Ramon Pena

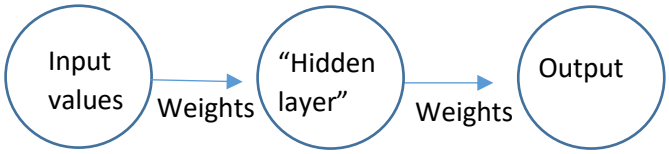
1. -There are 255*255*255 = 16,581,375 possible color values since each color red, green, and blue can go from 0 to 255.
   -There are 39 possible gray values and 39^9 possible 3x3 gray pixel patches since each patch has 9 gray values with each gray value having 39 possibilities.
   -A 3x3 pixel patch is not enough information to determine the center color since an image doesn't necessarily have a linear distribution of colors i.e. A blue sky compared to an image with a wide range of mixed colors.
   -The context that might be enough to associate a shade of gray with a color would be on images that are plain throughout, and doesn't have a wide variety of colors such as an image of just a blue sky with clouds.

2. -It might be a good idea to reduce the color-space because it would decrease the amount of colors the algorithm has to guess for, which in turn might make it more accurate in its predictions.
   -A good coloring algorithm should be able to pick the color green or red with the highest probability of what the color should be.

3. My algorithm when ran first loads all the necessary files, input.csv, data.csv and color.csv through a loadData(). The loadData() returns an ArrayList of double arrays where each array has 9 grayscale values, or 3 rgb values depending on which file it was called on. This is then fed to a scaleArray() method to scale all the values to be between 0.0 and 1.0. In order to train on the data, it iterates through the arraylist and passes the double array of the values to a train(). The train() takes in the input values, the expected output and three neurons, red, green, and blue. These neurons are created first with just the number of inputs n. This is done so each neuron can store its weights and errors after each iteration. The neuron takes just the number of inputs because once a neuron is created, it creates an array of weights of size n with random values between 0.0 and 0.2. This train() returns an array of weights in the order (r, g, b) for each color. This then writes these values to a file. This is done, so each time the program is run, if the file exists in the directory, then there is no need to train again. In the train() a bias is added to the beginning of the array before being passed on to the neurons.

The neuron contains a train(**input, expectedOuput**), update(**output_sum, error**), hiddenUpdate(**output_sum, expectedOutput**), error() and helper methods. The train() gets the sum of all the inputs times their weights. This is information is then passed to the hiddenUpdate()

where it adds a bias and given weights. Then, it gets the sum and predicts the output value by using the activation function, Sigmoid(sum). The error is then calculated, its weights are adjusted and the error is passed to the update(), where the weights of the grayscale values are adjusted. What this is trying to do is that the hiddenUpdate() is trying to simulate a hidden layer with two inputs.



This is done for each of the inputs that is passed to it. This is the "learning" part of the network. Each neuron returns its respective weights and saved in an array to be returned. To test the network a testnetwork(input, numberOfTestingValues) is called. Input is just the grayscale values and numberOfTestingValues is the size of the input * 0.2. So, the bottom 20% of the data is used for testing, and the top 80% is used for training. This split was done for simplicity. Each of the respective color's weights are loaded from a "Weights.txt" file.

Then for each 9 input grayscale, the sum of the input * weight is calculated and then passed to the Sigmoid activation function. This predicted value is then scaled to be between 0 and 255. These results are saved in a "Testing_Results.csv". To predict, the same thing is done but with data.csv. This file is saved as "Data_Results.csv".

Some difficulties I had to overcome were correctly implementing the formulas and function given to us in class. To do this, I had to first understand the notation being used and how each variable was related to each other. Once I understood this, implementing it became much easier. I also had to figure out a way to add a hidden layer to my current implementation. This was solved by the hiddenUpdate(). One difficulty I wasn't able to overcome was the accuracy of my network. As it stands now, it's not very accurate at all. I tried resolving this by adding an extra layer, which didn't help. I tried getting the overall error, and stop the training once a threshold was reached. This made the network under predict all the values for a multitude of thresholds. Because of this, I added up not using thresholds and let the network train until it was done with the training dataset. Since I decided to do the bonus, coloring the image was difficult since for every patch of 9 pixel values, there was only one rgb value. To

resolve this, I colored the image the same way I converted it to grayscale. I took 3x3 pixel patches and gave it one color and repeated this process.

4. I can use the provided data to evaluate how good my algorithm is by doing an 80-20 split. This means that it trains on 80% of the data and checks it's accuracy on the other 20% of the data. Some type of validation that can be done is taking the square error after each prediction and plotting it over time. Doing this I found out that my algorithm performs very poorly. The average error deviation for red was 44.87, green was 26.41 and blue was 37.95, which for predicting in my opinion is bad since the deviations were so high.

5. See Data_Results.csv for the computed values.

6. Overall my algorithm performed well with the green values than with the other two colors. Looking at the error of the greens compared to the blue or red, it is much smaller, this is in the context of "being less wrong". It also performed well predicting the shade of the dominant color. This means that while the color of that specific pixel is not exact, it will at least be a shade, lighter or darker, of the true color of the pixel. In the overall context, my algorithm performed poorly in its predictions. This could be due to poor performance during the training. I did notice that one of its weights was always high, above 1. I tried adding various bias's in order to lower this weight without affecting the others, but then this led to my predictions being all over the place. With greater time, I would've been able to improve its performance by that I would've came up with a solution to tweak that weight without affecting the others. Also, I could've restructured my algorithm, so I could add hidden layers to it. This means changing the body of the neuron, more specifically, its constructor. This would allow me to create a neuron with more specific parameters such as "boolean hiddenLayer", this would tell me if the neuron was being used in the hidden layer or if it was an output neuron. Another thing I could do would be to change how the weights are updated, such as picking a pseudorandom weight and updating that, instead of updating all the weights after each input.

7. What would happen is that my algorithm would more or less color the tiger with some shade of green, while some of parts of the tiger will be either a lighter or darker version of its true color.

Ramon Pena

Bonus) My algorithm visually doesn’t do very well. It did not perform as I thought it would when trained on the given input and color values. Visually my algorithm loses the subject of the image. The image has consistent shades of a lavender color with vertical lines. This could be either due to how I colored the image. The images that were used to test this were “duck with glasses.png” and “tiger.png”. Their grayscale counter parts are “GrayDuck.png” and “GrayTiger.png”. The output of my program after coloring them are “ColoredDuck.png” and “ColoredTiger.png”.