - 255 * \sigma(\underline{w} \cdot \underline{x}))$$
$$2 * (r - 255 * \sigma(\underline{w} \cdot \underline{x})) * (-255 * \sigma'(\underline{w} \cdot \underline{x}) * \frac{\partial}{\partial \underline{w}}(\underline{w} \cdot \underline{x}))$$
$$-2 * (r - 255 * \sigma(\underline{w} \cdot \underline{x})) * (255 * \sigma'(\underline{w} \cdot \underline{x}) * \underline{x})$$
$$\sigma'(\underline{w} \cdot \underline{x}) = \sigma(\underline{w} \cdot \underline{x}) * (1 - \sigma(\underline{w} \cdot \underline{x}))$$
$$-2 * (r - 255 * \sigma(\underline{w} \cdot \underline{x})) * (255 * \sigma(\underline{w} \cdot \underline{x}) * (1 - \sigma(\underline{w} \cdot \underline{x})) * \underline{x})$$
$$\nabla_{\underline{w}} Loss(f_{\underline{w}_t}(\underline{x}_i), y) = 2 * (255 * \sigma(\underline{w}_t \cdot \underline{x}_i) - y) * (255 * \sigma(\underline{w}_t \cdot \underline{x}_i) * (1 - \sigma(\underline{w}_t \cdot \underline{x}_i)) * \underline{x}_i)$$

In the above formula $f_{\underline{w}_t}(\underline{x}_i)$ is the predicted output given an input and y is the actual output. This gradient is used in our updates for the weights in the stochastic gradient descent learning algorithm. In simpler terms the gradient is:

2 * (predicted - actual) * (predicted) * (1 - predicted/255) * ($\underline{x}$)

We handle training the model in the improved agent by using stochastic gradient descent. First we initialize a 1 x 10 vector of weights to have initial weight values of 0.0005. Then, we pick a random pixel, i, to use as an input vector that predicts an output which is used in a calculation for the loss function. The updates are done as follows:

$$\underline{w}_{t+1} = \underline{w}_t - \alpha * \nabla_{\underline{w}} Loss(f_{\underline{w}_t}(\underline{x}_i), y)$$

In our implementation of the improved agent the update values were very small so it was difficult to find an update value that indicated convergence. Therefore, we repeated this process 10000 times until approximate convergence and to give the stochastic gradient descent algorithm a chance to use each pixel in the training half of the image around once. This is because stochastic gradient descent bases an update at each time step on only one input vector, so to increase the likelihood of the entire input space being used in the training process we run these updates 10000 times.

We accounted for over-fitting the weight values of our model by not repeatedly adding characteristics to the input vector. We kept the input as a 1 x 10 array and did not take a quadratic approach for the model to avoid over-fitting. Doing so would have resulted in potentially needing to repeatedly raise the power of the model which in turn could over-fit it, making it perform really well for the specific image it was trained on but poorly for other similar images. For this reason, we stuck with the sigmoid model and only manually changed values related to pre-processing, the learning rate, and the initial weights passed to the learning algorithm.

3.5 Comparison with Basic Agent

We can see that the output from the improved agent is significantly more satisfying visually than the output from the basic agent. It has a more diverse range of colors and preserves the textures and structure of the original image much better than the basic agent does. For example, we are able to more clearly see shadows and textures on the tree in the improved agent output, whereas in the basic agent these textures are seemingly lost. The improved agent also more closely captures places in the original image where there are gradients between colors that have slight shifts. While the improved agent is definitely more visually satisfying than the basic agent, they both still share the issue of not being able to capture the brown colors present in the original image. They both resort to coloring tree branches and sand green instead of brown, meaning that the weights of our red model were perhaps too low after the training process. In spite of these inaccuracies in the colors of certain objects, the improved agent clearly shows that it captures the structures and textures of the original image significantly better than the basic agent does.

The improved agent also beats out the basic agent numerically. The average euclidean distance between all of the recolored pixel values and actual pixel values from the original image for the improved agent was 204.86. This is better than the basic agent which had a value of 259.46. This comparison is partially fair because both models recolor the image with new pixel values so we can take the euclidean distance for both agents. However, the improved agent does simply beat the basic agent with regard to this measurement because of the nature of its model. Since the basic agent uses a classification model it can only assign pixels one of five values. This results in the difference between the actual and predicted values being higher than the improved agent's naturally. The improved agent uses a regression model that outputs continuous values so it is likely that the difference between the predicted and actual values of the pixel colors will be lower. Therefore, this measurement is only partially valid and only implies a slight performance increase for the improved agent over the basic agent numerically.

These qualitative and quantitative measures of the improved agent's performance indicate that it is slightly better than the basic agent in terms of the way it looks and its similarity to the original image's pixel values.

3.6 Further Improvements

The improved agent could definitely be improved with sufficient time, energy, and resources. One improvement we can do is increase the amount of training data that we have. Currently, we only use the left half of the image to train and then use that data to test on the right half of the image. If we were to use a large database of images, our model could potentially improve drastically, which would improve the image quality. Another improvement could be utilizing the resources to find a more optimal learning rate and initial weights for the model. We found while working on the assignment that it took a very long time to find a learning rate and initial weights that we were satisfied with, but if we had the resources we could probably automate this to find the most ideal ones. One other improvement we could do with more time / resources was in the training process. Instead of using stochastic gradient descent to minimize the loss function and find our optimal weights, we can use regular gradient descent. The regular gradient descent takes much longer than stochastic gradient descent, but it will minimize the loss function by actually changing every single weight for every single x, instead of choosing an x randomly. This will result in a more accurate update of the gradient. Another improvement could be to create a more complicated model to improve the accuracy of the agent. Our current model is a very simple regression model that has an input space of 9 pixels and outputs one red, green, and blue value. If we had the time, we could also try to implement a more complicated model (possibly through the use of already made libraries such as TensorFlow or SciKit-Learn). When doing this, instead of using a simple regression model, we would implement a neural network (since neural networks are really just advanced regression models).

3.7 What Didn't Work and What We Could Do Better

Looking at the result of our improved agent, we didn't think it was as good as what we were expecting. The image itself is pretty green overall (which does not really capture the sand on the bottom). We think what didn't work in our model is that we were not choosing the proper initial weights and learning rate for each of our three models. Possibly, what could have been happening was that our weights and learning rate for the green model were accurate, but the red and blue models were not. Therefore, what we could have done better is spend more time finding an optimal initial weight / learning rate for each of our models, instead of just using the same values for each of them. Also, something else that we could have done better was to actually find the correct converging threshold value. Currently, we run the update algorithm 10000 times (we chose this number from trial and error), but if we had more time to test, we could have actually spent time figuring out the correct value to stop at. This could have possibly made our agent much more accurate than it currently is. The last thing that caused us issues was the sigmoid function. It was hard to use the sigmoid function in this model because it had a tendency to output values too close to 1. This was happening even when we normalized our inputs and tried to choose initial weights that were very small. The range of the input values for the sigmoid function that would result in an output that was not 0 or 1 was very small, so there was not much room to find inputs that produced outputs that were not 0 or 1. To avoid this issue, we could potentially use a different activation function (maybe the tan function), or think of a completely new model.

3.8 Bonus: Why a Linear Model is a Bad Idea

A linear model is a bad idea because we know in advance that the output must be between 0 and 255 (as these are RGB values for the pixel). If we were to use a linear model, then it is entirely possible for the weights to converge at any value, which results in the predicted value to be any value (potentially greater than 255 or less than 0). If the output from a linear model is not between 0 and 255, then it loses relevance and would mean that the model is not accurate. Even though we pre-process our input data to be between 0 and 1, our output could not end up being between 0 and 255. Therefore, we use the sigmoid function to force our output to be between 0 and 1 (the weights would be allowed to converge to anything in this case), and then multiply it by 255 to get our final output to be between 0 and 255.

4 Contributions

In this project we tried our best to split up the workload evenly. For the implementation of the basic agent Ashwin did the setup for processing the image into RGB arrays and the algorithm for k-means clustering to find the 5 representative colors for the image. Ritin did the second half of the basic agent where this information was used to recolor the right half of the image by going through patches on the right half and finding similar patches on the left. For the improved agent, Ashwin did the setup for creating all necessary initial vectors and obtaining values from the image. Ritin then used these to implement stochastic gradient descent and produce an output for the improved agent. Ashwin and Ritin both worked on determining a model to use, a loss function to use, and the derivation of the gradient of the loss function. For the report, Ashwin worked on the introduction, the basic agent overview, and questions 2.3, 3.6, 3.7, 3.8. Ritin worked on questions 2.1, 2.2, 3.1, 3.2, 3.3, 3.4, and 3.5. After answering questions individually, we both then looked over each other's work and worked on parts of the report we needed to fix together. Ashwin also worked on commenting the code to give a clearer understanding of what we did in each part of it.

5 Honor Code

This assignment was done on our own. None of the code or the report was copied or taken from online sources or any other student's work.