CS 520 - Colorization Report

Sriya Vudata (sv520)

December 14th 2020

Figure 1: cuteredpanda.jpg

1 Introduction

In this project, given an image that is grayscaled, we were required to implement an AI that could predict the original colors in the image. The following report details two approaches to the problem, one being a basic agent that limits the color palette to 5 representative colors and uses classification to predict the original image, and the other being a neural network that uses regression to predict the original image. For this project, I used the 120x120 pixel image in Figure 1.

2 Implementation Details

This section will discuss how the project was implemented in code. The implementation is in Python 3.7. The code itself is split up into several files:

- `colorization.py`: Run this to actually execute the project. This will prompt the user for a path to the image, which in my case is just `cuteredpanda.jpg` from the project root. The user is then prompted with the option to run either the basic agent or the improved agent. This file also contains all of the code necessary to run the basic agent.
- `analysis.py`: This contains the function for mean squared error, which is used to assess how close a model's prediction is to the actual image. The function `pix_diff()` does this.
- `neuralnet.py`: This contains the classes `Layer` and `Node` that are used in the neural network.
- `improved.py`: This contains the functions used to implement `improved_agent()`, which will all be discussed in further detail below. The improved agent implements a neural network using logistic regression to predict the right half of the image.

2.1 Common Code

The following functions implemented were used for both agent. More info is in the comments for each function.

- `grayscale()` This simply converts a given image into grayscale, using the formula specified in the project description: $\text{Gray}(r, g, b) = 0.21r + 0.72g + 0.07b$.
- `pix_diff()` Given the original image and the predicted image, this will compute the mean square error between the two. This is used to analyze the performance of the improved and the basic agents. The mean square error is given by the equation $mse = \frac{1}{n}\sum_{i=1}^{n} |f(\vec{x}) - \vec{y}|^2$, where n is the total number of pixels, $f(\vec{x})$ is the predicted color vector (r, g, b) given that $\vec{x}$ is a vector of 9 pixels, and $\vec{y}$ is the actual (r, g, b) value of the pixel. As $f(\vec{x}) - \vec{y}$ is a vector, doing the dot product of itself, ie $\vec{v} \cdot \vec{v} = |\vec{v}|^2$, gives the square difference.

The amount of time in seconds it took to run the model is also outputted along with the mean square error. My code also makes use of the libraries `numpy` for simple and efficient vector and matrix calculations, and of `matplotlib` to display the figures of the changed images. Otherwise, all other code relating to the models is from scratch.

2.2 Basic Agent

This section will discuss how the basic agent was implemented. Most of the code relating to the basic agent is in `colorization.py`. The basic agent showcased unsupervised learning, using the k-mean algorithm for recoloring the image using 5 colors and training, and the nearest-neighbor algorithm to predict the color of a pixel based on the 8 gray pixels around it. Essentially, the algorithm attempted to predict the color of the pixel based on how the data is already structured and based on what category of representative color the pixel falls into.

As per the project instructions, I implemented the basic agent by running k-means clustering on the colors that were present in the left half of the original image, using $k = 5$ for 5 representative colors, and recolored the left half of the original image. I then implemented the nearest neighbor algorithm by splitting the right half gray image into 3x3 patches with a stride of 1 pixel and finding the 6 nearest 3x3 patch neighbors in the left half of the gray image. The one change I made for optimization was splitting the left half of the image into 3x3 patches and comparing the testing data's patch to each of the training data's patches, rather than creating a 3x3 patch for each and every pixel in the training data. The final output is the original image with the left half recolored using k-means, and the right half predicted using the nearest-neighbor algorithm. In my implementation, the user can specify a value for k (the number of representative colors) and for n (the number of nearest neighbors to choose from). For the purposes of this project, I used $k = 5$ and $n = 6$.

When the user specifies running the basic agent, the function `basic_agent()` is called. First, training is done on the left half of the image by calling the `kmeans()` algorithm. This randomly selects k initial pixels from the left half of the original image as the initial centers for the k clusters, sorts each of the pixels into the cluster who's euclidean distance from the center to that pixel is smallest (using the function `sort_clusters()`, and re-calibrates the centers by taking the average of all of the pixels in each cluster. In the code, this is done in `new_center()`, where given a cluster set c, the sum is computed by summing all $pixel = (r, g, b)$ in that cluster, and dividing this by $|c|$. This is repeated until the difference between all of the previous centers and their current centers, taking the sum of this vector, is equal to 0, meaning that the centers have converged.

Finally, once convergence is hit, the centers are returned as the list of representative colors, as is a dictionary containing key, value pairs of the for $(row, column) : index\_of\_rep\_color$, where the $(row, column)$ is the index for a pixel in the left half of the original image and $index\_of\_rep\_color$ is the corresponding index in the representative colors list of which color the pixel is clustered with. This dictionary is later used to recolor the image according to the representative color it was clustered into.

For testing using the nearest neighbor algorithm, the main functions used is `nearest_neighbor()`, which returns a dictionary for the right half of the image of the form $(row, column) : index\_of\_rep\_color$ with the same specifications as above. As mentioned above, for each gray pixel in the right image, a 3x3 matrix is constructed consisting of the 8 pixels around the given pixel, and the n most similar patches in the training data are found by using the euclidean distance between the patches. Out of those n patches, the representative color for the middle pixel from the testing data is chosen as per the instructions in the project. As mentioned in a discussion post, this agent does not take into account the edge pixels for simplicity sake, so all of the edge pixels have a color value of the first representative color in the list. For this project, I used $k = 5$ and $n = 6$, and also played around with k to find the best number of representative colors to pick, the results of which will be discussed below.

2.3 Improved Agent

This section will discuss the coding implementation of the improved agent, and in the later question I will further discuss the specification of this solution. For this project, I treated converting from gray to color as a regression problem, and thus implemented a neural network with logistic regression.

In `neuralnet.py` is the representation for `Layer` and `Node`, where `Layer` represents a hidden or output layer in the neural network and contains a list of `Node` objects and how many inputs go into the later, and `Node` represents a single node in a layer of the neural network. `Node.out` keeps track of the output of that node, `Node.w` is a weight vector or matrix for that specific node of size $num\_inputs + 1$ (the plus 1 accounts for the bias w_0), and `Node.derivative` is the slope computed for that node, which is used later in backwards propagation error. For this project, I used the same approach as the basic agent in taking information from the 8 surrounding pixels for each pixel. I created a simple neural network with three layers:

1. The input where $\vec{x}$ is a 1x9 matrix consisting of the target pixel and its 8 surrounding pixels (as in the basic agent, edge pixels are not considered and are automatically colored the (r, g, b) value for black). As the (r, g, b) values are original between $[0, 255]$, I divided each of the values by 255 to scale the colors down to between $[0, 1]$.
2. A hidden layer, where each node takes 9 inputs from the previous layer, and in total has 20 nodes representing different features.
3. The output layer, which has a total of 3 nodes (one for each value (r, g, b)) and each node takes in 20 inputs from the previous layer.

For training the model, I implemented stochastic gradient descent and simply did forward propagation to attain a result. I used the sigmoid function, $\sigma(z) = \frac{1}{1+e^{-z}}$, where $z = \vec{w} \cdot \vec{x}$ with $\vec{w}$ being the weight vector with the size being the number of inputs, and $\vec{x}$ being the inputs. In my code, the bias w_0 that is apart of $\vec{w}$ is separately added to the dot product of $\vec{w}$ and $\vec{x}$, as x_0 is assumed to be 1, and the initial values for $\vec{w}$ for each node is randomly assigned between $[0, 1]$, with the weight vector being updated every step using backwards propagation. e, which

represents the epoch or the number of iterations to train the model for, as well as α, which is the step size or learning rate, are both specified by the user. For this project, I used $\alpha = 0.1$ and $e = 72,000$. 0.1 is a typical rate to use for α, and 72,000 is used for e to account for the number of pixels in the image multiplied by 5, ie 120x120x5.

For debugging purposes and to see how good my model is, I have the improved printing out the mean square error between the predicted color and the actual color of the pixel for every 1000 steps/data points that it does training with.

3 Part 1

1. How good is the final result for the basic agent? How could you measure the quality of the final result? Is it numerically satisfying, if not visually?

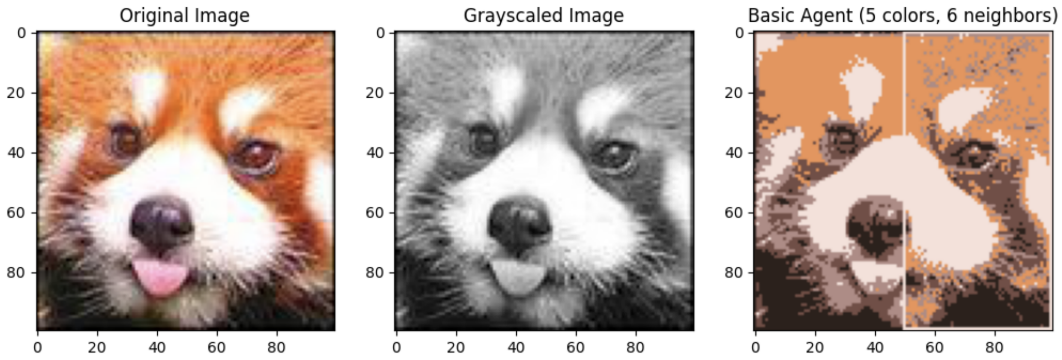

Figure 2: Basic Agent results for k=5, n=6.

Visually, from a purely subjective perspective, the basic agent actually looks quite good for this picture, as if a filter was applied to it. The left half of the image was exactly recolored according to the representative color that a specific pixel was clustered with, while the right half of the image was predicted using the nearest-neighbor. However, you can also see clearly in the right half of the image that the recoloring was not perfect, with random specs of color in places where the color should be uniform, such as in the upper right corner of the image. So visually, the image summarizes the colors relatively well using the limited 5-color palette, but is missing other aspects of the image such as depth and complexity.

Numerically, it is difficult to assess how "good" an image is, since the pixels are all discrete values, so I used the mean sum error function as described above to assess how different the original image is from the basic agent, where $mse = \frac{1}{n}\sum_{i=1}^{n} |f(\overrightarrow{x}) - \overrightarrow{y}|^2$. Since the rgb values of a pixel range from $[0, 255]$, the highest that the error can be is about $255^2 = 65,025$. The table below shows values for MSE, which were relatively on the lower side than 65,025. This can be used when comparing against other models, to see how close the actual image is to a model's prediction.

I also timed how long it took for the image to output as a means to measure how computationally expensive is the agent. In general the basic agent with k=5 and n=6 took between 2-3 minutes to output, meaning this agent is quite computationally expensive.

The following table shows the results for three runs of the basic agent for k=5, n=6.

Iterations until convergence	Time (s)	MSE
25	142	4152
28	151	3978
26	139	4072

2. Bonus: Instead of doing a 6-nearest neighbor based color selection, what is the best number of representative colors to pick for your data? How did you arrive at that number? Justify yourself and be thorough.

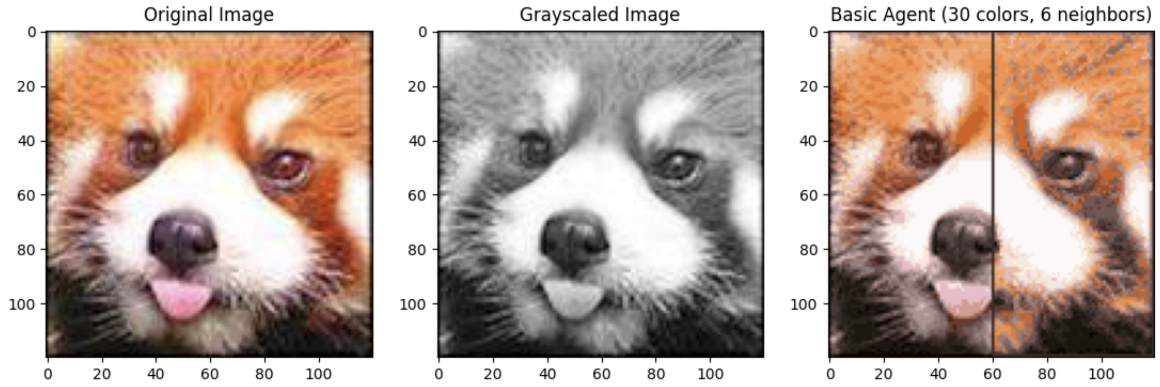

Figure 3: Basic Agent results for k=30, n=6.

From my modification of the algorithm, it looked like choosing from between [1,10] nearest neighbors did not make too significant an impact on the final output. So, I started varying the number of representative colors and stopped when the MSE was much smaller than the original and the picture visually looked very close to the original. 30 appeared to best represent the picture. I came up with this after testing a range of numbers from 5 to 32 with increments of 3, of which has the results in the table below. 30 seemed to be a sweet spot between 29 and 32 where it looks as though every color present in the original image is represented. As Figure 3 shows, the left half of the image is very close to the original image, while the right half of the image is also close and detailed, with a few random specs of color, which are less noticeable than $k = 5$.

Reasonably, the best number of representative colors appears to be close to the number of colors that are present in the original image. However, this is computationally quite expensive (this image took 6 minutes to output, as opposed to the normal 2-3 minutes), and you are more likely to run into the case with ties between neighbors when running the nearest neighbor algorithm. As can be seen below, the higher numbers >20 over all required more iterations before they final converged when running k-means. Overall however, the MSE decreased as the value for k increased.

k	Convergence	MSE
5	24	4152
8	33	3978
11	28	3384
14	39	4072
17	74	4134
20	79	3744
23	61	3529
26	62	3281
29	69	2983
32	68	2535
30	68	2557

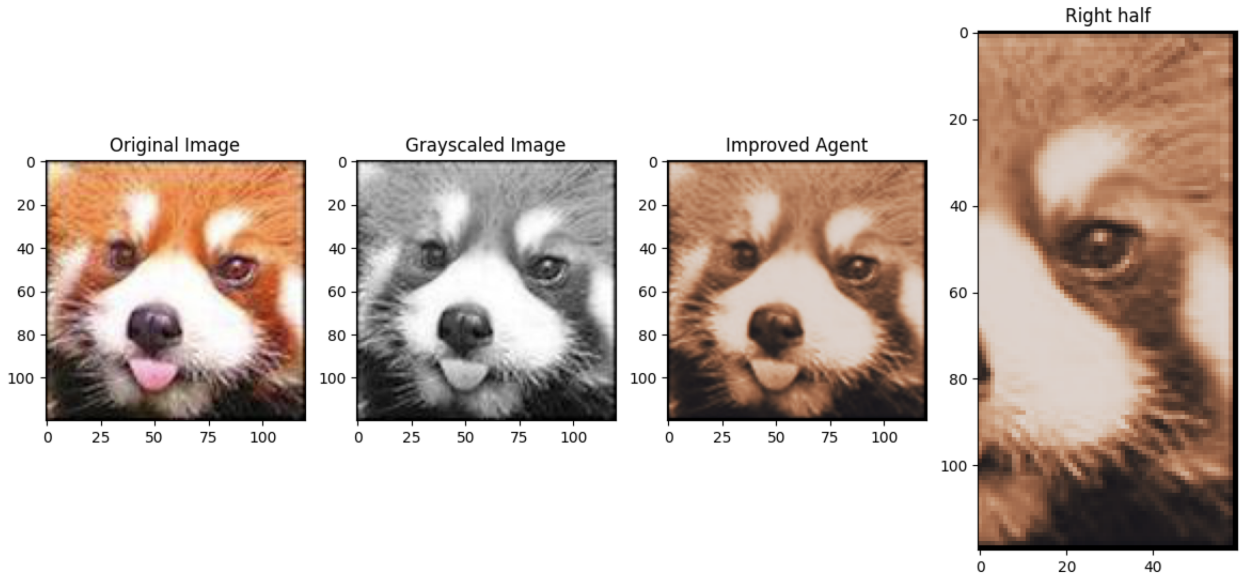

Figure 4: Improved agent results for $\alpha = 0.1$, $e = 72000$

3. Specification of solution, including input space, output space, model space, error/loss function, and learning algorithm.

   - **Input Space:** X, which represents a matrix representing the grayscaled image, where each $x \in X$ is a value between $[0, 255]$.
   - **Output Space:** Y, which represents a matrix representing the predicted colored image, where each $y \in Y$ represents a pixel vector of form $y =< r, g, b >$ that is outputted from the model f used, such that $f(x) = y$.
   - **Model Space:** The neural network, where the output at the time t starting at $t = 0$ to $t = 2$ and for node j is $\vec{out}_j^t = \sigma(\vec{w}^{t-1}(j) \cdot \vec{out}_{t-1})$, where $\sigma(z) = \frac{1}{1+e^{-z}}$, $\vec{w}^{t-1}(j)$ is the vector of weights on the inputs to node $\vec{out}_j^t$, and $\vec{out}_{t-1}$ is the output vector from the previous

layer, used for all j nodes in the current layer.

- **Loss Function:** The loss of this model is computed by computing MSE at the final layer, where $mse = \frac{1}{n}\sum_{i=1}^{n} |f(\overrightarrow{x}) - \overrightarrow{y}|^2$, n being 1 in this case because the output is one 1x3 vector, which is generated once for a datapoint in one step of the training algorithm.
- **Learning Algo:** This used logistic regression with stochastic gradient descent for learning, roughly of this algorithm:
  (a) Set $\alpha = 0.1$ for the learning rate.
  (b) Randomly initialize the weights $\vec{w}$ for each layer of the neural network.
  (c) For e iterations, choose a random pixel in the left gray image, and construct the 1x9 vector $\vec{x}$ for this pixel using the surrounding 8 pixels in the image. Run $f(\vec{x})$ to get the predicted result. Update all of the $\vec{w}$ for each node at each layer, using back propagation, based on how the network performs at iteration k by doing: $\vec{w}(k+1) = \vec{w} - \alpha[f_{\vec{w}_k}(\vec{x}) - \vec{y}]\vec{x}$, where $\vec{y}$ is the actual pixel's color.
  (d) Repeat.

  For back propagation, the first derivative computed is the final output $\vec{out}_2^j$ for each node j (which is 3). This is the derivative of the sigmoid function, which is $\sigma'(z) = sum((1.0 - z) * z)$, in this case $z = \vec{out}_2^j$. This value is propagated to the previous layer, which computes as its derivative for out_j^t as $\vec{w}_j^t * derivative(out_{t+1})$, which is finally multiplied by the derivative from $t = 2$, the final time.

4. How did you choose the parameters (structure, weights, any decisions that needed to be made) for your model?

   The number of hidden layers (in this case 1) was chosen out of convenience and simplicity, as this seemed to be the optimal layers to output an image somewhat better than the basic model. Using $\sigma(z)$ for the activation function was optimal, as the output of this function is a value between $[0, 1]$, which can easily be mapped to an rgb value between $[0, 255]$ and requires minimal processing.

   The weights were initially all initialized to a random real number between $[0, 1]$. At each node, the number of weights was $|\vec{x}| + 1$, where $|\vec{x}|$ is the size of $\vec{x}$, and $+1$ accounts for the bias value w_0. The number of weights should include the number of components in $\vec{x}$, so that multiplying $\vec{x}$ and $\vec{w}[:-1]$ (ie without the bias) results in a scalar value that can be used as z in e^z, thus giving a number between [0,1]. Between the first and second layer for each of the 20 nodes, $\vec{w}$ is of size 1x10, for 9 inputs (gray pixels) and 1 bias value. To the final layer for each of the 3 nodes, $\vec{w}$ is 1x21 for 20 inputs from the previous nodes and 1 for the bias again.

   The size of the outputs for each layer correspond to the weight vector. So for out_0^j, for each node j this was a vector of size 1x9 for the 9 gray pixels being inputted. For out_1^j, for each node j this was 1x20. Finally for out_2^j for each node j, this was a 1x3, where each node corresponds to an rgb value.

   As mentioned in the implementation above, the epoch was chosen based on the size of the pixel and to avoid overfitting, while $\alpha = 0.1$ was chosen as the typical learning rate.

5. Any pre-processing of the input data or output data that you used.

   To use a 3x3 patch of the gray image, the representative 3x3 matrix for each pixel was found. This was then flattened to be a 1x9 vector to be inputted into the neural network.

As mentioned, the activation function $sigma(z)$, and this neural network in general, outputs a value between [0,1]. So to make sense of the data, the inputs that were originally between [0,255] were pre-processed, by dividing all the values of the pixels by 255 to scale them to [0,1] and keep consistency across the model. Then, once there was a final output, this was processed again to be between [0,255] by simply multiplying by 255. Thus, we are able to make sense of the output of this model.

6. How did you handle training your model? How did you avoid overfitting?

   As mentioned, I used stochastic gradient descent to train the model. In other works, for each step/iteration, I chose a random pixel from the training data, in this case the left half of the gray image. Using forward propagation where the input would go through the layers of the network, an predicted color is generated. The weights for the entire neural network are then updated based on the loss between the actual and predicted, starting by taking the derivative of the loss function at the final layer of the neural network for each of the three nodes, and propagating that slope back into the other layer's nodes, and repeating until all layers and nodes are included in the final slope value, which is used to update the $\vec{w}$ vectors for each layer at each node. This training went for some number of iterations until the error was minimized.

   I attempted to prevent overfitting by choosing the most ideal number of epochs. Ideally, the model should randomly training with every data point (pixel) that is in the training data a few times, so it seemed reasonable to say that the number of training iterations should be (size_of_image)*5. Also shown in the table below, increasing the number of iterations did not really change the outut of the model, so making sure not to overtrain the model was achieved by choosing a smaller number of iterations, and by making the data points that were being trained random to account for variation.

7. Evaluate the quality of your model compared to the basic agent. How can you quantify and qualify the differences between their performance? How can you make sure that the comparison is "fair"?

e	Time (s)	MSE
72000	26	3256
75000	27	3479
150000	52	3199

   Visually, the output of this model looks more like a redish-filter was applied to the gray-scaled image, so in terms of color variation, the basic model has higher color quality. However, in terms of the quality of depth and computation, the improved image appears to perform better. The basic model uses classification and this model uses regression, so its a little challenging to truly compare the two. However, we can fairly compare the differences in performance of these two by looking at the time it took to output, and the MSE of the two, ie how different each outputted image is from the original image. As shown in the trials run, the MSE for the basic agent was around 4000, while for the improved agent was around 3000, so it seems that the improved agent captures more of the data of the image than the basic agent. The time it took to run the improved agent was also much smaller than the basic agent and more computationally efficient, as the basic agent required a comparison with every training dataset and keeping track in memory of how each training point was categorized, while the improved agent just required using a mathematical model and much

less memory usage. Overall numerically, the improved model seems to perform better than the basic image if not visually.

8. How might you improve your model with sufficient time, energy, and resources?

   Given sufficient time, energy and resources, one way this model can be improved is by using a neural network for classification rather than regression. It was clear from this project that for image processing in this manner, treating this as a classification problem rather than a regression problems yields a more satisfactory result. If going in the direction of getting a more visually satisfying image, the basic agent wins for the variety of colors. So, this could have been improved by using the same k-means algorithm to choose colors that best represent the image, and then instead of using the nearest-neighbor algorithm to predict the color, we can use a neural network that implements multi-classification to decide with representative color the gray pixel is likely to be. This combines the computationally less expensive and more complex nature of using a neural network with the benefits that came from using k-means. It would also be interesting to use a library like `TensorFlow` or `skilearn` to implement a more complicated neural network with regression, maybe with more layers and nodes at each layer.