CS 520: Assignment 4 - Colorizer

1. **Sri Gautham Subramani (ss2999)**
2. **Divyaprakash Dhurandhar(dd839)**
3. **Abdulaziz Almuzaini (aaa395)**
4. **Shubhada Suresh(ss2970)**

Summary:

- Collected 36 images
- Resizing to 256,256
- Finding 8 neighbors gray with one corresponding color using path of size 3 and 1 steps
- Split these two files
- Converting them to Gray for linear regression and NN
- Using rgb2lab for CNN
- Feed it to the linear regression
- Feed it to the NN
- Used Keras and Sklearn
- How did Keras uses Gradient Descent and with variant loss function
- Used a RELU , sigmoid
- What optimization we have used → we used adam

Data: Where are you getting your data from to train/build your model? What kind of pre-processing might you consider doing?

Since we are running the algorithm in our machines, we only collected 36 HD images from **Unsplash.com.** We applied multiple image preprocessing as follow: first we resize all images to a 512*512 dimension. Then, all images are converted to grayscale. So we have (36 , 512 * 512 colored images) and (36, 512*512 gray images).

For this assignment we tried different experiment in terms of how to represent our training data. Specifically, input space and output space. One approach we had applied as mentioned in the assignment description is using a patch to find the surrounding neighbors and for every row we have a 8 gray-values which will be used as inputs and the corresponding value is the rgb values corresponding to that cell. See Figure 1.

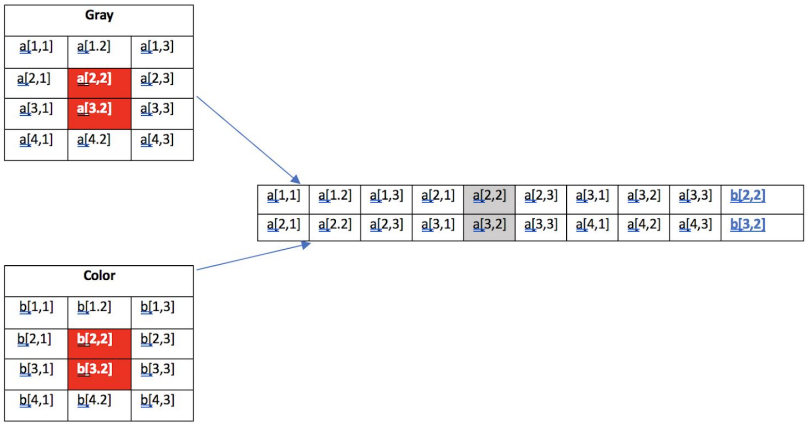


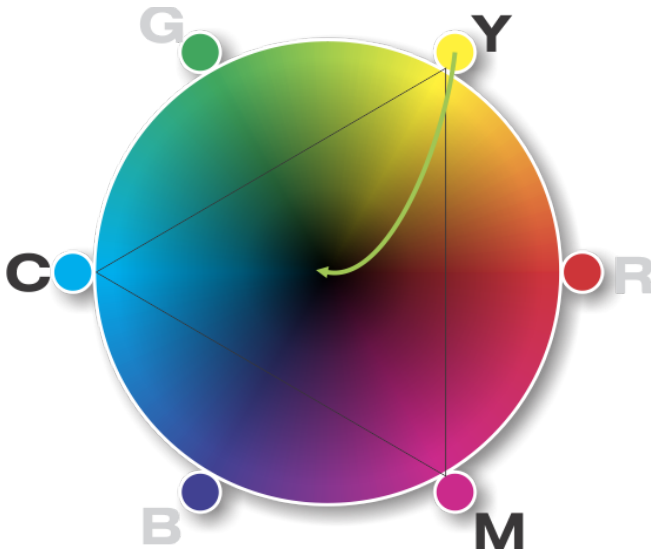
Figure 1. 8 neighboring cells

Using the previous approach, we have for only one training image 260100 rows and 9 columns for gray and 3 columns corresponding to the rgb values. Later we expand this approach to multiple images.

Other approach using all images as inputs and convert their color space from RGB to LAB (CIELAB). RGB operates on three channels: red, green and blue. Lab is a conversion of the same information to a lightness component L*, and two color components - a* and b*. Lightness is kept separate from color, so that you can adjust one without affecting the other. "Lightness" is designed to approximate human vision, which is very sensitive to green but less to blue. If you brighten in Lab space, the result will often look more correct to the eye, color-wise. In general we can say that when using positive values for the saturation slider in Lab space, the colors come out more 'fresh', while using the same amount of saturation in RGB makes colors look 'warmer'.
The premise is that one perceived color, as humans do, can not be at the same time its complementary color.
Let us use yellow as an example.

- Yes, you can have a "reddish" yellow
- Yes you can have a "greenish" yellow
- But you can not have a bluish yellow, because before it turns into blue it needs to pass through green.

Yellow is the complementary color of blue.



On the other color models you add a component, for example RGB you can have no red (0) or a lot of red (255). But in Lab you have two "complementary" colors on the same scale. This means that the ’a’ and the ‘b’ are not representing one color but an axis of complementary colors. Thus when the coefficients of the components ‘a’ and ‘b’ are assigned to zero, the result yields a grayscale image to work with. And so in our further implementations, we convert the image from ‘RGB’ to ‘LAB’, and trained the grayscale part

to predict its corresponding ‘a’ and ‘b’ components so that together the three components represent the final colored image.

Evaluating the Model: Given a model for moving from grayscale images to color images (whatever spaces you are mapping between), how can you evaluate how good your model is? How can you assess the error of your model (hopefully in a way that can be learned from)? Note there are at least two things to consider when thinking about the error in this situation: numerical/quantified error (in terms of deviation between predicted and actual) and perceptual error (how good do humans find the result of your program)

We used supervised learning to convert a grayscale image to color image. We considered simple linear regression, neural networks under supervised learning.

If we just performed using linear regression on the entire dataset, the model estimates the coefficient for the entire massive scale of input. Due to which performance of this model was not very good. To evaluate the overall fit of a linear model, we use the **R-squared** value. It is the proportion of variance in the observed data that is explained by the model. By training a model on training data, we got **R-squared** =’ ’ and higher the values better the variance explained by the model.

We built neural networks to effectively map grayscale image to colour image. We used Multi-layer Perceptron regressor. We initialize MLPRegressor with **alpha** as a regularization parameter. The higher this value, the higher the penalty for high network weights. Our neural network consists of 10 **hidden layers**. For each nodes in the initial layer,there is a connection to every other node in the second layer. In the final layer, we have 3 nodes, one for each in RGB. This again makes a fully connected structure, were each node in layer 2 is connected to each node in
layer 3 and so forth. **Activation** defines the activation function. It calculates a “weighted sum” of its input, adds a bias and then decides whether it should be “activated” or not. Popular choices are ‘tanh’ ,‘relu’ and ‘sigmoid’ function. We have used **ReLu** which is a nonlinear approximator and activates output only if its positive otherwise 0. **Sigmoid** is smooth and step like function which causes significant change in output for any small change in input.This model optimizes the squared-loss using **Adam** optimizer.

Adam is an optimization algorithm that can used instead of the classical stochastic gradient descent procedure to update network weights iterative based in training data. Adam is different to classical stochastic gradient descent. Stochastic gradient descent maintains a single learning rate (termed alpha) for all weight updates and the learning rate does not change during training. A learning rate is maintained for each network weight (parameter) and separately adapted as learning unfolds. Adam combines the advantages of two other extensions of stochastic gradient descent. Specifically:

- **Adaptive Gradient Algorithm** (AdaGrad) that maintains a per-parameter learning rate that improves performance on problems with sparse gradients (e.g. natural language and computer vision problems).
- **Root Mean Square Propagation** (RMSProp) that also maintains per-parameter learning rates that are adapted based on the average of recent magnitudes of the gradients for the weight (e.g. how quickly it is changing). This means the algorithm does well on online and non-stationary problems (e.g. noisy).

Adam realizes the benefits of both AdaGrad and RMSProp.

$$m\_0 := 0 \ \ ext\{(Initialize\ initial\ 1st\ moment\ vector)\}$$

$$v\_0 := 0 \ \ ext\{(Initialize\ initial\ 2nd\ moment\ vector)\}$$

$$t := 0 \quad ext\{(Initialize\ timestep)\}$$

$$t := t + 1$$

$$lr\_t := \ \ ext\{learning\_rate\} * \sqrt\{1 - beta\_2^t\} / (1 - beta\_1^t)$$

$$m\_t := beta\_1 * m\_\{t-1\} + (1 - beta\_1) * g$$

$$v\_t := beta\_2 * v\_\{t-1\} + (1 - beta\_2) * g * g$$

$$variable := variable - lr\_t * m\_t / (\sqrt\{v\_t\} + \epsilon)$$

Instead of adapting the parameter learning rates based on the average first moment (the mean) as in RMSProp, Adam also makes use of the average of the second moments of the gradients (the uncentered variance). Specifically, the algorithm calculates an exponential moving average of the gradient and the squared gradient, and the parameters beta1 and beta2 control the decay rates of these moving averages.

The initial value of the moving averages and beta1 and beta2 values close to 1.0 (recommended) result in a bias of moment estimates towards zero. This bias is overcome by first calculating the biased estimates before then calculating bias-corrected estimates.

Training the Model: Representing the problem is one thing, but can you train your model in a computationally tractable manner? What algorithms did you draw on? How did you determine convergence? How did you avoid overfitting?

We have used different approaches, the first one we used a Linear regression approach based on our 8-neighboring cells dataset but the result was not convincing. The image has gained slightly colors but the algorithm can't do well. See Figure 2 (a),(b)

Figure 2 (a)

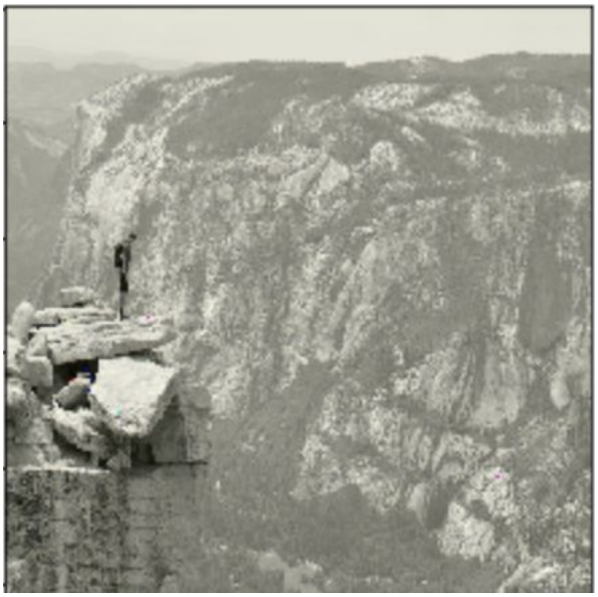


Figure 2 (b)

The accuracy was not good at all. Evaluating the prediction results in terms of the predicted and the actual the accuracy was low.

Then we applied general Neural network approach with multiple hidden layers and with different activation functions namely “relu” and “sigmoid”. The inputs for the NN is the 9 dimension and the number of nodes in each layers was a 9, 16, 32, 64, 128, 256, 128, 64, 32, 16, 3. Determining the number of nodes in each layer and how many layers we should have was arbitrarily random. We tried different number but the accuracy and the minimum loss error has not changed. For this approach, we ran the NN for 500 iterations with batch-size 32 and the maximum accuracy we have received is about 70%. It is quite better than the linear regression. See figure 3.

Figure 3. The image on the left is the actual and the predicted on the right

Figure 4. Actual color, gray image , and the predicted

Another picture was tested as in Figure 4. It was clear that algorithm still can't generalize well enough but it successfully recovered some shades of the colors such as ski, trees and the building.

Assessing the Final Project: How good is your final program, and how can you determine that? How did you validate it? What is your program good at, and what could use improvement? Do your program's mistakes 'make sense'? What happens if you try to color images unlike anything the program was trained on? What kind of models and approaches, potential improvements and fixes, might you consider if you had more time and resources?

The program was built on the idea of a simple neural network. The model was trained with one image where the image was vectorized and fed to the model as input and it was trained for 100 epochs using adam optimizer and mean square error as the loss function. When tested with another image with almost similar features, the model was able to reproduce close to real results as shown below.




The pictures on the top are the X and Y labelled data that was used to train the model and the picture on the bottom left was the test data used to predict on the model. The output of the test data is the picture on the right bottom. From the result, it is pretty clear that the model needs a lot of tuning. We can see that due to the immersion of the trees in the clouds of our train data, the clouds in the test have also been affected with patches of green color as shown. Also, there is very little presence of brown color in the Test results as the amount of inadequate feature extraction from the train data. This could be possibly because when considering pixel wise training of a model like this, the colour of the ground is highly correlated with that of the tree branch and hence the Test reproduced the ground color appropriately but failed to detect the branches properly. Also when the program was trained on a data that is completely new, the output was really noisy with random predictions of colours that only appear on the train data. This is basically because of the lack of training on the model.

Thus we concluded that the model is tuned to perform well only for pictures that have similar features as the train data. This means that the model is not generalized. Thus to improve the performance, we moved to a CNN mode implementation with more pictures to train than just using one picture.

In the CNN implementation, the main goal was to generalize different colours of data so that given a picture, the model has different range of colors to choose from. Here we took the advantage of using the LAB color space that was explained before. We convert the data from RGB to LAB using inbuilt function and the we extract only the first column of the color space which only corresponds to the grayscale component of all the pictures. We then use this data on the model and we train it to predict the corresponding 'a' and 'b' component of the pictures. Then we feed only the grayscale data of a new image and then we try to predict the other two components which are then combined together to produce the final colored image. Although the model may not be perfect, its performance can still be improved by getting more images to train on. Due to the lack of proper data, we download random images online and resized them to use for training. The drawback is the when resized some of the features are lost or some become unclear which may result in some noise in the weights. Our neural network finds characteristics that link grayscale images with their colored versions. First, you look for simple patterns: a diagonal line, all black pixels, and so on. You look for the same exact pattern in each square and remove the pixels that don't match. You generate 64 new images from your 64 mini filters. If you scan the images again, you'd see the same small patterns you've already detected. To gain a higher level understanding of the image, you decrease the image size in half. You still only have a 3x3 filter to scan each image. But by combining your new nine pixels with your lower level filters, you can detect more complex patterns. One pixel combination might form a half circle, a small dot, or a line. Again, you repeatedly extract the same pattern from the image. This time, you generate 128 new filtered images. As mentioned, you start with low-level features, such as an edge. Layers closer to the output are combined into patterns. Then, they are combined into details, and eventually transformed into a picture.

The neural network operates in a trial and error manner. It first makes a random prediction for each pixel. Based on the error for each pixel, it works backward through the network to improve the feature extraction. It starts adjusting for the situations that generate the largest errors. In this case, the adjustments are: whether to color or not, and how to locate different objects. The network starts by coloring all the objects blue. It's the color that is most similar to all other colors, thus producing the smallest error. Because most of the training data is quite similar, the network struggles to differentiate between different objects

Here are some of the results of the implementation.

After training all the images for just 1 epoch:

From this we can clearly see that since majority of the test has density of blue more than any other colour, the predictions are more aligned to blue and this shows that we need to train for more iterations to give better results.

After 10 epochs, we can see that the sky and sands are being accurately predicted which shows improvement from the base model. Now increasing the number of iterations to 500,

We can see that the colours are more natural and even though they are not the same as the original image, the model can identify features in the image and was able to give appropriate color on its own which is close to the original image. But there are still spots of shadow not colored properly and the color of the sky is not so distinct from the color of the sea. Diving even more deeper at 2500 iterations, we achieve almost state of the art accuracy as shown below

With the images being almost absolutely identical. Here are some other samples tested on the same model to verify the results we obtained.

Although these images looks almost alike, this could be because the model is overfit, meaning it is tuned too precisely to perform well for any image from the training dataset. To experiment this, we took some random images that were not used in training, resized them and used for testing. The results are as shown below,

Now the changes here may be highly influenced because the model was not treated with any picture of the same kind during the training. Hence some of the data even though is colored, is not as same as the original pictures. From the first image, most of the sky was trained to be colored blue and hence the spread of blue on the result. As well as none of the training images had sun or yellow shade on any of the trees, resulting the final image to have green leaves and no yellow backlight.

Future improvements of the model can be adding more convolutional layers so that the model need not be restricted to a fixed resolution. We can also add a GAN network at the end of the output of the CNN to have more accurate results than a normal CNN model.

References

1) https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/
2) https://www.tensorflow.org/api_docs/python/tf/train/AdamOptimizer
3) Adam: A Method for Stochastic Optimization Diederik P. Kingma, Jimmy Ba
4) https://unsplash.com/
5) https://www.codementor.io/isaib.cicourel/image-manipulation-in-python-du1089j1u
6) http://neuralnetworksanddeeplearning.com/chap1.html