Introduction to Machine Learning: Final Project

Aastha Shah, Nandini Agrawal, Reuel John

Abstract:

Our project deals with coloring black and white images in such a way that they appear natural. It was initially recommended to us that we use the Labelled Faces in the Wild dataset for the project, however we have used a similar dataset of faces but with a better pixel quality, an AI generated dataset of images of faces of all races and ages. Our training dataset comprises of 1350 images, validation comprises of 150 images and we have tested the project on test images of faces with similar compositions (a close up of a face without noisy backgrounds). We are getting training accuracy of 72.78% and validation accuracy of 71.66% on our final model on running it for 20 epochs. Increase in the accuracy with increased epochs was marginal. Through this report, we will describe our approach, the difficulties we faced, the results we achieved and the ways we can use to further improve our model.

Main Objective:

Given a grayscale image, obtain a coloured image which appears natural. Our model uses the La*b* color space and gives a mapping between the 'L' component of an image (grayscale, dimensions H*W*1) and the 'a' and 'b' components of the image (colors, dimensions H*W*2).

A Short Note on Color Spaces:

Although RGB is the most common color space, it is a daunting task to work with it as it cannot be easily expressed as a mathematical function. Further, giving a grayscale image for colorization to a model would require prediction of 3 values: R, G, and B. Thus, we are using the CIE La*b* color space mainly because the color space is perpetually linear than other color spaces, and requires just two values to be predicted for each L value: a and b. It also matches human perception closely. The L channel is for luminance, the 'a' channel controls hues between red and green and the 'b' channel controls hues between blue and yellow. As the L channel is statistically independent from the ab channels, it is very easy to extract the grayscale image (just the L channel) and our model only needs to learn the ab channels which can be expressed as a mathematical function.

Background and Common Approaches:

The image colorization has been attempted using several different approaches, many of which have been turned into commercial software. On researching for a few days, we stumbled upon a few common approaches that papers have taken:

1. User-guided approach: A few implementations of this problem have allowed the user to draw a line of colour or add input related to the image to get an idea of what area of the image should be of which colour.

   Why we *didn't* opt for this: We wanted to build an automatic colorization model without any user input.

2. Encoder-decoder: Considering images have local correlation in terms of color and grayscale value, this approach provides a simple way to scale the image down and compress local features and reconstruct the image while filling in similar colours in local neighbourhoods.

   Why we *didn't* opt for this: On trying this basic approach, we found that training was extremely slow and increasing the amount of data, epochs, and altering the layers of the model was still giving dissatisfactory results. Though sometimes results were not too bad, training took a lot of time and we thought of looking at other approaches.

3. Probability distribution: This approach either takes convolutional neural networks (CNNs) in the encoder-decoder style or just a normal CNN and builds a probability distribution of the colors for each pixel at the last layer, thus reducing the number of colours a pixel can take.

   Why we *didn't* opt for this: It was daunting for the given time period! We found another approach that we thought was more within our scope.

4. Only pre-trained model and decoder-style CNN: Some approaches took the route of classifying the object in the image to reduce the number of possible colors for pixels. This involved either using a classifier (CNN) or building on a pre-trained model like VGG-16/19, ResNet, Inception, and Inception-ResNet.

   Why we *didn't* opt for this: Initially we thought a decoder CNN following the object recognizer would colorize the image based on what the previous classifier recognizes the image as. However, we did not realize that the classifier would just give probabilities of different classes and not necessarily recognize local features within the image and so the CNN may not be able to colorize the image properly. On trying a basic model with this approach we also ran into other problems, as explained later.

5. Pre-trained model combined with CNNs in the encoder-decoder style: Another popular and effective approach is to pass the image into an encoder-type CNN that recognizes local features as well as passing the image parallely to a pre-trained model that classifies the image by recognizing global features and then combining the outputs of these two constructs. The combination then has information about local features and global features that gives a better idea about what colour a pixel should be. This is then passed through a decoder-type CNN that colorizes the pixels based on provided 'a' and 'b' labels.

   Why we finally opted for this: This seemed more within our scope and seemed more sophisticated than just an encoder-decoder style, and combining both local and global features seemed like a useful tool for determining colour. We have described our approach in greater detail as well as our trajectory and mistakes in a later section.

<u>Our Approach</u>:

As mentioned, we chose to use a pre-trained model for object recognition and global feature extraction and an encoder-type CNN for local features. This was then combined and passed through a decoder-type CNN to colorize the pixels based on provided 'a' and 'b' labels.

Preprocessing: This involved conversion of images from RGB to the Lab colour space and to resize images to pass them through the pretrained model. We take the 'L' channel as input in the independent Xtrain array and 'a' and 'b' values as labels in the dependent Ytrain array.

Pre-trained model: We chose the InceptionResnetV2 model as our pretrained models to extract global features for our images. It has been trained on the ImageNet dataset and gives probabilities for 1000 classes. There were several other options like VGG-16, VGG-19, ResNet-18 etc., but the reason for choosing this was that it offered lighter weights (thus faster to work with) as compared to the other models, and also had a better accuracy (it is the latest among many such models).

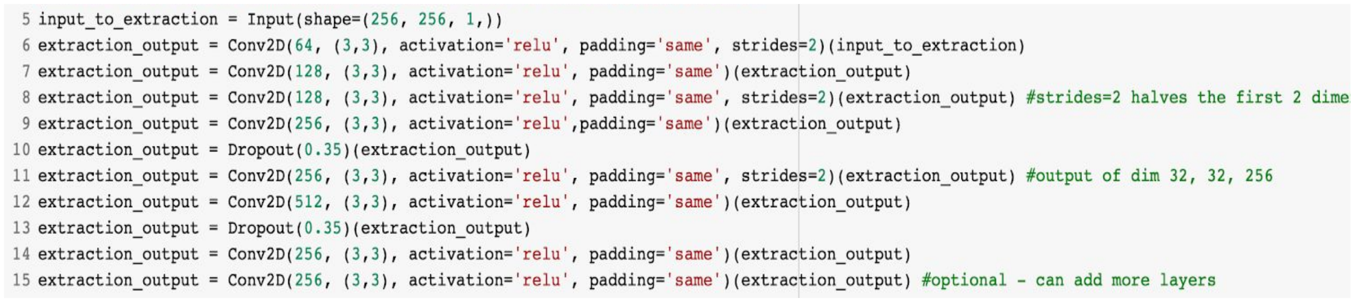
Architecture:

Input: Our images are of size 256 x 256. We extract only the 'L' channel out of the La*b* colour space to pass into the model, and the 'a' and 'b' values serve as labels. Thus, input is of dimensions (256, 256, 1).

Encoder: We have a CNN which helps us extract the local features from the 'L' channel. It also acts as an encoder as it compresses the image without losing much data and takes care of local correlation. We use the conventionally optimum width and height to bring the image down to, i.e., 256/8 = 32 for the bottleneck region. Hence, in this CNN we try to bring the image down to 32 x 32, and try to extract a power-of-2 number of features, in this case, 256.

To do this, we use multiple Conv2D layers where we are choosing the number of filters in an increasing order, to increase the third dimension to 256. To bring the image dimensions (256 x 256) down to (32 x 32), we use strides = 2 in 3 of the Conv2D layers. We also added a Dropout layer of strength 35% after the first 3-4 layers to prevent overfitting. By adding the dropout layers, we made sure that the network did not rely on any one feature as it could be randomly eliminated by the aforementioned layer. However, we placed them slightly later in the network to make sure that the network learned enough before eliminating random inputs. We had to keep experimenting with the number of convolution layers and the size of their filters. On one hand we wanted to the network to learn as much information as it could, while on the other hand we had to make sure that the network was learning only what was important without overfitting.

Our encoder CNN looked like a variation of this (we played around with number of layers):

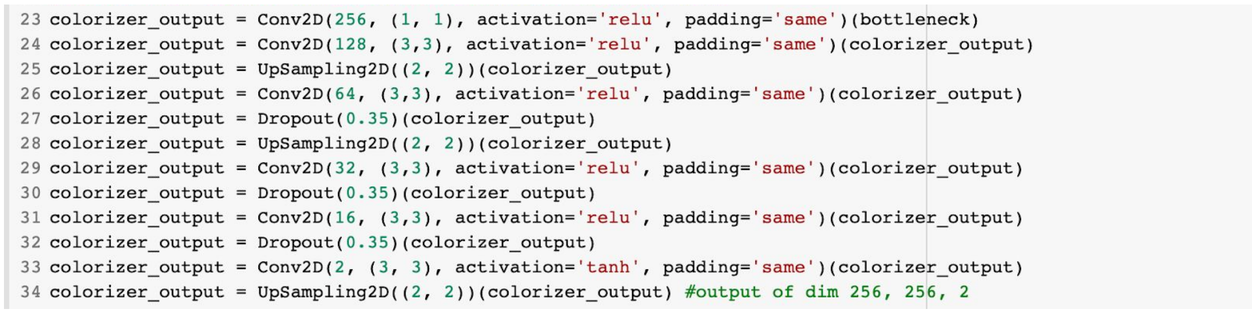


```
input_to_extraction = Input(shape=(256, 256, 1,))
extraction_output = Conv2D(64, (3,3), activation='relu', padding='same', strides=2)(input_to_extraction)
extraction_output = Conv2D(128, (3,3), activation='relu', padding='same')(extraction_output)
extraction_output = Conv2D(128, (3,3), activation='relu', padding='same', strides=2)(extraction_output) #strides=2 halves the first 2 dime
extraction_output = Conv2D(256, (3,3), activation='relu',padding='same')(extraction_output)
extraction_output = Dropout(0.35)(extraction_output)
extraction_output = Conv2D(256, (3,3), activation='relu', padding='same', strides=2)(extraction_output) #output of dim 32, 32, 256
extraction_output = Conv2D(512, (3,3), activation='relu', padding='same')(extraction_output)
extraction_output = Dropout(0.35)(extraction_output)
extraction_output = Conv2D(256, (3,3), activation='relu', padding='same')(extraction_output)
extraction_output = Conv2D(256, (3,3), activation='relu', padding='same')(extraction_output) #optional - can add more layers
```

Combination with pretrained model and decoder: We then combine this output with the features extracted from the pretrained model (which gives probabilities for 1000 classes). This combination is then passed through the decoder CNN (the colorizer), that gives colors to the pixels based on the 'a' and 'b' labels. The decoder is a CNN that has filters in a decreasing fashion to bring the last dimension back to 2 (for the 'a' and 'b' channels). We increase the image size back to the original by using Upsampling2D layers. The dimensions of the output are (256, 256, 2) (two outputs for the a and b channels each).

The Upsampling layers increased the scale of our pictures so that there was enough information for the image to be adequately coloured. However, we had to be careful as upsampling doesn't exactly enlarge the picture; it effectively guesses what each extra pixel could be, based on the pixels around it. Therefore, as much advantage as upsampling gives us, it conversely gives us data that isn't exactly "real". Therefore, we used the given type of layers sparingly, using them only when we needed to re-enlarge the picture from the bottleneck region.

Our decoder CNN looked like a variation of this (we played around with number of layers):

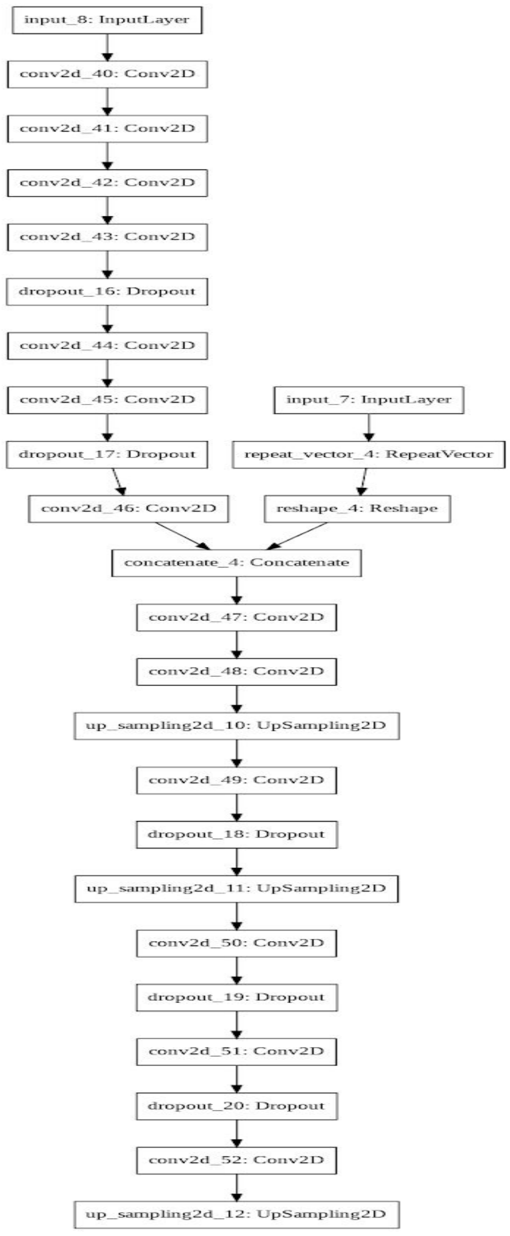


```
colorizer_output = Conv2D(256, (1, 1), activation='relu', padding='same')(bottleneck)
colorizer_output = Conv2D(128, (3,3), activation='relu', padding='same')(colorizer_output)
colorizer_output = UpSampling2D((2, 2))(colorizer_output)
colorizer_output = Conv2D(64, (3,3), activation='relu', padding='same')(colorizer_output)
colorizer_output = Dropout(0.35)(colorizer_output)
colorizer_output = UpSampling2D((2, 2))(colorizer_output)
colorizer_output = Conv2D(32, (3,3), activation='relu', padding='same')(colorizer_output)
colorizer_output = Dropout(0.35)(colorizer_output)
colorizer_output = Conv2D(16, (3,3), activation='relu', padding='same')(colorizer_output)
colorizer_output = Dropout(0.35)(colorizer_output)
colorizer_output = Conv2D(2, (3, 3), activation='tanh', padding='same')(colorizer_output)
colorizer_output = UpSampling2D((2, 2))(colorizer_output) #output of dim 256, 256, 2
```

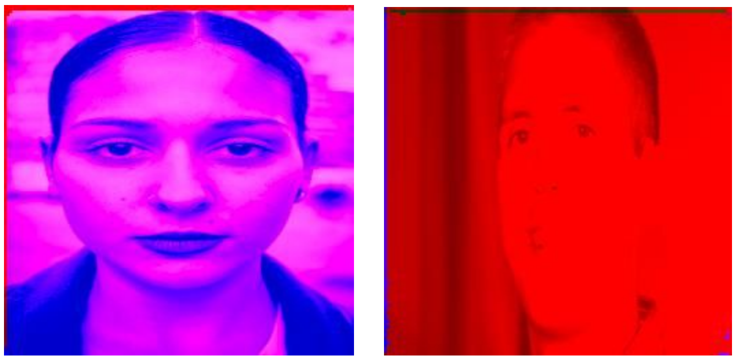
Activation Functions: Each layer uses a relu activation function and the last layer uses a tanh activation function so that we can calculate the loss between the desired a, b values (normalized between -1 and 1) and the actual a, b values our model produces. We use relu because it does not saturate with larger weights unlike sigmoid, and we need to take care of that because we are adding many layers in our model. Besides, relu is also a standard conventional choice only changed under specific circumstances.

Loss Function: The loss function we have used is the L2 loss function i.e. Mean Squared Loss. We chose this because we want to penalize wrong colors heavily, but not penalize those colors that are approximate to the desired color as much (since they are colors, similar colors have similar numerical values). The issue with this loss function is that its not robust to the multi-modal nature of the colorization problem and gives conservative, safe predictions (in this case, brown because it is the average of many colors), so there is almost always a slight brown tint to our results. For further improvement, we could consider color choice as a multinomial classification and consider bins for colors/ probabilities for each color (explained later). However, that is beyond the scope of this model.

Optimizer: We use adam that is somewhat of a combination of rmsprop and adagrad, and adjusts the learning rate efficiently and appropriate to the slope. It is also a standard convention to start with adam; on trying a few alternatives we did not see any improvements, so adam seemed like a good option.



<u>Difficulties faced</u>:
There were many trials where we got strange results and single-colored images like all red and all purple:



- When we tried to replace the encoder CNN by a pre-trained model in the encoder-decoder approach, which meant we were classifying the image and then passing the results through a CNN to colorize it. However, that approach clearly failed because we were not taking into account the local features and now that we look back, the approach does not seem effective. We also tried to feed in features from the pre-trained model into the CNN as opposed to the final classification. This involved excluding the last layer of the pre-trained model and feeding it into the decoder CNN. However, we took very long and finally failed to match dimensions so that the features output from the pre-trained model generated a 256 x 256 image through the decoder CNN.

- Another instance when we got single colored images or strange results is when we used the 'MSLE' loss function. We feel this is because MSLE will treat small differences between true and predicted values approximately the same as big differences between large true and predicted values.

<u>Limitations/ What we could have done better</u>:

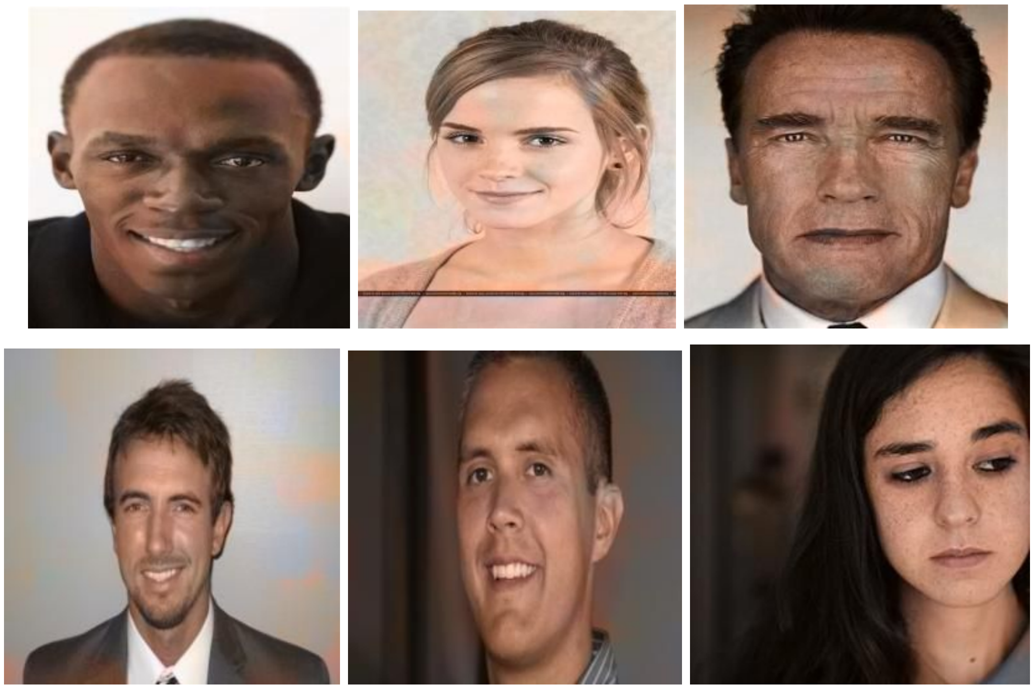
- Our model is trained only on images of faces, and only on 1500 images. Running it on more diverse images, for example a subset of ImageNet, it would produce much better results for other types of images.

- Our model uses MSE, which results in conservative predictions of the colour brown. Perhaps a better loss function would give better results, although trying the same on other loss functions like MAE and MSLE (that would penalize confident but wrong predictions *lesser* than MSE) did not give us as good results. Besides, this brown tint is acceptable on faces because of similar colours, but wouldn't look as good on other more colourful images.

- While finally training our model with a large dataset, we experienced something very unusual. Once we added more images i.e. moved from 1500 to 2000 images in our training set, our accuracy graph displayed a weird trend moving from 71 in one epoch to 29 in another epoch and so on. We tried shuffling data, but this didn't help us overcome this issue. Although we have ruled out any problems in the model because it runs on larger sets of other data. The problem could be within some of the images and the input Xtrain. Initially, we thought it was an issue with the images in the dataset but even after reshuffling the data, the issue continued to happen.We suspect, its some technical issue and not a conceptual issue, as our model works well with less number of images (upto ~1500-1600). The problem could also be with another aspect of one of the images that we cannot notice, or something to do with the array/numpy array conversion.

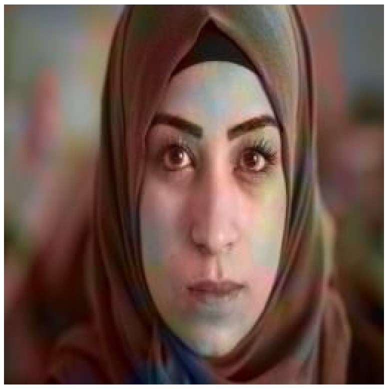
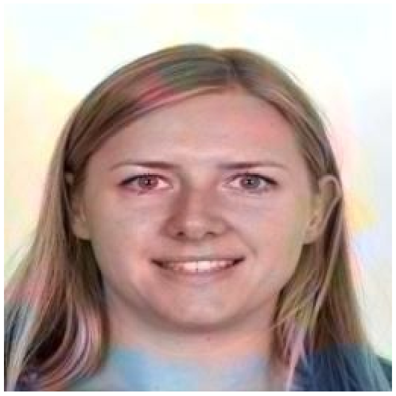
Results:

These were our final results with 1500 images of faces:



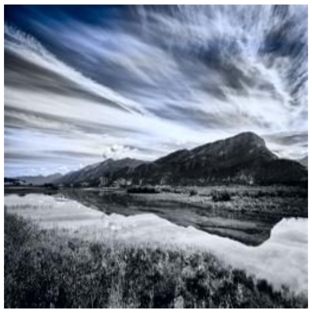
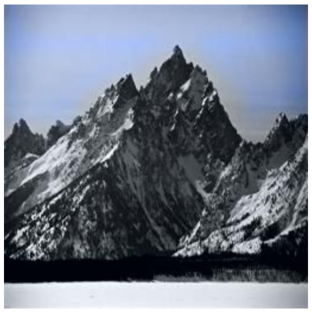
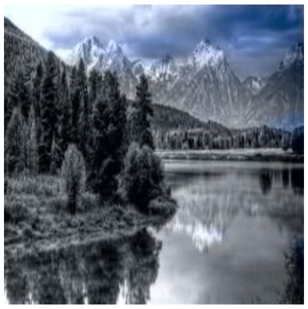
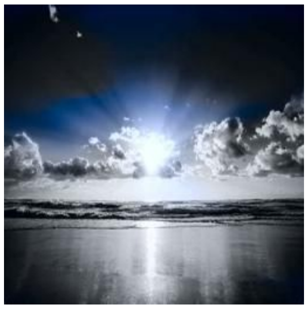
Here we can see that not only is the model recognizing smaller features e.g., difference in colour between chin and lips/ teeth, streaks in hair/ eyebrows, etc., but also larger features such as the collar, suit, outline of face and hair on background, etc. This could be due to the encoder CNN and pre-trained model focusing on different parts of images and the combination of this knowledge. We also see the brownish tint because of the conservative predictions. Perhaps this tint would be more of a problem on another dataset. Considering most faces have similar colors, perhaps a more general object dataset would give slightly worse results.

Our results, like the two images below, were not satisfactory when we used the Labelled Faces in the Wild dataset:

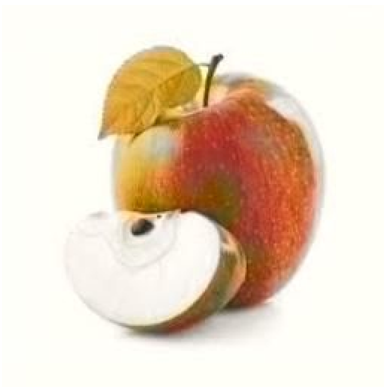
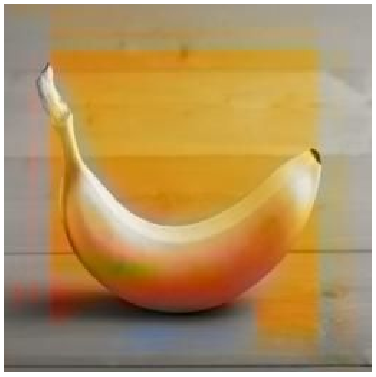
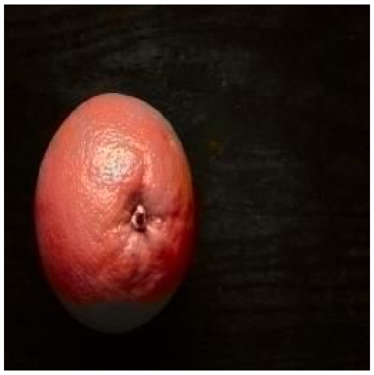
This could be because of the noise in the background of the LFW dataset. Besides, the images were too small and were resized to almost double their size.

These were few of our results using 370 images of mountains:

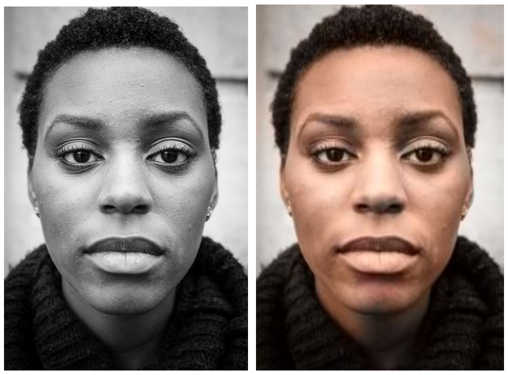
Here we see that though the model does not colorize the mountains, it does recognize edges well enough to color the sky, and also recognizes that it is the sky and colors it blue. Perhaps a better, larger dataset would give better results.

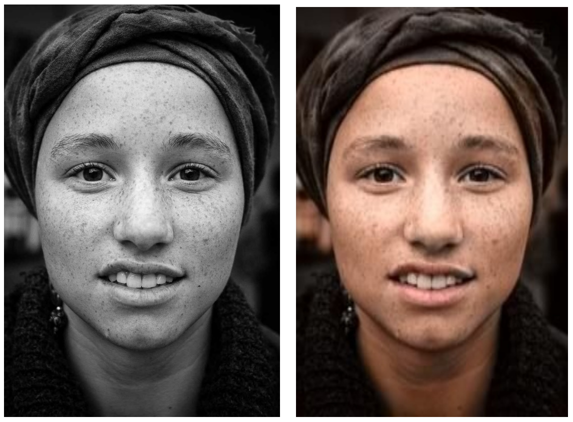
These were few of our results using 260 images of fruits (bananas, apples, and oranges - classes were disproportionate):

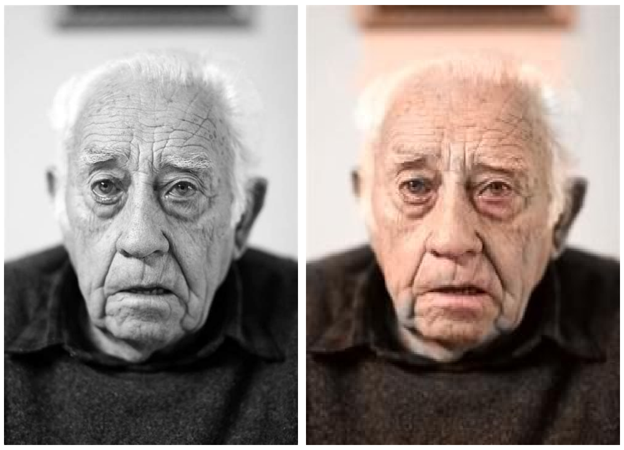




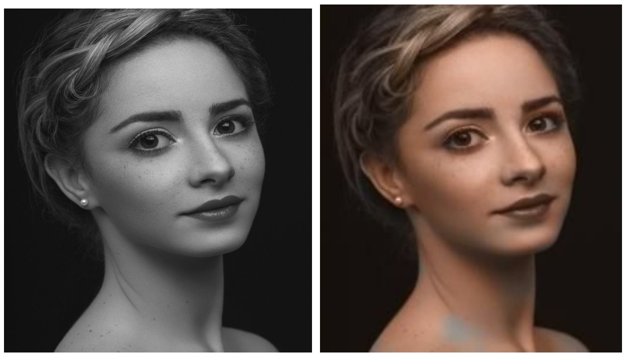
This did not give as good results but considering the tiny dataset, there was certainly a lot of scope for improvement.

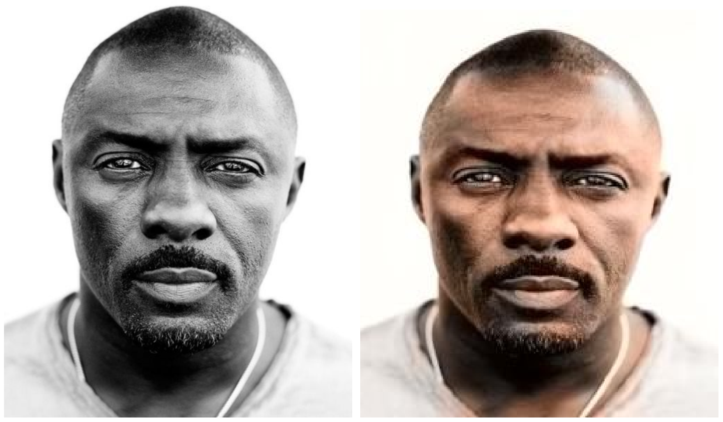
Best Results:











Note: We did not use the LFW dataset to train our model because:

1. It had a lot of background noise and our model wasn't adapting to it very well, thus giving a less accurate result on test cases.
2. The images in the LFW dataset were very small i.e. mostly 150 x 150. We need to rescale our images to 299 x 299 to send it to the pre-trained model and this upscaling of images was generating a lot of noise resulting in low accuracy.

Considering our dataset has a simple composition of a face with a solid coloured background, test cases where the background is less noisy gives better results as compared to the images with some kind of background noise.

Conclusion & Learning:

During the course of this project, we not only learnt how to write neural networks in code but also understood how to approach different types of problem statements. In the process of finding and building a model we understood how neural networks break down and work with images. This will help us break down our problem and approach it in a more structured manner in future projects. We also have a better idea of how to deal with bugs and mistakes by understanding where our model might make mistakes, whether the mistake is structural or due to overfitting, and by understanding the nature of the data better.

References:

During our understanding and implementation of the project we consulted a few papers on the topic. Some were research papers and some were college students' attempts at the same project.

- http://pages.cs.wisc.edu/~stunuguntla/MidTermReport.pdf
- https://arxiv.org/abs/1712.03400
- http://cs231n.stanford.edu/reports/2016/pdfs/219_Report.pdf