# Coloring Black and White Images

Kunjan Khatri Kathan Patel

CSCI S-89, Introduction to Deep Learning Harvard Extension School

We were recently watching a classic and historic Bollywood Movie which was originally in black and white, now colorized. This made us eager to try something like this using deep learning. So, we decided to start with colouring black and white images. Today, colorization is done by hand in Photoshop. A picture can take up to one month to colorize. It requires extensive research. A face alone needs up to 20 layers of pink, green and blue shades to get it just right. So, we thought of trying this using neural networks. First attempt was to train a model on input image and get results based on that. This gave us very great result as the model was trained on one image only. Second, we used the inception resnet v2 — a network trained on 1.2M images. This way model will try to generalize color schema on unknown images. Starting with few images and a smaller number of epochs didn't yield any good results. We gradually increased training data and epochs and ended up with images having more of a red shade. Major issue was hardware i.e. if we had more GPU power, we can train around 10k images 10K images with 21 epochs will take about 11 hours on a Tesla K80 GPU.

## Requirements/Configurations:

- Python 3
- TensorFlow 1.15
- Keras 2.2.4

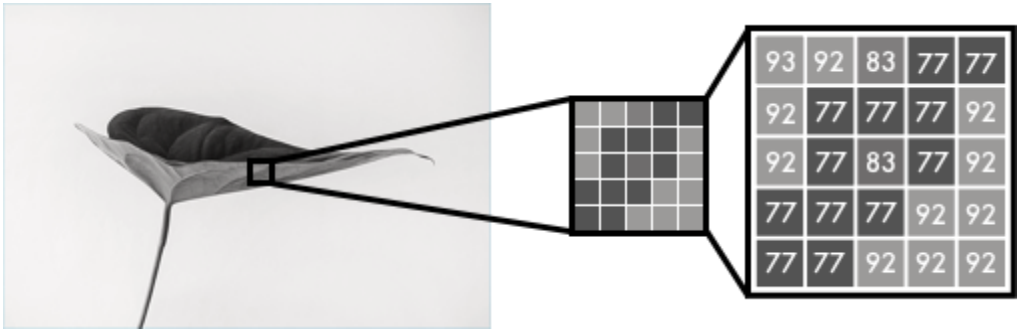
## Data:

- https://www.floydhub.com/emilwallner/datasets/colornet

## Core Logic:

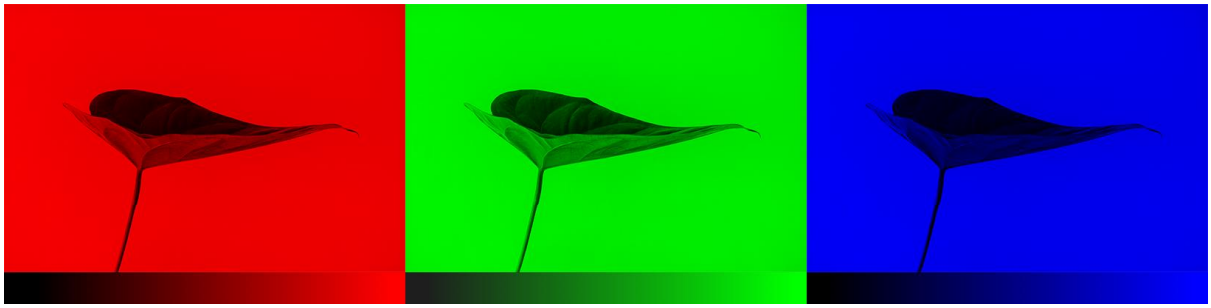
In this section, we will outline how to render an image, the basics of digital colors, and the main logic for our neural network.

Black and white images can be represented in grids of pixels. Each pixel has a value that corresponds to its brightness. The values span from 0 - 255, from black to white.




Color images consist of three layers: a red layer, a green layer, and a blue layer. This might be counter-intuitive to you. Imagine splitting a green leaf on a white background into the three channels. Intuitively, you might think that the plant is only present in the green layer.

But, as you see below, the leaf is present in all three channels. The layers not only determine color, but also brightness.



To achieve the color white, for example, you need an equal distribution of all colors. By adding an equal amount of red and blue, it makes the green brighter. Thus, a color image encodes the color and the contrast using three layers:

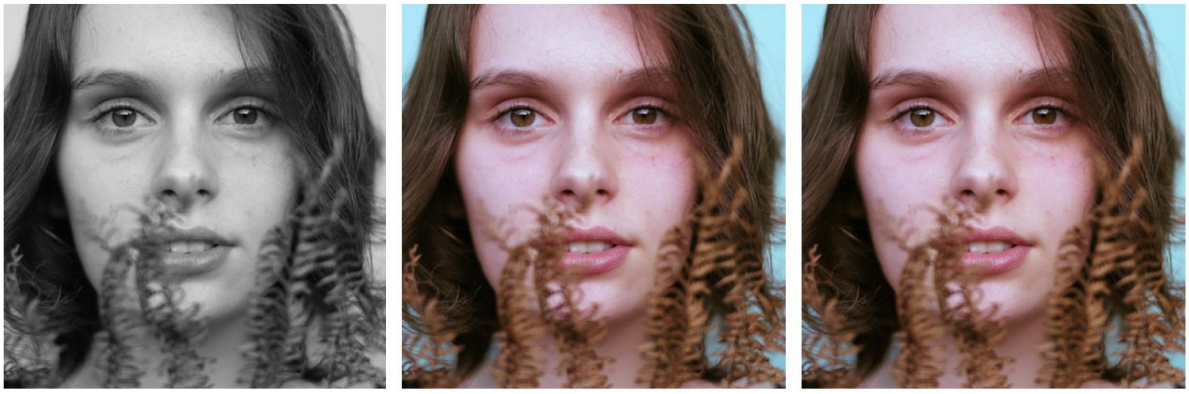

Just like black and white images, each layer in a color image has a value from 0 - 255. The value 0 means that it has no color in this layer. If the value is 0 for all color channels, then the image pixel is black.

As you may know, a neural network creates a relationship between an input value and output value. To be more precise with our colorization task, the network needs to find the traits that link grayscale images with colored ones. In sum, we are searching for the features that link a grid of grayscale values to the three-color grids.

## Version 1:

We'll start by making a simple version of our neural network to color an image of a woman's face.

The middle picture is done with our neural network and the picture to the right is the original color photo. The network is trained and tested on the same image.



## Color space:

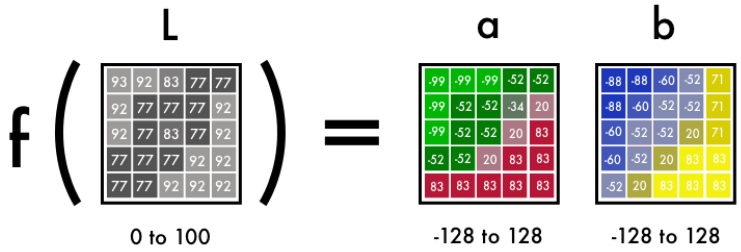
First, we'll use an algorithm to change the color channels, from RGB to Lab. L stands for lightness, and a and b for the color spectrums green–red and blue yellow.

As you can see below, a Lab encoded image has one layer for grayscale and have packed three color layers into two. This means that we can use the original grayscale image in our final prediction. Also, we only have two channels to predict.

As you can see in the above image, the grayscale image is a lot sharper than the color layers. This is another reason to keep the grayscale image in our final prediction.

## From B&W to color:

Our final prediction looks like this. We have a grayscale layer for input, and we want to predict two color layers, the **ab** in **Lab**. To create the final color image, we'll include the **L**/grayscale image we used for the input, thus, creating a **Lab** image.




To turn one layer into two layers, we use convolutional filters. Think of them as the blue/red filters in 3D glasses. Each filter determines what we see in a picture. They can highlight or remove something to extract information out of the picture. The network can either create a new image from a filter or combine several filters into one image.

For a convolutional neural network, each filter is automatically adjusted to help with the intended outcome. We'll start by stacking hundreds of filters and narrow them down into two layers, the **a** and **b** layers.

## Python Notebook:

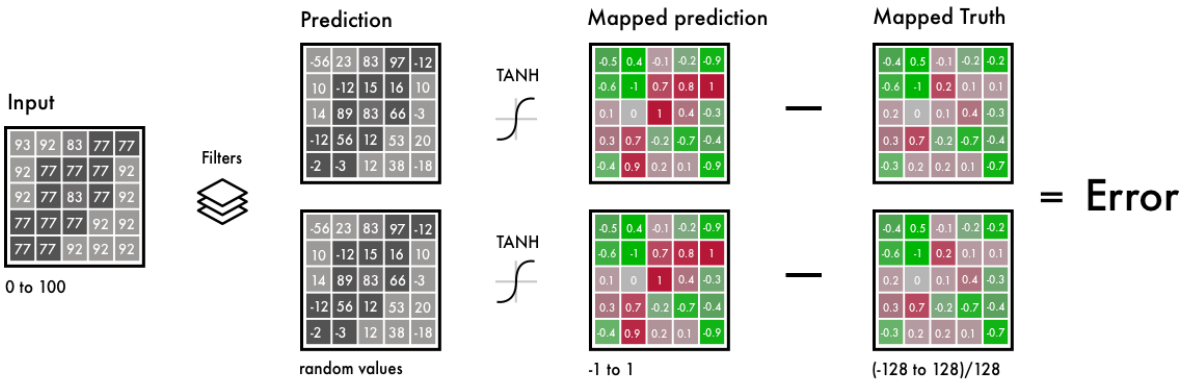
Locate Version_1/version_1.ipynb Open it and run all the cells.

## Implementation:

To recap, the input is a grid representing a black and white image. It outputs two grids with color values. Between the input and output

values, we create filters to link them together, a convolutional neural network.

When we train the network, we use colored images. We convert RGB colors to the Lab color space. The black and white layer is our input and the two-colored layers are the output.




To the left side, we have the B&W input, or filters and the prediction from our neural network.

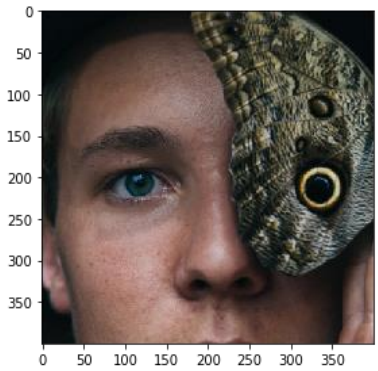
We map the predicted values and the real values within the same interval. This way, we can compare the values. The interval goes from -1 to 1. To map the predicted values, we use a Tanh activation function. For any value you give the Tanh function, it will return -1 to 1.

The true color values go from -128 to 128, this is the default interval in the Lab color space. By dividing them with 128, they too fall within the -1 to 1 interval. This enables us to compare the error from our prediction.

After calculating the final error, the network updates the filters to reduce the total error. The network stays in this loop until the error is as low as possible.

## Input Image:

```
# Original input image
imshow('man.jpg')
```




## Load Image:

```
# Get images
image = img_to_array(load_img('man.jpg'))
image = np.array(image, dtype=float)
```

## Scaling:

```
X = rgb2lab(1.0/255*image)[:,:,0]
Y = rgb2lab(1.0/255*image)[:,:,1:]
```

**1.0/255**, indicates that we are using a 24-bit RGB color space. It means that we are using 0-255 numbers for each color channel. This is the standard size of colors and results in 16.7 million color combinations. Since humans can only perceive 2-10 million colors, it does not make much sense to use a larger color space.

```
Y /= 128
X = X.reshape(1, 400, 400, 1)
Y = Y.reshape(1, 400, 400, 2)
```

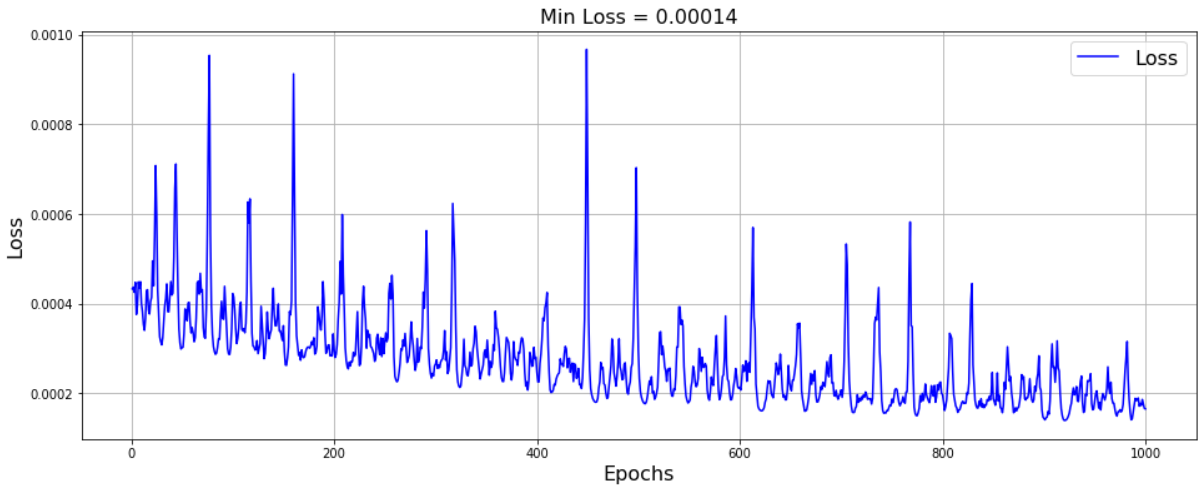
The Lab color space has a different range compared to RGB. The color spectrum ab in Lab goes from -128 to 128. By dividing all values in the output layer with 128, we force the range between -1 and 1. We match it with our neural network, which also returns values between -1 and 1.

After converting the color space from **rgb2lab()**, we select the grayscale layer with: **[:, :, 0]**. This is our input for the neural network. **[:, :, 1:]** selects the two color layers green–red and blue–yellow.

# Model Summary:

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, None, None, 8)     80
_________________________________________________________________
conv2d_2 (Conv2D)            (None, None, None, 8)     584
_________________________________________________________________
conv2d_3 (Conv2D)            (None, None, None, 16)    1168
_________________________________________________________________
conv2d_4 (Conv2D)            (None, None, None, 16)    2320
_________________________________________________________________
conv2d_5 (Conv2D)            (None, None, None, 32)    4640
_________________________________________________________________
conv2d_6 (Conv2D)            (None, None, None, 32)    9248
_________________________________________________________________
up_sampling2d_1 (UpSampling2 (None, None, None, 32)    0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, None, None, 32)    9248
_________________________________________________________________
up_sampling2d_2 (UpSampling2 (None, None, None, 32)    0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, None, None, 16)    4624
_________________________________________________________________
up_sampling2d_3 (UpSampling2 (None, None, None, 16)    0
_________________________________________________________________
conv2d_9 (Conv2D)            (None, None, None, 2)     290
=================================================================
Total params: 32,202
Trainable params: 32,202
Non-trainable params: 0
```

# Model Loss:



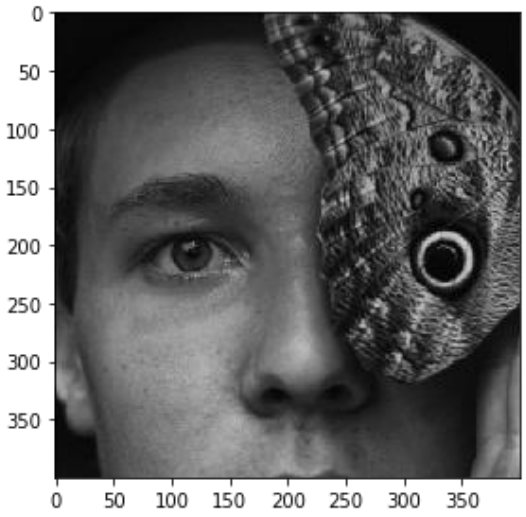
After training the neural network, we make a final prediction which we convert into a picture.

```
output = model.predict(X)
output *= 128
```

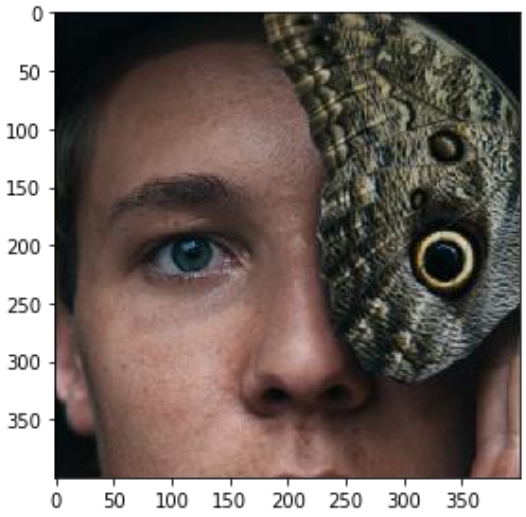
Here we use a grayscale image as input and run it through our trained neural network. We take all the output values between -1 and 1 and multiply it with 128. This gives us the correct color in the Lab color spectrum.

```
# Output colorizations
cur = np.zeros((400, 400, 3))
cur[:,:,0] = X[0][:,:,0]
cur[:,:,1:] = output[0]
```

Lastly, we create a black RGB canvas by filling it with three layers of 0s. Then we copy the grayscale layer from our test image. Then we add our two-color layers to the RGB canvas. This array of pixel values is then converted into a picture.




Gray scale image

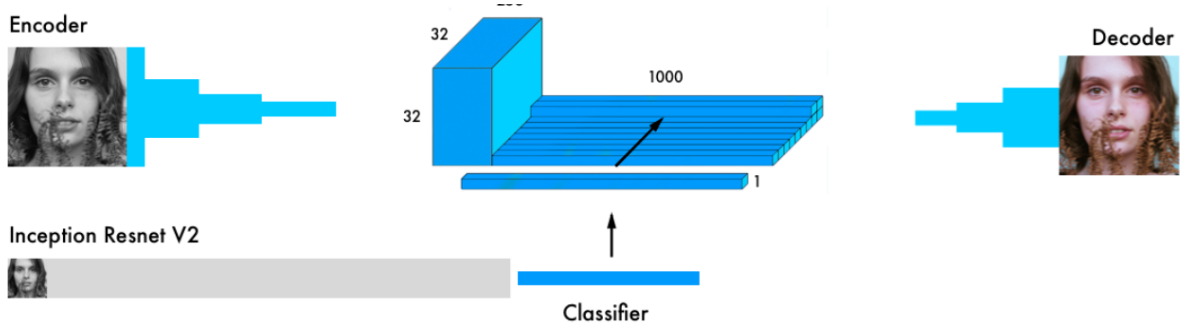



Output image

## Final version:

Our final version of the colorization neural network has four components. We split the network we had before into an encoder and a decoder. Between them, we'll use a fusion layer.

In parallel to the encoder, the input images also run through one of the today's most powerful classifiers — the inception resnet v2 — a network trained on 1.2M images. We extract the classification layer and merge it with the output from the encoder.




By transferring the learning from the classifier to the coloring network, the network can get a sense of what is in the picture. Thus, enabling the network to match an object representation with a coloring scheme.

## Implementation:

## Python Notebook:

Locate Final_Version/final_version.ipynb Open it and click shift+enter on all the cells.

### Input and Scale Images:

```
# Get images
X = []
for filename in os.listdir('Train/'):
    X.append(img_to_array(load_img('Train/'+filename)))
X = np.array(X, dtype=float)
Xtrain = 1.0/255*X
```

Keras's functional API is ideal when we are concatenating or merging several models.

First, we download the inception resnet v2 neural network and load the weights. Since we will be using two models in parallel, we need to specify which model we are using. This is done in TensorFlow, the backend for Keras.

```
#Load weights
inception = InceptionResNetV2(weights='imagenet', include_top=True)
inception.graph = tf.get_default_graph()
```

To create our batch, we use the tweaked images. We turn them black and white and run in through the inception resnet model.

```
grayscaled_rgb = gray2rgb(rgb2gray(batch))
embed = create_inception_embedding(grayscaled_rgb)
```

First, we have to resize the image to fit into the inception model. Then we use the preprocessor to format the pixel and color values according to the model. In the final step, we run it through the inception network and extract the final layer of the model.

```
def create_inception_embedding(grayscaled_rgb):
    grayscaled_rgb_resized = []
    for i in grayscaled_rgb:
        i = resize(i, (299, 299, 3), mode='constant')
        grayscaled_rgb_resized.append(i)
    grayscaled_rgb_resized = np.array(grayscaled_rgb_resized)
    grayscaled_rgb_resized = preprocess_input(grayscaled_rgb_resized)
    with inception.graph.as_default():
        embed = inception.predict(grayscaled_rgb_resized)
    return embed
```

Here we apply some transformations to images so that our model can learn better.

```
# Image transformer
datagen = ImageDataGenerator(
        shear_range=0.2,
        zoom_range=0.2,
        rotation_range=20,
        horizontal_flip=True)

#Generate training data
batch_size = 10
```

Going back to the generator. For each batch, we generate 20 images in the below format. It takes about an hour on a Tesla K80 GPU. It can do up to 50 images at a time with this model without having memory problems.

```
yield ([X_batch, create_inception_embedding(grayscaled_rgb)], Y_batch)
```

This matches with our colornet model format.

```
model = Model(inputs=[encoder_input, embed_input], outputs=decoder_output)
```

Encoder input is fed into our Encoder model, the output of the Encoder model is then fused with the embed input in the fusion layer; the output of the fusion is then used as input in our Decoder model, which then returns the final output, decoder output.

```
#Fusion
fusion_output = RepeatVector(32 * 32)(embed_input)
fusion_output = Reshape(([32, 32, 1000]))(fusion_output)
fusion_output = concatenate([encoder_output, fusion_output], axis=3)
fusion_output = Conv2D(256, (1, 1), activation='relu', padding='same')(fusion_output)
```

In the fusion layer, we first multiply the 1000 category layer by 1024 (32 * 32). This way, we get 1024 rows with the final layer from the inception model. It's then reshaped from 2D to 3D, a 32 x 32 grid with the 1000 category pillars. These are then linked together with the output from the encoder model. We apply a 254 filtered convolutional network with a 1X1 kernel, the final output of the fusion layer.

Now we scale and embed our test images using the below function.

```
color_me = []
for filename in os.listdir('Test/'):
    color_me.append(img_to_array(load_img('Test/'+filename)))
color_me = np.array(color_me, dtype=float)
color_me = 1.0/255*color_me
color_me = gray2rgb(rgb2gray(color_me))
color_me_embed = create_inception_embedding(color_me)
color_me = rgb2lab(color_me)[:,:,:,0]
color_me = color_me.reshape(color_me.shape+(1,))
```

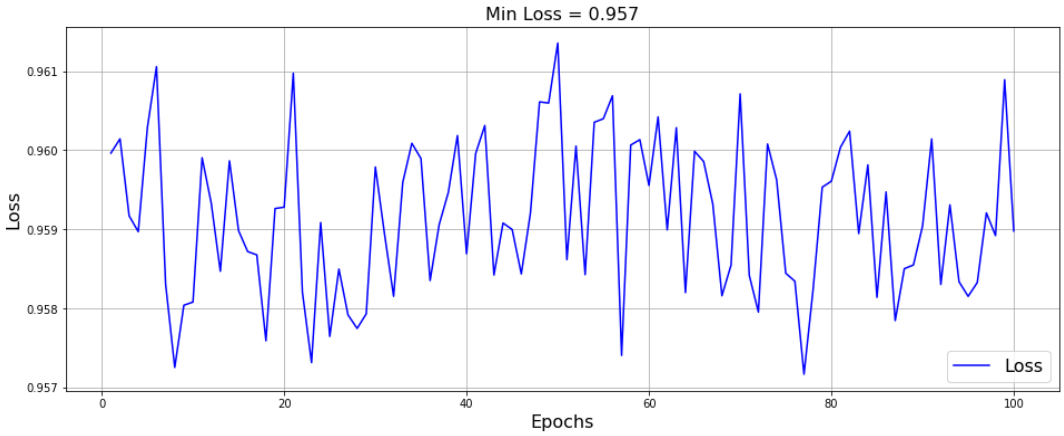
Next we run our model.

# Model Summary:

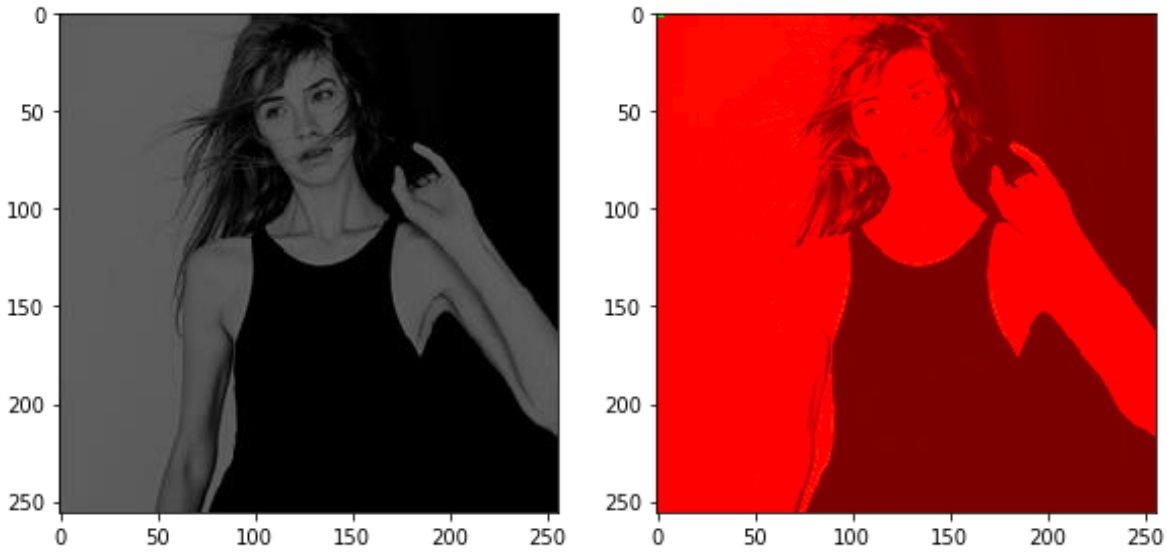
| Layer (type) | Output Shape | Param # | Connected to |
|---|---|---|---|
| input_3 (InputLayer) | (None, 256, 256, 1) | 0 | |
| conv2d_204 (Conv2D) | (None, 128, 128, 64) | 640 | input_3[0][0] |
| conv2d_205 (Conv2D) | (None, 128, 128, 128 | 73856 | conv2d_204[0][0] |
| conv2d_206 (Conv2D) | (None, 64, 64, 128) | 147584 | conv2d_205[0][0] |
| conv2d_207 (Conv2D) | (None, 64, 64, 256) | 295168 | conv2d_206[0][0] |
| conv2d_208 (Conv2D) | (None, 32, 32, 256) | 590080 | conv2d_207[0][0] |
| conv2d_209 (Conv2D) | (None, 32, 32, 512) | 1180160 | conv2d_208[0][0] |
| input_2 (InputLayer) | (None, 1000) | 0 | |
| conv2d_210 (Conv2D) | (None, 32, 32, 512) | 2359808 | conv2d_209[0][0] |
| repeat_vector_1 (RepeatVector) | (None, 1024, 1000) | 0 | input_2[0][0] |
| conv2d_211 (Conv2D) | (None, 32, 32, 256) | 1179904 | conv2d_210[0][0] |
| reshape_1 (Reshape) | (None, 32, 32, 1000) | 0 | repeat_vector_1[0][0] |
| concatenate_1 (Concatenate) | (None, 32, 32, 1256) | 0 | conv2d_211[0][0]<br>reshape_1[0][0] |
| conv2d_212 (Conv2D) | (None, 32, 32, 256) | 321792 | concatenate_1[0][0] |
| conv2d_213 (Conv2D) | (None, 32, 32, 128) | 295040 | conv2d_212[0][0] |
| up_sampling2d_1 (UpSampling2D) | (None, 64, 64, 128) | 0 | conv2d_213[0][0] |
| conv2d_214 (Conv2D) | (None, 64, 64, 64) | 73792 | up_sampling2d_1[0][0] |
| up_sampling2d_2 (UpSampling2D) | (None, 128, 128, 64) | 0 | conv2d_214[0][0] |
| conv2d_215 (Conv2D) | (None, 128, 128, 32) | 18464 | up_sampling2d_2[0][0] |
| conv2d_216 (Conv2D) | (None, 128, 128, 16) | 4624 | conv2d_215[0][0] |
| conv2d_217 (Conv2D) | (None, 128, 128, 2) | 290 | conv2d_216[0][0] |
| up_sampling2d_3 (UpSampling2D) | (None, 256, 256, 2) | 0 | conv2d_217[0][0] |

Total params: 6,541,202
Trainable params: 6,541,202
Non-trainable params: 0

# Model Loss:



**Output:**




Gray scale image Output image

Seems the model underfits and captures on color only. This happens because training was done on very a smaller number of images. Given more training data and GPU power there is huge scope for improvement.

## Future Work:

- Implement it with another pre-trained model
- Use more GPU power
- A different dataset
- Enable the network to grow in accuracy with more pictures
- Apply it to video
- Try on larger images, by tiling smaller ones

## YouTube Video Link:

https://youtu.be/9acbLWBy3oA

## Reference:


- https://arxiv.org/abs/1712.03400
- https://arxiv.org/abs/1611.07004
- https://github.com/NVIDIA/pix2pixHD
- https://blog.floydhub.com/colorizing-b-w-photos-with-neural-networks/
- https://github.com/emilwallner/Coloring-greyscale-images