CS440 Project 4: Colorizing

Mustafa Sadiq (ms3035)

May 5, 2021

Introduction

In this assignment, we color a black and white image using some basic techniques in supervised learning and computer vision.

Libraries required: *numpy*, *Image from PIL*, *Counter from collections*

Setup

Given an original color image we first make a copy of it and convert this copy to black and white using the following color to gray conversion forumla:

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

We then split both the color and bw image into training and testing data. The left side of the image is training data and the right side testing data.

We now have:

- color_image_training
- color_image_testing
- bw_image_training
- bw_image_testing

Basic agent

The basic agent runs as follows:

- colorized_image = np.copy(bw_image_testing)
- find 5 mean color of color_image_training using k-means algorithm and Euclidean difference for color difference
- color color_image_training using these 5 colors
- for x, y in bw_image_testing:
  - if x, y border pixel of image:

    colorized_image[x][y] = np.array([0, 0, 0])

    continue
  - else:

    colors = [ ]

    current patch = 3x3 patch in bw_image_testing with center x, y

    for each 6 most similar 3x3 pixel patches in bw_image_training with center z, w in decreasing order of similarity:

        colors.append(color_image_training[z][w])

    if len(mode) of colors != 1:

        colorized_image[x][y] = colors[0]

    else:

        colorized_image[x][y] = mode of colors
- output image
  - left as color_image_training
  - right as colorized_image

Basic agent: results

We run our basic agent on the following image of Taj Mahal:
(Image credit: saiko3p Shutterstock) (Dimensions: 512×341)

Figure 1: Original image (Size: 29.8 KB)

Converting it to black and white and splitting it:

Figure 2: Black and white image (Size: 27.8 KB)

- Left: bw_image_training
- Right: bw_image_testing

Splitting the original image and coloring color_image_training with 5 mean colors of color_image_training:

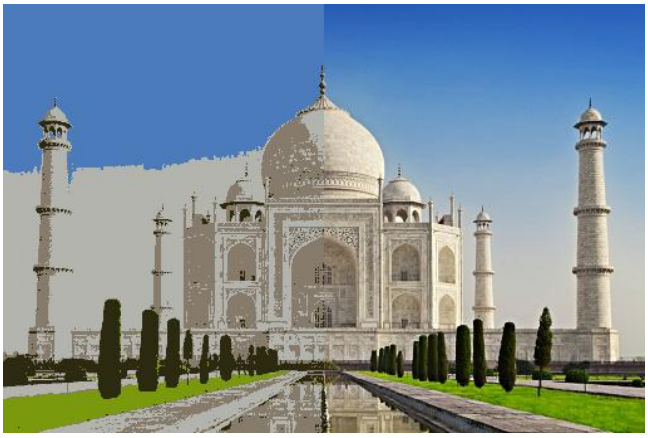

Figure 3: Color image (Size: 28.9 KB)

- Left: color_image_training colored with 5 mean colors of color_image_training
- Right: color_image_testing

Output image after colorizing:

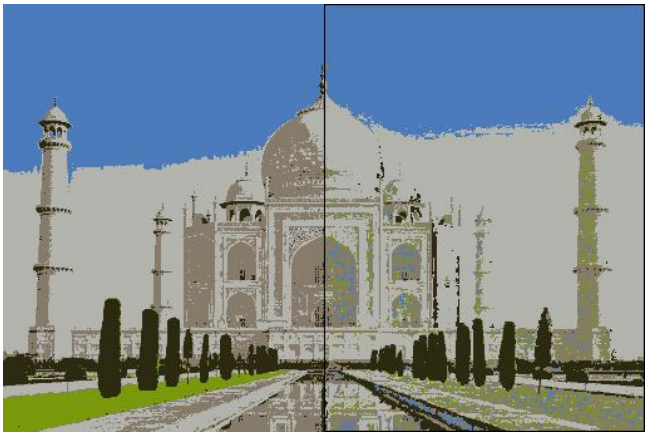

Figure 4: Output image (Size: 30.9 KB)

- Left: color_image_training colored with 5 mean colors of color_image_training
- Right: colorized_image of bw_image_testing

Basic agent: analysis

Visually, the basic agent does perform well, at least enough that all features of the image are differentiable in color. The green grass got colored a mix of both green and the grey shade, unlike the uniform green on the color_image_training side of the picture. There are a few blemishes on the Taj Mahal itself, where some parts got colored dark green. To find the difference between the original image and the output image we find the Mean Square Error as follows:

- Add a black pixel border to color_image_testing so that comparison is fair with colorized_image.
- Find Mean Square Error

MSE: 94.8820526331867

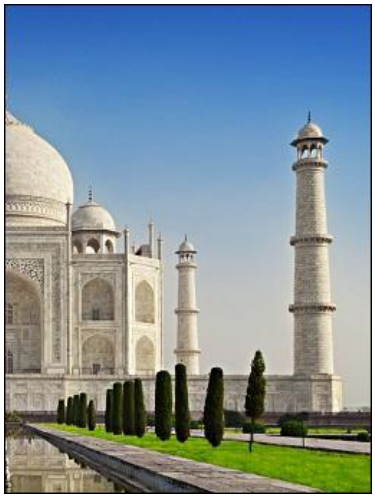

Figure 5: Original image

Figure 6: Colorized image

(Bonus) K-means: which k to choose?

The best number of representative colors for our Taj Mahal image is 6. We can group the colors present in the original image into 6 clusters:

- Blue: shade of sky
- White: shade of sky, Taj Mahal itself
- Dark green: bushes
- Green: grass
- Gray: non-white features of Taj Mahal and reflections
- Black: shadows and features of Taj Mahal tiles and algae on pavement

If we pick any number less than 6, we will lose information e.g. blacks, if we pick 5, get colored as dark green. If we pick any number greater than 6, we will have two similar clusters, which will not add much information to identify each part of the image.

Improved agent

For our improved we implement three Logistic Regressions models for the red, green and blue values respectively. The input values are a patch of 9 grayscale values with center x, y and the output red/green/blue value for x, y pixel between 0 and 255.

Given the sigmoid σ function as:

$$sigmoid(z) = 1/(1 + np.exp(-z))$$

We define our regression models as:

$$R[x] = 255 * sigmoid(w * x)$$

The loss function as:

$$L[i] = (R[x[i]] - r_i)^2$$

The update function as:

$$w_{new} = w - \alpha \nabla_w L[i]$$

The loss function derivative w.r.t. weights as:

$$\text{gradient} = \frac{d}{dw_j} L[i] = 2 * (R[x[i]] - r_i) * R[x[i]] * (1 - \frac{R[x[i]]}{255}) * x_i$$

Our update function becomes:

$$w_{new_j} = w_j - \alpha * \text{gradient}$$

We prepare the data for modeling as follows:

- for every non-border x, y value in color_image_training:

  color_image_training_r_data = red value
  color_image_training_r_data = green value
  color_image_training_r_data = blue value

- for every non-border x, y value in bw_image_training:

  bw_image_training_data = 3x3 patch grayscale values

- for every non-border x, y value in bw_image_testing:

  bw_image_testing_data = 3x3 patch grayscale values

- for every non-border x, y value in color_image_testing:

  color_image_testing_r_data = red value
  color_image_testing_r_data = green value
  color_image_testing_r_data = blue value

Before fitting our model we first pre-process bw_image_training_data and bw_image_testing_data:

- divide all values by 255
- insert 1 as bias so that we have $\{1, g1, \ldots, g9\}$

We are now ready to train our model. We devise the following algorithm using Stochastic Gradient Descent to minimize our loss function:

Taking bw_image_training_data as input and color_image_training_r_data, color_image_training_g_data, color_image_training_b_data as outputs respectively.

```
weight = np.random.uniform(0.0, 0.01, 10)
for iteration in max_iterations:
    random_sample = np.random.choice(input.shape[0])
    random_input = input[random_sample]
    random_output = output[random_sample]
    predicted = 255*sigmoid(np.dot(random_input, new_weights))
    for i in range(10):
        new_weights[j] = new_weights[j] - (learning_rate * gradient)
```

We are now ready to fit our model using bw_image_testing_data:

```
predicted_r = 255*sigmoid(np.dot(bw_image_testing_data, weight_r))
predicted_g = 255*sigmoid(np.dot(bw_image_testing_data, weight_g))
predicted_b = 255*sigmoid(np.dot(bw_image_testing_data, weight_b))
```

Improved agent: results and analysis

Given the predicted r, g, b values we combine them to produce the following colorized image.

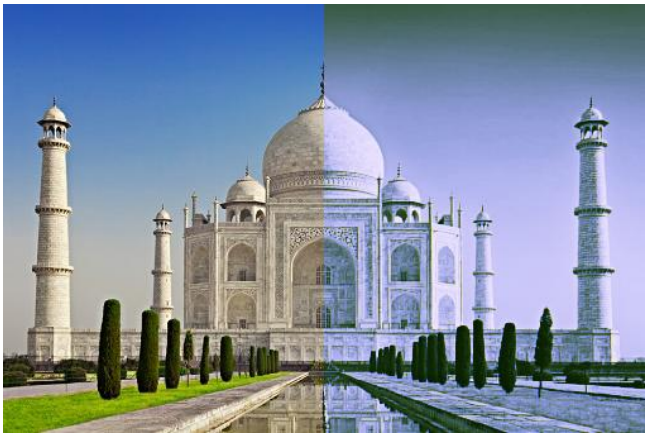

Figure 7: Improved agent colorized image

- Left: color_image_training
- Right: Improved agent colorized image of bw_image_testing

Visually, the improved agent does not perform that well, just enough that green is differentiable from blue/white. The blue and the white got mixed up. To find the difference between the original image and the output image we find the Mean Square Error:

MSE: 82.77412723851997

In terms of MSE, our improved agent beats the basic agent.

Figure 8: Original image

Figure 9: Colorized image

To find out why our model didn't perform so well we look at our loss function over time with each iteration of the Stochastic Gradient Descent:

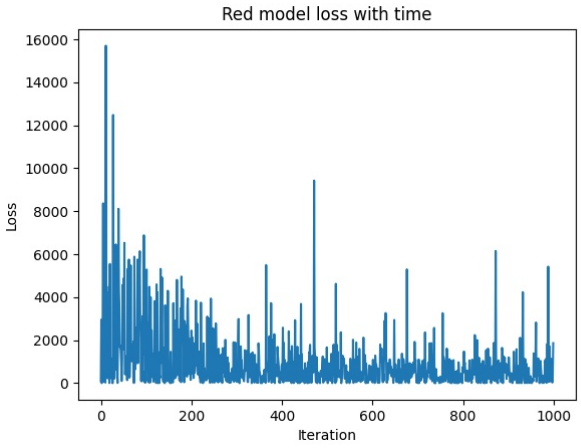

Figure 10: Red model loss with time

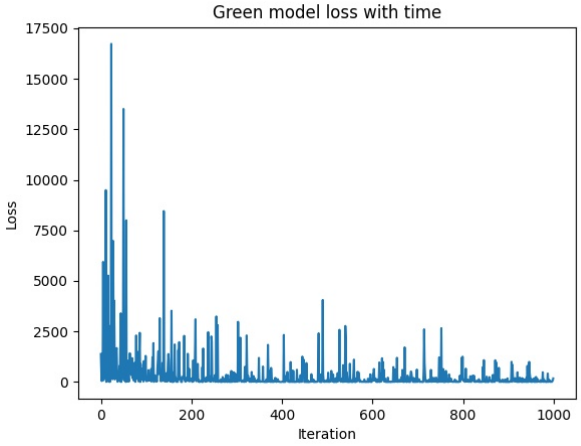

Figure 11: Green model loss with time

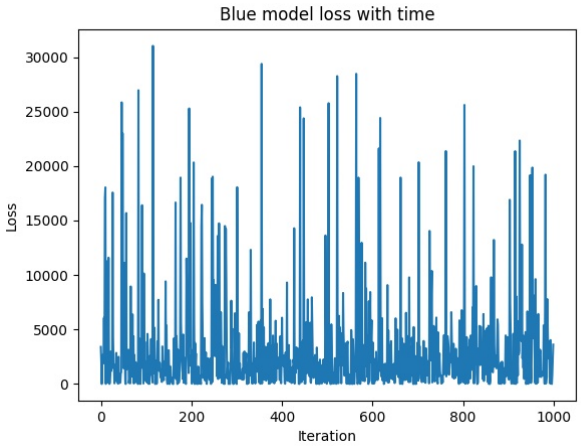

Figure 12: Blue model loss with time

As expected we find that the Red and Green model trained well, whereas the Blue model doesnt fit at all and oscillates, most likely for the confusing b values which are same for white and blue.

A Logistic regression does not fit well to our problem. If we had sufficient time, energy, and resources, we should look at other models, particularly at a neural network which will a multi layer regression and take into account different features such as shapes and edge detection.

Bonus: Improved agent acronym

We name our Improved agent *AGRA* ... (Taj Mahal's location)

Always Get Regressions Accurately.

Bonus: Why is linear model a bad idea?

Using a linear model is a bad idea for two reasons:

- The function will be unbounded, giving output values from $-\infty$ to $+\infty$. For an image, we need an output bounded between 0 and 255.
- It overlooks features such as edge detection and shapes.

Bonus: Using a ML framework

We use sklearn.neural_network.MLPRegressor from scikit-learn to colorize the image now. MLPRegressor is a Multi-layer Perceptron supervised learning regression algorithm and uses L2 regularization, avoiding overfitting by penalizing weights with large magnitudes.

We use the following default parameters for our MLPRegressor():

- hidden_layer_sizes = (100)
- activation = 'relu'
- solver = 'adam'

'relu', the rectified linear unit function, returns f(x) = max(0, x)
'adam' refers to a stochastic gradient-based optimizer proposed by Kingma, Diederik, and Jimmy Ba

Given the following data:

- color_image_training_r_data
- color_image_training_g_data
- color_image_training_b_data
- bw_image_training_data
- bw_image_testing_data

We first use sklearn.preprocessing.MinMaxScaler to scale our data by calling scaler.fit_transform(data). This transform features by scaling each feature to the given range 0 to 1.

Now that our data is preprocessed and ready, we fit our MLPRegressor and predict on bw_image_testing_data to produce a colorized image.

- R_model = fit(bw_image_training_data, color_image_training_r_data)
- G_model = fit(bw_image_training_data, color_image_training_g_data)
- B_model = fit(bw_image_training_data, color_image_training_b_data)
- R_predict = R_model.predict(bw_image_testing_data) * 255
- G_predict = G_model.predict(bw_image_testing_data) * 255
- B_predict = B_model.predict(bw_image_testing_data) * 255

We combine the r, g, b values to produce the following colorized image:

Figure 13: MLPRegressor colorized image

- Left: color_image_training
- Right: MLPRegressor colorized image of bw_image_testing

Visually, the MLPRegressor perform very well, with a few blemishes here and there. The green grass got colored a mix of both green and the grey shade, unlike the uniform green on the color_image_training side of the picture. There are a few blemishes on the Taj Mahal itself, where some parts got colored cyan. To find the difference between the original image and the output image we find the Mean Square Error:

MSE: 67.43688399569523

In terms of MSE, our MLPRegressor beats both the basic and improved agents.

Figure 14: Original image

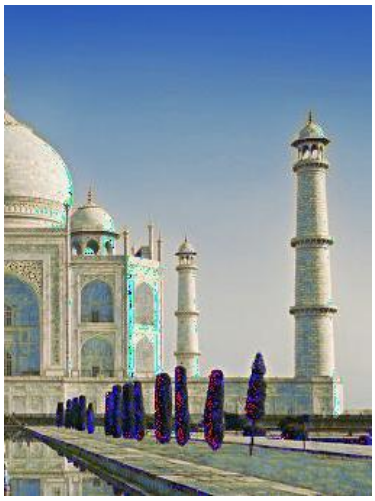

Figure 15: Colorized image

Addendum

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: Mustafa Sadiq