ST. CLAIR COLLEGE

Project Report
On

Coloring Black and White Images using Keras

Group number 21

Group Members:

Name	Student ID
Sachchid Wast	0753615
Megha Kanwar	0753633

Table of Content

Introduction

Historically, coloring black and white images required manual labour and was altogether very costly, time consuming and required artistic skills. Even today with the use of specialized softwares like Adobe Photoshop, not only do we require skills, but also must devote significant time to complete the coloring process. In this project we believe that training the neural networks rightly can prove to be super-efficient to do this task. Since this concept is relatively new in this area, different authors have come up with their models to produce the most efficient results possible.

For this project, we plan to colour greyscale images with the use of convolutional neural networks. The model will analyse a set of coloured images which are fed to it and after training, will be able to colour any unseen grayscale image appropriately. Upon experimenting we found out that U net like architecture for our model was the most appropriate to give decent results. We want the final product should come out to be as close as real looking as possible, that is, it shouldn't look like the picture was originally a grey scale. There can be a possibility that the real colours don’t match with the ones produced by the model but as far as the difference isn’t apparent, the model solves the purpose.

RGB (Red, Green, Blue) is the most popular color space, but in this project, we are going to use the LAB color space. LAB color space includes 3 channels viz. Luminance (having values between 0-100), A channel (values between -128 [bluish green] to 127[pinkish magenta]), B channel (having values between -128 [blue] to 127[yellow]). As our input is a black and white image (i.e. only contains information about luminosity) we will already have one channel with us, so only the other two channels will need a prediction. Whereas RGB requires prediction for 3 channels.

Related Work

Our project was inspired in part by Jeff Hwang's Image Colorization with deep Convolution neural networks [1]. His CNN based system colorized images using a classification-based model rather than a regression-based model. His model reads images with pixel dimension 224 x 224 and 3 channels with RGB color space which are then converted to CIELUV color space. U and V are his target channels. He uses a portion of VGG16 and used rectified linear unit as the activation function to accelerate training convergence, batch normalization layer prior to non-linearity layer to reduce training convergence time and improve accuracy. He used vanilla loss function and Huber penalty function. His final model looks as follows:

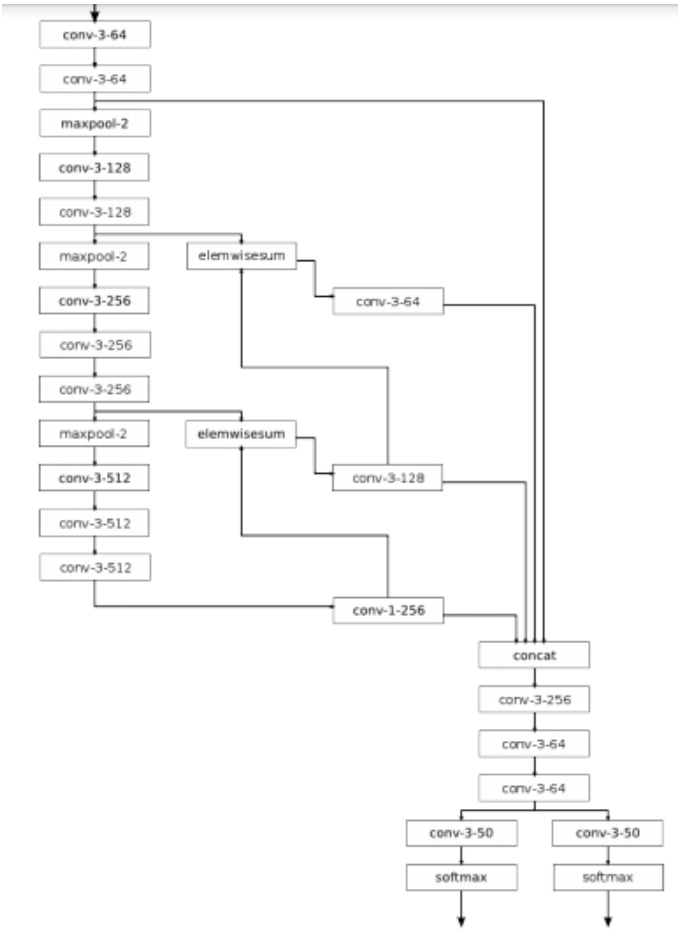


Fig1: Classification network schematic
Source: http://cs231n.stanford.edu/reports/2016/pdfs/219_Report.pdf

Fig 2: Output results for test images(left) as seen for regression model(middle) and classification model(right)

Source: http://cs231n.stanford.edu/reports/2016/pdfs/219_Report.pdf

Another amazing work was done by Zezhou Cheng [2] who aims to convert the greyscale images to fully colorized images with high quality. He proposed an adaptive image clustering to render training ambiguities. He used a large set of images from different categories and incorporated global image information. His major contributions include the effectiveness of model to color images from various scenes and carefully analyses low to high level of discriminative image feature descriptors like patch feature and DAISY feature. The limitations of his model are that it requires such a huge dataset that should contain images of all objects possible which is practically impossible.

Inspired by Zezhou Cheng's work [2], Sindhuja Kotala's paper [4] revolves around colorizing greyscale images and further using them to color a video. He makes use of VGG like architecture without any pooling layers along with multinomial loss function with color rebalancing. His model does very well specifically to images of cats, dogs and sunsets and colors video (sequence of images) in some cases.

Other related work includes that of Stephen Koo [3] who used deep convolutional generative adversarial networks (DCGANs) with a per-pixel Euclidean loss function on the chrominance values to colorize greyscale images. He used YUV color space (Y channel encodes luminance; U&V encode chrominance) instead as YUV minimizes per-pixel correlation between channels. He didn't make use of pooling or upscaling layers, so the output images have the same dimensions as the input images. His results showed problems with tuning the adversarial training in terms of how gradients were propagated from discriminator to the generator.

Methods

Data Pre-processing

First, we rescaled the size of the image using the ImageDataGenerator method of keras.processing.image class. The size of image after would 256*256 (height * width) pixels.

To train the network we are using colored images which are by default in RGB color space, but we are using LAB color space in our project. Before converting the color space, we need to normalize the RGB values i.e. convert the values which were originally between 0-255 to 0-1 by dividing them by 255, as it is preferred in neural networks to boost the speed.

So, to convert the images into LAB color space we are going to use the rgb2lab method from *skimage package's color class.* This method returns the image's size and the 3 channels for ex. Our data looks like this (256,256,3) after using the *shape method* where (256,256) are the width and the height of the photo and 3 is the number of channels in the photo. The zeroth channel contains the luminosity information (black and white image) and the other two channels have the color information.

So, we store the black and white image information i.e. the zeroth channel in X and the color information in Y. The following images are shown to further clear the concept.

Fig 3.1: Original color image

Source: http://shutha.org/node/851

Fig 3.2: Color image split into individual LAB channels
Source: http://shutha.org/node/851

Choosing a Dataset

We used many datasets in our project but finally chose to settle on dataset comprising of about 305 images of landscape images varying in image size. 305 of these are color images and 18 are grayscale images. Images used for training are stored in a folder called 'train' and those used for testing are in 'test' folder.

Colored images are used to train the model and the grayscale are used to test the model i.e. to color them. The output of the model i.e. the colored images are stored in a folder called 'results'.

Concept of using Autoencoder in our model.

Autoencoders are a specific type of feedforward neural networks where the input is the same as the output. [5] They reduce the size of the image (down sampling) and gets what the prominent features of the image are using Convolution layers, along with Max Pooling, which reduce the size of the image. This part is encoding is therefore encoding the data and therefore called encoder. since we do not need a compressed image but the image of the same size as the input, we need to increase the size of the image. This is done by using UpSampling2D layer which increases the size of the image, in our case it doubles the size of the image as compared to the previous layer. This part is decoding the image to its original size and hence called Decoder.

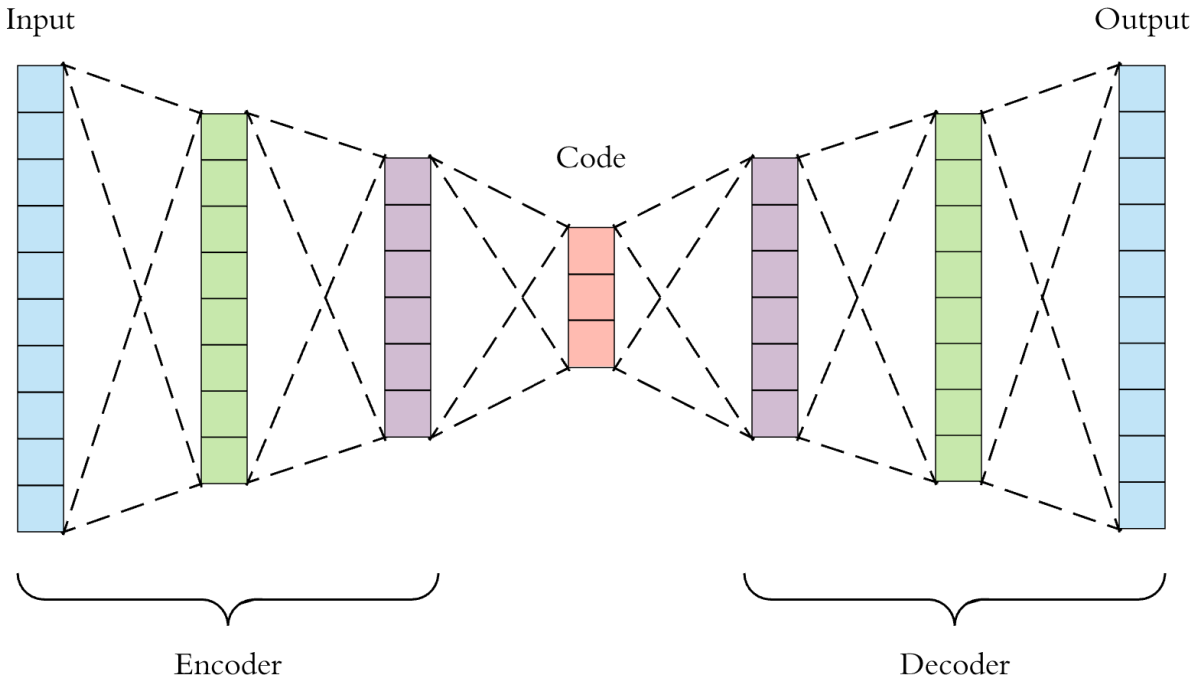


Fig 4: Architecture of Autoencoders

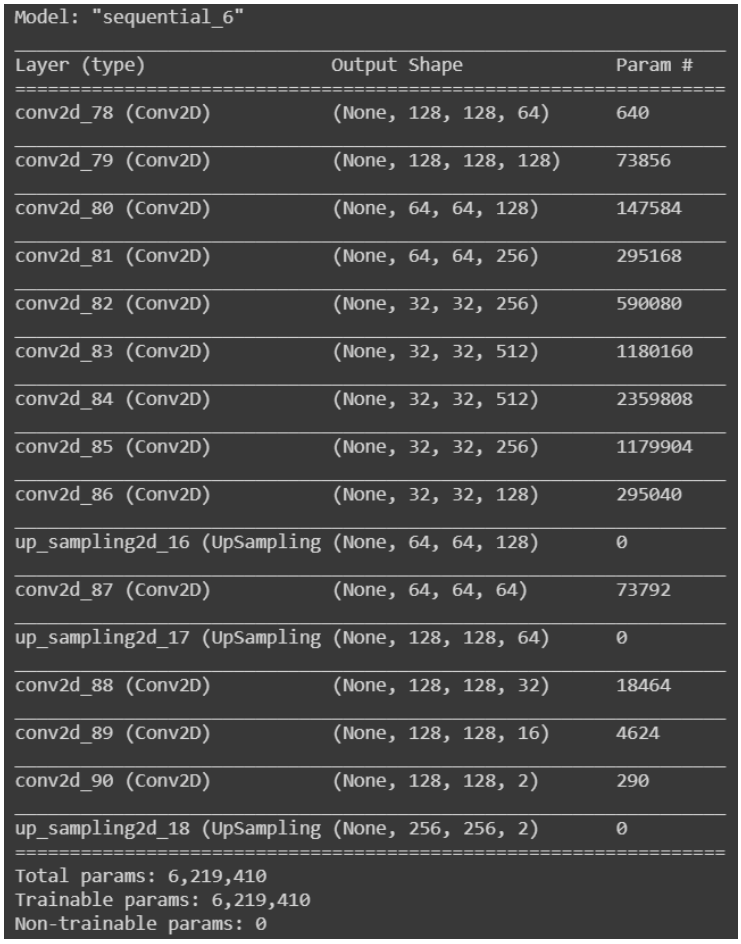
*Source: https://miro.medium.com/max/875/1*44eDEuZBEsmG_TCAKRI3Kw@2x.png*

By down sampling, the model better understands "WHAT" is present in the image, but it loses the information of "WHERE" it is present. [6]

In our project, the encoder gets the features of the image i.e. what colors are most prominent, which pixels should be applied which colors. The decoder tells where to put those colors i.e. the position of the pixels.

Training and building the model:

We have built our model by using the U-net model as reference as it contains encoders and decoders which we require in our case. The architecture of our model is given in the following figure:



```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_78 (Conv2D)           (None, 128, 128, 64)      640
_________________________________________________________________
conv2d_79 (Conv2D)           (None, 128, 128, 128)     73856
_________________________________________________________________
conv2d_80 (Conv2D)           (None, 64, 64, 128)       147584
_________________________________________________________________
conv2d_81 (Conv2D)           (None, 64, 64, 256)       295168
_________________________________________________________________
conv2d_82 (Conv2D)           (None, 32, 32, 256)       590080
_________________________________________________________________
conv2d_83 (Conv2D)           (None, 32, 32, 512)       1180160
_________________________________________________________________
conv2d_84 (Conv2D)           (None, 32, 32, 512)       2359808
_________________________________________________________________
conv2d_85 (Conv2D)           (None, 32, 32, 256)       1179904
_________________________________________________________________
conv2d_86 (Conv2D)           (None, 32, 32, 128)       295040
_________________________________________________________________
up_sampling2d_16 (UpSampling (None, 64, 64, 128)       0
_________________________________________________________________
conv2d_87 (Conv2D)           (None, 64, 64, 64)        73792
_________________________________________________________________
up_sampling2d_17 (UpSampling (None, 128, 128, 64)      0
_________________________________________________________________
conv2d_88 (Conv2D)           (None, 128, 128, 32)      18464
_________________________________________________________________
conv2d_89 (Conv2D)           (None, 128, 128, 16)      4624
_________________________________________________________________
conv2d_90 (Conv2D)           (None, 128, 128, 2)       290
_________________________________________________________________
up_sampling2d_18 (UpSampling (None, 256, 256, 2)       0
=================================================================
Total params: 6,219,410
Trainable params: 6,219,410
Non-trainable params: 0
```

Fig 5: Architecture of our model.

We changed the following parameters of the model:

- Kernel_size: 3,5,7
- Padding: ‘same’, ‘valid’
- Strides: 1,2
- Optimizer: ‘adamax’,’adam’,’rmse’
- Epochs: 100,500,600,1000

We found that changing optimizer and epochs had a huge change as compared to other parameters. At epochs = 100, the model did poorly i.e. it was underfitted did not color the model as intended. So, we increased the epochs to 500 and saw better results and the validation loss was lesser compared to previous. So, we again tried with 600 and 1000 epochs whether the model would perform any better but saw that the model started under performing i.e. the validation loss increased suddenly and did not decrease. So, we recommend keeping the number of epochs less than 600.

We changed the optimizer's and found that **Adamax** has the highest validation accuracy and **RMSE** had the lowest validation accuracy. The difference between both was nearly 5%.

Coloring the images

We created an empty NumPy array and structured it so that it is ready to find the predicted values. We created an empty python list *'colorizer'* and import our test images by using load_img and converting that array by using img_to_array method and then resize it to our output size of 256*256. We do this for all our images using a for loop and append the resulting *img* to the *colorizer* list. We then convert that list to a NumPy array so that it can further used to find the predicted values. It is important to remember here that the colorizer array contains black and white images.

We use the model's predict method and pass the *colorizer* array as our argument and stored in an array called *output*. We multiply that array with 128 to bring our data into original scale as LAB color space has values between -128 to 127. The output array contains the color information i.e. A and B channels. Now, we need to combine them. We did this by creating an array called result which had the shape (15,256,256,3). 15 shows that we have that many images in our test set. 256*256 is the size of the image and 3 is the number of channels. To combine the results of *colorizer and output* we just need to work with the channel part. For the luminosity channel we give the ***colorizer*** as the input and for A and B channel we give ***output*** as the input to ***result.*** Below is the code snipped for better understanding:

```
result = np.zeros((256, 256, 3))
result[:,:,0] = colorizer[i][:,:,0]
result[:,:,1:] = output[i]
```

We then convert the *result* array to RGB format by using lab2rgb method and then saving that image by using imsave() method. We do this for every image by using a for loop.

Results

After training our model with 500 Epochs we got a validation accuracy of 76.94% and train accuracy of 90.95%. The model colored the grey-scale images successfully though not perfectly. Out of the 18 images provided in test images, one image was colored perfectly to the point where upon asking individuals about it, they couldn't say confidently that it could be a machine colored image.

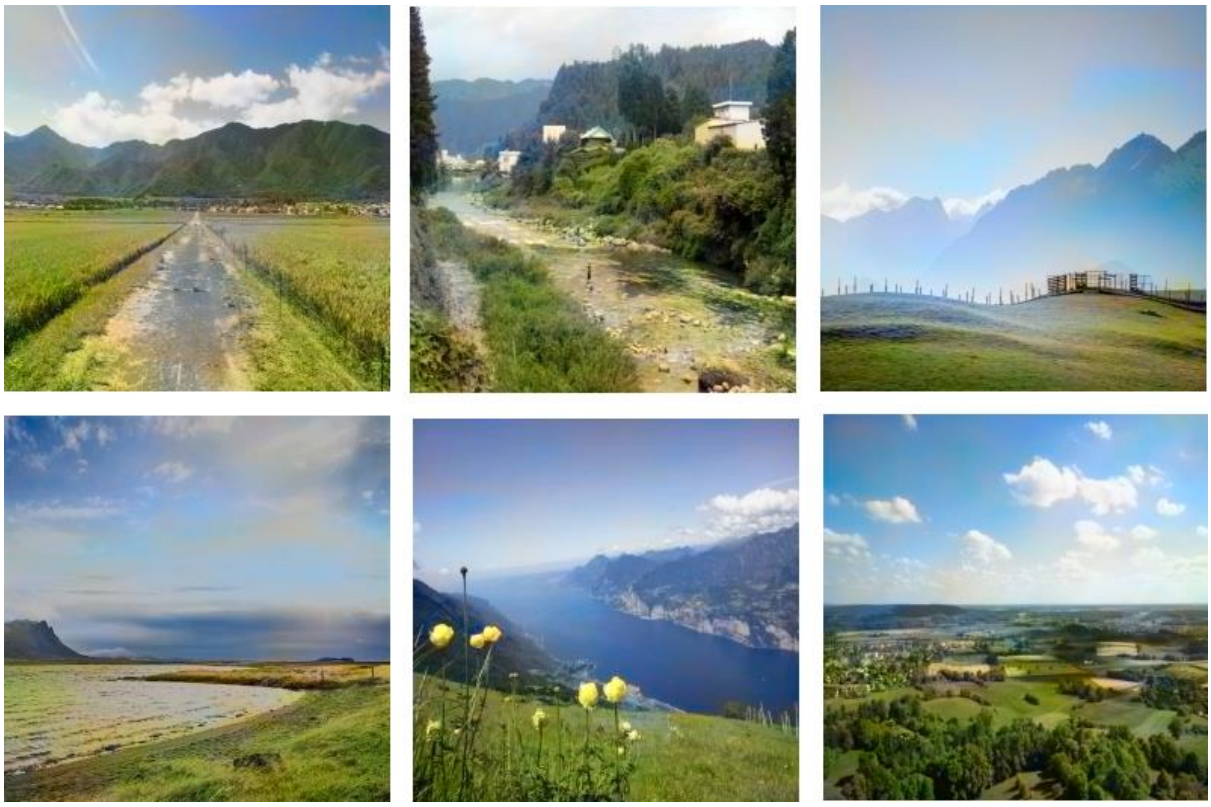


Fig 6: Test images colored by our model

Discussion

Why we chose this dataset?

At first, we were going to use the ImageNet dataset, but we didn't get the approval till now to use the dataset.

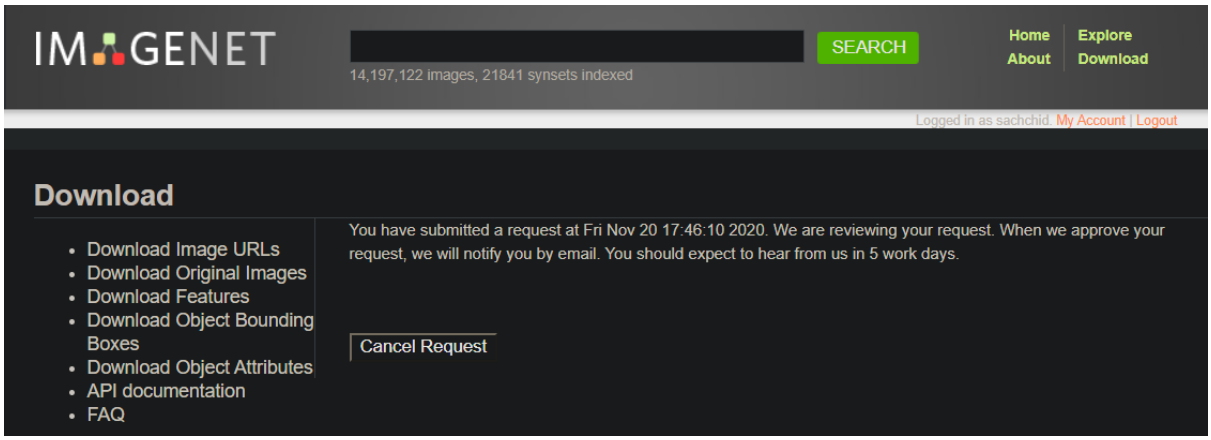


Fig 6: Approval status of ImageNet

Therefore, we used dataset from FloydHub which contained nearly 10,000 images. As it turned out, we couldn't use the whole dataset because Google Colab would crash because it had insufficient RAM (12 GB) to handle all of that data. We kept on reducing the size of the dataset till the point where Colab won't crash. We hit this point at about 4000 images. So, we trained our model on this dataset and we got poor results as the data was too diverse and our model couldn't color the images as expected. We didn't want to increase the size of our model as it was taking around 4 hrs. to completely fit the model.

So, we decided to take few hundred images of a specific category to test whether our model is able to color the images or not. That's why we took around 300 images of landscapes to train on and 18 landscape images in test set. The results we got were somewhat satisfactory. We asked our friends and course mates whether they can distinguish between the original image and artificially colored image, they could differentiate nearly all of the images except one image.

Batch size and flow_from_directory vs img_to_array

While importing our dataset, first we used the img_to_array method, but we found out that it is too slow to import a huge amount of dataset. That's the reason we switched over to flow_from_directory. We found that this method is very fast at importing huge datasets by using the batch size parameter. This method is useful when we are just pre-processing the data using its own methods for eg: rescaling. We were unable to use flow from directory's functionality for iterating images while fitting the model since we are processing them. But we want to process the whole dataset and not just the batch of the dataset. So, to work around this

problem we put the whole dataset in the batch_size parameter so that the model is trained over the entire dataset. But this way, the functionality of the batch size is lost.

Conclusion

Even though many authors have worked on a similar problem statement, the results have never been perfectly accurate for each and every image given for testing. We can conclude that our model did perform to a certain extent given the limitation of computational resources how difficult it is to get the most accurate results.

Reference

1. http://cs231n.stanford.edu/reports/2016/pdfs/219_Report.pdf
2. https://arxiv.org/pdf/1605.00075.pdf
3. http://cs231n.stanford.edu/reports/2016/pdfs/224_Report.pdf
4. http://ijcsn.org/IJCSN-2019/8-2/Automatic-Colorization-of-Black-and-White-Images-using-Deep-Learning.pdf
5. https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798
6. https://towardsdatascience.com/understanding-semantic-segmentation-with-unet-6be4f42d4b47
7. https://forums.fast.ai/t/images-normalization/4058/6
8. https://medium.com/@emilwallner/colorize-b-w-photos-with-a-100-line-neural-network-53d9b4449f8d