Adding a Splash of Color

Team Members: Vaibhav Malik and Jonathan Diaz

Contributions: We worked on the majority of the project together sharing ideas, methods, approaches and algorithms. Jon built his neural network in C and Vaibhav in Python. We wrote the report together and helped debug each other's code.

In order to represent the coloring process in a way that computers can interpret we need to represent the color of a pixel numerically. A natural representation is the vector of RGB values for colored pixels or the greyscale value for grayscale pixels. When choosing spaces to map between we considered the complexity of the neural network as well as its ease of coding. We figured we wouldn't be able to map from the grayscale values of EVERY pixel in the image to the color values of every pixel because training the model would take a lot of time to update all the weights as well as a lot of data to accurately train the model. We also chose to use a regression method rather than a classification method as we thought it would be easier to code. The maps we have chosen to consider are ones from a set of grayscale values for several pixels to a single RGB vector of a specific pixel. Initially we were mapping from 25 grayscale values in 5 by 5 grid to the RGB values of the center pixel but due to performance constraints we had to scale back our input to take in a 3 by 3 grid instead. Based on the hints given in the assignment instructions we created a preprocessing function for training images. The first step in this function is to add padding to every side of the image. This allows every pixel in the image to be at the center of the "window" that is created around the pixels. The padding pixel colors were the same colors as the colors in the original neighbor pixels. Additionally, every RGB integer array was converted into a float array because online research pointed out that deep learning image operations are more efficient and precise with use of float RGB values instead of integers. Each RGB pixel array was divided by 255 before being input into our neural network. The RGB pixel array was reconstructed for outputting a colored image by taking the output RGB float array and multiplying it by 255.

To get data to train our model we would take a single, color image and convert it into a grayscale image. This gave us several 3 by 3 pixel grids to train on from a single image.

To evaluate our model's accuracy, we used the mean square error of the predicted RGB values of the center pixel vs the actual RGB values of the center pixel. We tried to find a way to programmatically tell if one set of RGB values was visually close to another set but were unable to find such a method. Furthermore, since we are only looking at 9 pixels at a time to determine the color of the center pixel we didn't think there was a good way to tell if the computers prediction was visually good as opposed to numerically good besides the aforementioned visual color closeness.

To train our model we used a vanilla neural network and back propagation. We started with 9 input nodes feeding into two hidden layers with 6 nodes each including the bias and finally leading into a 3 node output layer and using the sigmoid function as our

activation function on every node. We would take every 3 by 3 grid of pixels feed it forward through our network and then calculate the loss in the manner specified in the previous section. Then using this loss we calculate the gradient at each node and then did stochastic gradient descent to update the weights. To determine convergence after training our model on the entirety of a picture we would feed the entire black and white photo forward again this time saving the predicted RGB values of each pixel and using it to create a new color image. We then compared how visually close the image was to the original. When we first started checking how good our program was we noticed it was just applying a colored filter corresponding roughly to the average color of each pixel over the original image. We attributed this to the model overfitting on the colors in our one image so we figured we could rectify this by training it on multiple images. When we started training it on multiple images the filter phenomena became much stronger and it became very difficult to tell what the original image was though you could see the outline of some trees in the beach image if you looked hard enough. Then in an attempt to have something that colors at least one photo well we decided to embrace overfitting and try to get the model to learn one picture really well. To achieve this we had the model train on all the pixels in one image repeatedly but the filter effect remained.

At this point the two of us took two different approaches to try to get something that looked good. One approach was to replace the sigmoid activation function with a rectified linear unit one and the other approach was to try and beef up the neural network to 4 hidden layers with 11,11,6,6 nodes with the biases included. The logic of using rectified linear units over the sigmoid function was to avoid vanishing gradients as maybe the weights at the beginning of the neural network weren't getting updated enough and that was having a significant impact. Also using rectified linear units is less computationally intensive than using the sigmoid function so it allowed us to train the model faster and give it "better" training. The logic behind beefing up the neural network was that maybe there weren't enough nodes per layer or the layers weren't deep enough to allow complex relationships between the input nodes to form so adding to both of these would ideally solve that problem. The ReLu implementation originally yielded promising results, but it became clear that it needed much more training data than a neural network that uses the sigmoid function. Additionally since the ReLu-based neural network was trained in Python, the training image needed to be changed to a smaller size. After 100 generations of training on the same image of a strawberry, it seemed like the neural network was starting to learn the color scheme. However, Python is not computationally strong enough to train on the number of examples required to make the ReLu neural network accurate.

Results:

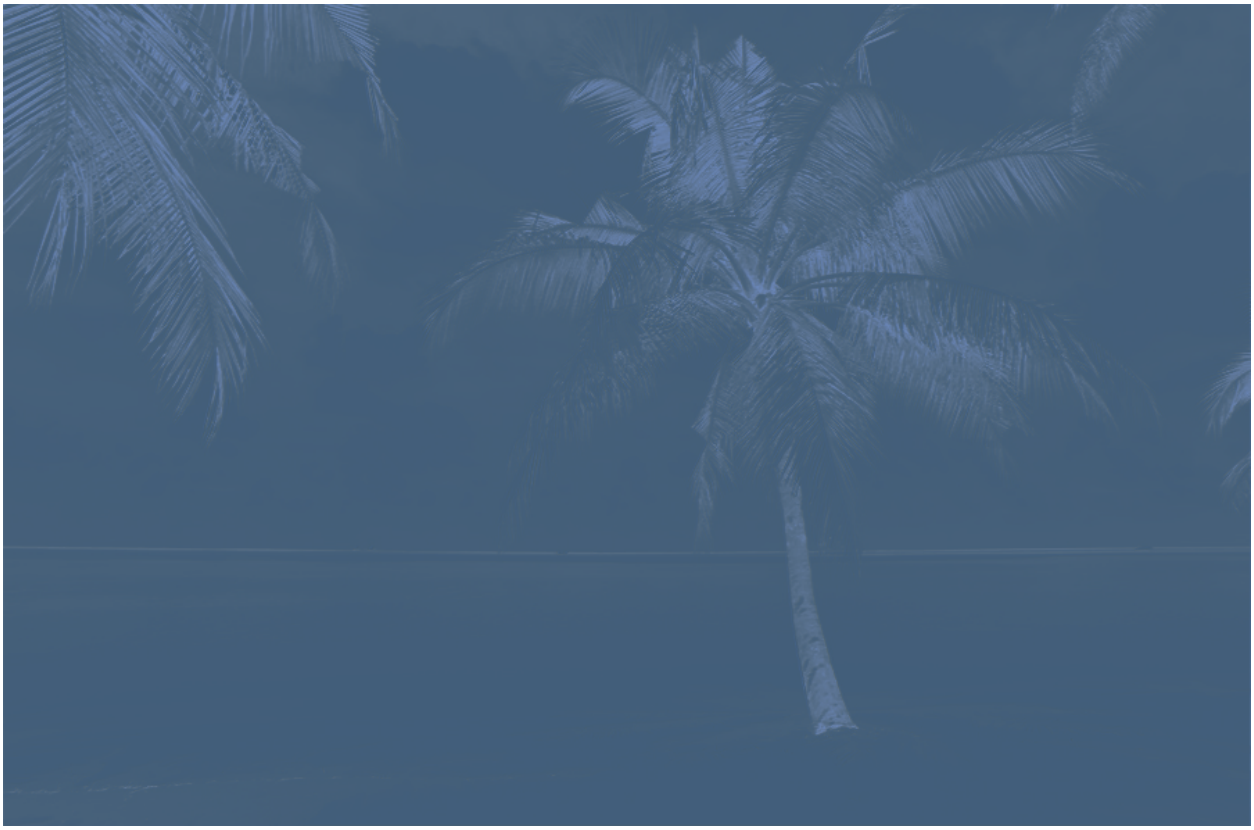
Original Black and White Image

Original Colored Image

Colorization of Black and white Image when only trained on this specific image 200 times

Colorization when trained on 3 images (in the colorization of the other two images it is impossible to discern any shapes so we left them out)

Colorization after training on the other beach 200 times.



Colorization After training only on the Tiger.

Greyscale strawberry image:

Colorized strawberry image trained over 100 generations using ReLu activation function:

We can tell from looking at the images that our final program turned out to be not that great. First, the only thing it's good at is applying a colored filter to a specific black and whte image that it gets trained on, and it can also manage to not completely screw up an image similar to the one it was trained on. Beyond that it generalizes terribly to images completely different from what it is trained on. It's failures do make sense in that the filter it applies tends to be a similar shade to the predominant colors of the image it was trained on and it's also not surprising that when it is trained on multiple images it has a hard time coloring any of them as it is likely too small to learn all the complexities of all the different shapes it could see and instead gets pulled in a random directions eliminating all of its learning. Another consequence of being too small and weak would then be its inability to generalize to completely different images because it will have no experience with the completely different shapes present in the new image.

If we had more resources we would try and build two convolutional neural networks both much deeper and fatter than the one we built. If Python was computationally stronger, the ReLu neural network could have been trained on more examples and ,we think, would have produced accurate results. One of which would be capable of taking an entire image as input as opposed to small 3 by 3 chunks and using all of that context to color the entire image. We could then use a lot more time to train this network on billions of different images so that it could learn to identify broad things like tigers or trees or clouds and then using that context would much more accurately color images. We would use the second network to take in a color image and try and detect if the image is real or generated. We wouldn't want to give it the grayscale image to compare it to as all it would need to do is check if the conversion is perfect or not and it can potentially get very hard to fool.