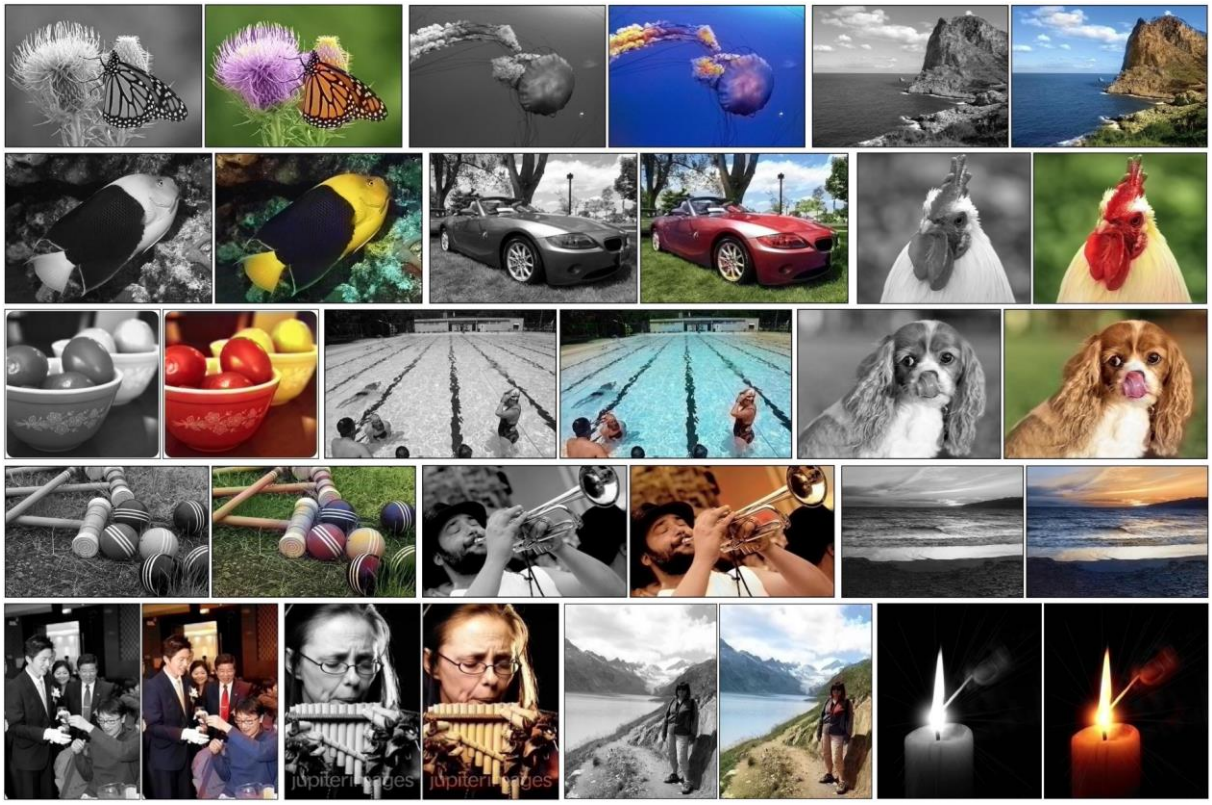
COLORIZATION OF GRAYSCALE IMAGES USING NEURAL NETWORKS



Team Members:

Sharath Venkatesh, CWID:893324566.

INDEX

INTRODUCTION

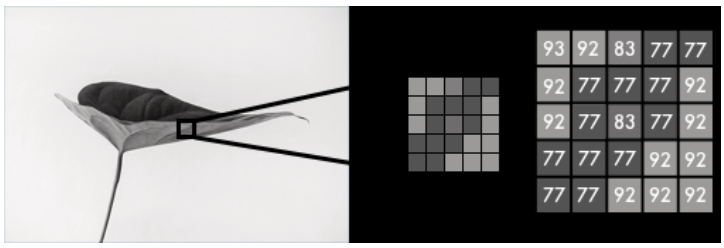
I feel the idea of colorizing grayscale images is something uniquely satisfying and helps bring life into a picture or a photograph. In this project I'm using the concept of artificial neural networks to colorize grayscale images. Black and white images can be represented in grids of pixels. Each pixel 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. To achieve the color white, for example, you need an equal distribution of all colors. In this project I'm using LAB color image format instead of RGB, since it easier to work with LAB.

A neural network is trained on thousands of color images to classify an image's black and white and LAB versions. So, when a random black white image is inputted to the neural network, it predicts its corresponding LAB version. When the LAB version is combined with the original grayscale image the RGB color version of the image is obtained.

IMAGE FORMATS

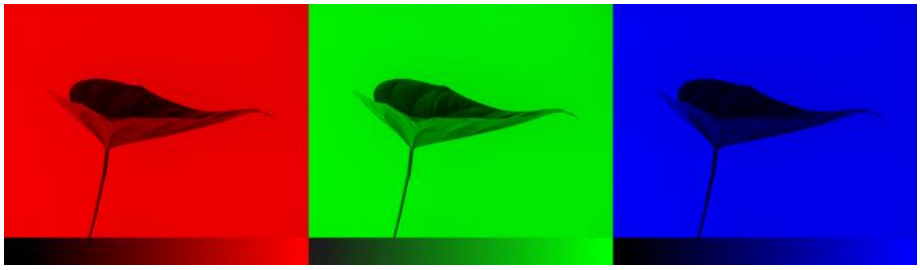
GRAYSCALE:

In a grayscale image each pixel varies from values 0-255, which determines how light or dark the pixel is.



RGB COLOR FORMAT:

In RGB color format each pixel is a combination of RED, GREEN, BLUE, whose individual values range from 0-255.



RED GREEN BLUE

Combination of certain values of the RGB layers determines the colour pixel.

LAB COLOR FORMAT:

The lab color format has three layers Light(L), a, b. The L layer determines the lightness of the pixel ranging from 0-100. The ‘a’ layer values span from -128 to 128, from green to red. The ‘b’ layer values span from -128 to 128, from blue to yellow. combination of the three layers gives the color image.

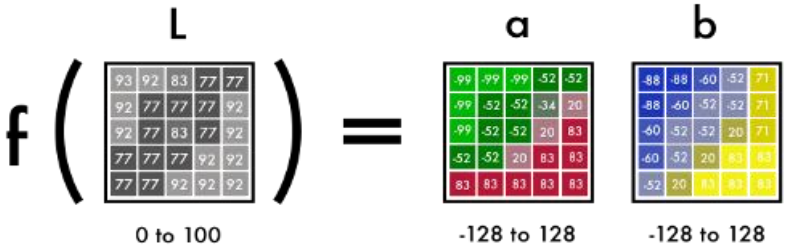
L (Lightness) ‘a’ ‘b’

CORE LOGIC

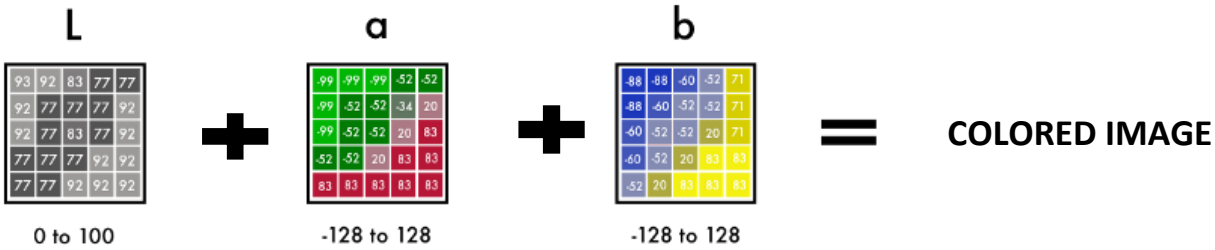
In this project I've used the LAB space since it is easier and less complex to perform computations in the LAB space compared to the RGB space. The RGB format uses 3 dimensional arrays to store each pixel's values whereas the LAB uses two dimensional arrays, because the L layer can be neglected since it's the same as the input grayscale image and there is no need to do computation on it. Computations on 2D arrays are simpler compared to 3D arrays and takes less training time.

The core logic of training is to extract the LAB format from the RGB color training image. Then the neural network is trained to link the L layer with the corresponding 'a', 'b' layers.

So, when a random Grayscale (i.e. equivalent to L layer) image is given as an input to the neural network model, it can predict its corresponding 'a', 'b' layers. Then the predicted 'a', 'b' layers are combined with the input grayscale image to get the desired color image.



F () is the trained neural network model



ALPHA VERSION

The ALPHA version of the project is the beginner version. Here I've used a simple multilayered Convolutional Neural network to train on the images.

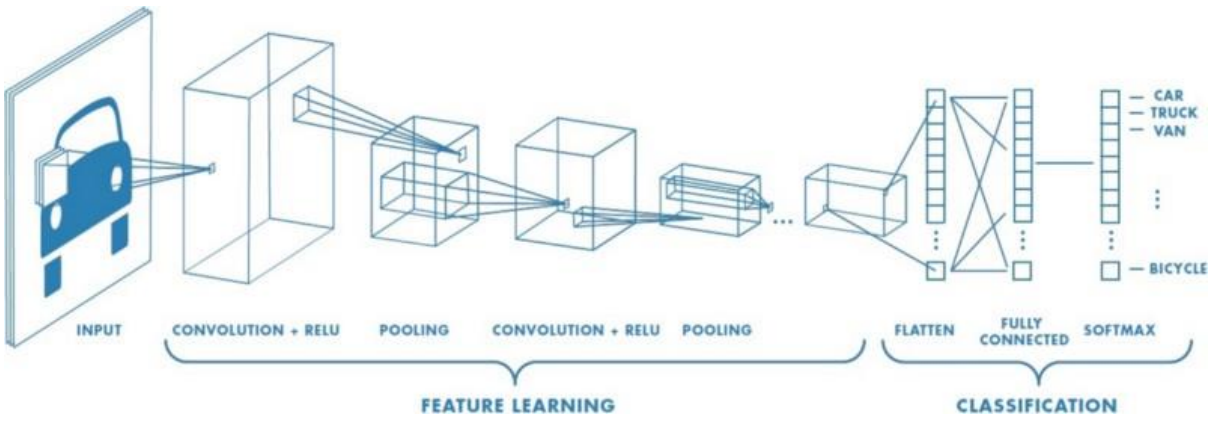
Training images are a bunch of colored images. In the alpha version, the neural network is trained with about 10 to 40 images.

CONVOLUTION NEURAL NETWORK

A convolutional neural network or CNN is a deep learning neural network mainly used for image classification. Here in our project we train multiple layers of CNN's to ultimately predict ab layers of the LAB color space for a given input grayscale image. Now let's get into details of how a CNN works.

A CNN is also one type of a multilayered neural network, what makes it different from the other neural networks like the multi layered perceptron neural network, are the hidden layers called the convolution layers.

The image below shows how a single CNN is structured.



The feature learning layers of the CNN are called the convolution layers, they are also called as filters. The main function of these filters is to extract features from the image like multiple edges, shapes, textures, objects etc. based on the features extracted, the classification layers of the CNN predicts the label for the input. In our case the label is a filtered image (can be multiple filtered images).

ALPHA VERSION NETWORK MODEL

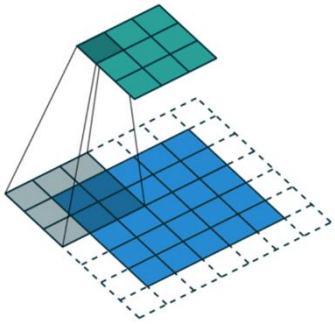
```
model = Sequential()
model.add(InputLayer(input_shape=(256, 256, 1)))
model.add(Conv2D(8, (3, 3), activation='relu', padding='same', strides=2))
model.add(Conv2D(8, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(16, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(16, (3, 3), activation='relu', padding='same', strides=2))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same', strides=2))
model.add(UpSampling2D((2, 2)))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(UpSampling2D((2, 2)))
model.add(Conv2D(16, (3, 3), activation='relu', padding='same'))
model.add(UpSampling2D((2, 2)))
model.add(Conv2D(2, (3, 3), activation='tanh', padding='same'))
```

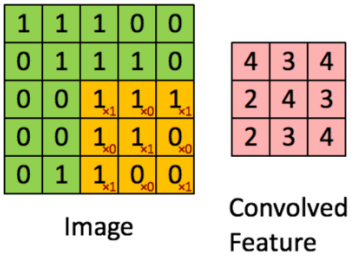
The above is the python script for the Alpha version network. It's a sequential network i.e.: output of one layer is an input to the next. It uses a series of 2-Dimensional Convolution neural networks. Conv2D python function is used to model the convolution network. Each Conv2D model uses a 3x3 filter and a ReLU activation function.

When strides=2, the model loses the certain amount pixels of the image hence Upsample2D is used to make up for the lost pixels. The output of the final Conv2D network are 2 filtered images, they are the predicted 'a', 'b' layers.

CONVOLUTION LAYER

The first layer in a CNN is always a Convolutional Layer. In the convolution layer a specific filter runs over the entire input image stride by stride and computes the sum of products of all the neurons and produces one single output neuron.





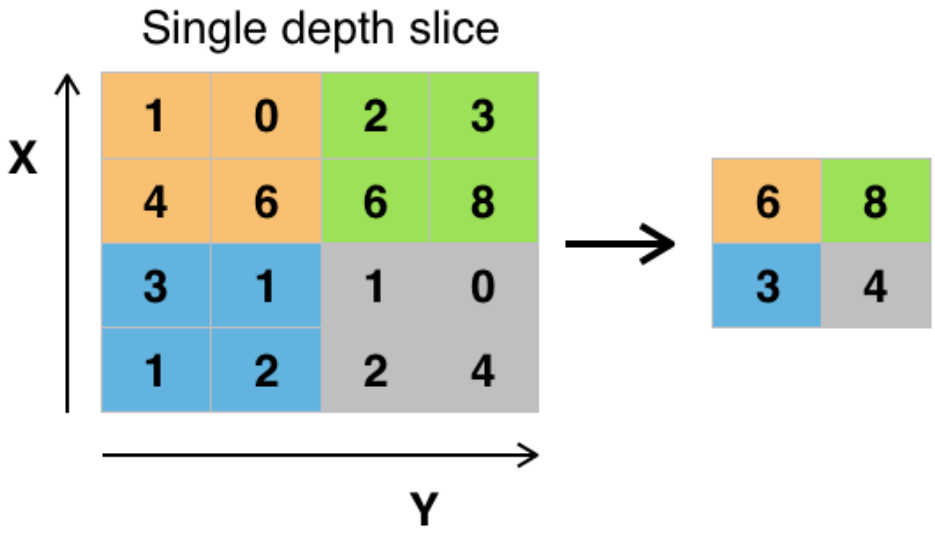
The objective of the Convolution Operation is to extract the high-level features such as edges, from the input image. CNN's need not be limited to only one Convolutional Layer. Conventionally, the first CNN Layer is responsible for capturing the Low-Level features such as edges, color, gradient orientation, etc. With added layers, the architecture adapts to the High-Level features as well, giving us a network, which has the wholesome understanding of images in the dataset.

ReLU (Rectified Linear unit) LAYER

After each convolution layer, it is a convention to apply a nonlinear layer (or **activation layer**) immediately afterward. The purpose of this layer is to introduce nonlinearity to a system. In the past, nonlinear functions like tanh and sigmoid were used, but researchers found out that **ReLU layers** work far better because the network is able to train a lot faster without making a significant difference to the accuracy. It also helps to alleviate the vanishing gradient problem, which is the issue where the lower layers of the network train very slowly because the gradient decreases exponentially through the layers. The ReLU layer applies the function f(x) = max (0, x) to all of the values in the input volume. In basic terms, this layer just changes all the negative activations to 0. This layer increases the nonlinear properties of the model and the overall network without affecting the receptive fields of the conv layer.

MAXPOOLING

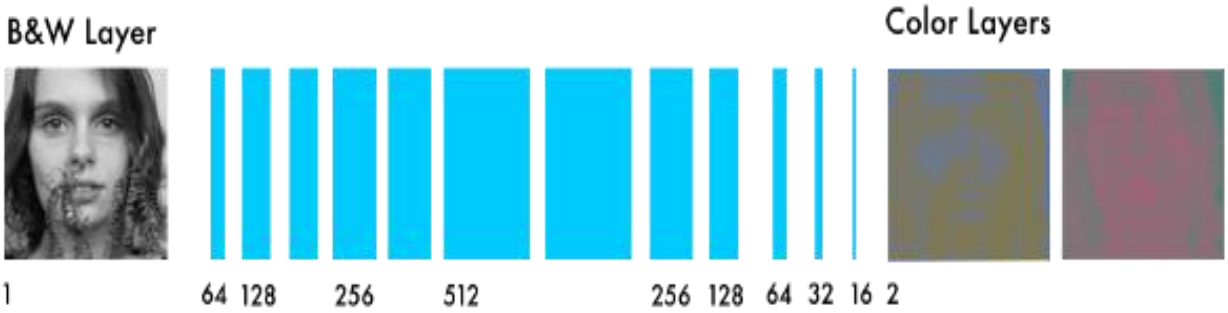
After some ReLU layers, I choose to apply a pooling layer. It is also referred to as a down sampling layer. In this category, there are also several layer options, with maxpooling being the most popular. This basically takes a filter (normally of size 2x2) and a stride of the same length. It then applies it to the input volume and outputs the maximum number in every subregion that the filter convolves around.



Other options for pooling layers are average pooling and L2-norm pooling. The intuitive reasoning behind this layer is that once we know that a specific feature is in the original input volume its exact location is not as important as its relative location to the other features. As you can imagine, this layer drastically reduces the spatial dimension of the input volume. This serves two main purposes. The first is that the number of parameters or weights is reduced by 75%, thus lessening the computation cost. The second is that it will control **overfitting**. This term refers to when a model is so tuned to the training examples that it is not able to generalize well for the validation and test sets.

IMPLEMENTATION OF CNN's IN ALPHA VERSION

In the ALPHA version multiple layers of CNN's are used to obtain the final predicted 'ab' filter image as shown below.

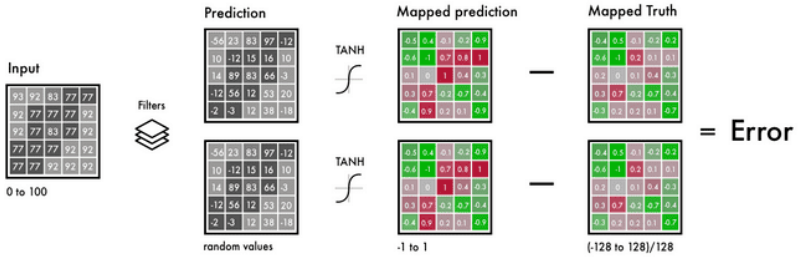


Each blue strip is a CNN and the number below them are the number of output filtered images.

The output of each CNN is passed to the preceding CNN, and the final CNN produces 2 filtered images 'ab' of the LAB space.

TRAINING

The output from the final CNN is mapped using a tanh activation function to obtain the final mapped prediction. The mapped prediction is compared with the mapped truth from the original color image. Based on the resulted error the weights of the neural network are changed using back propagation.



Python script for training:

```
model.fit(x=trainX, y=trainY, batch_size=1, epochs=300)

print(model.evaluate(trainX, trainY, batch_size=1))
```

RESULTS FOR ALPHA VERSION

Test1:

training data: 10 human face images
input data: one of the human faces in black and white
epoch:100
batch size:1

Input image(256x256)

original image(256x256)

output image(256x256)

Test2:

training data: 40 pictures of Dogs
input data: not one of the 40 pictures, in black and white.
epoch=50
batch size=3

Input image(256x256)

original image(256x256)

output image(256x256)

DRAWBACKS OF ALPHA VERSION:

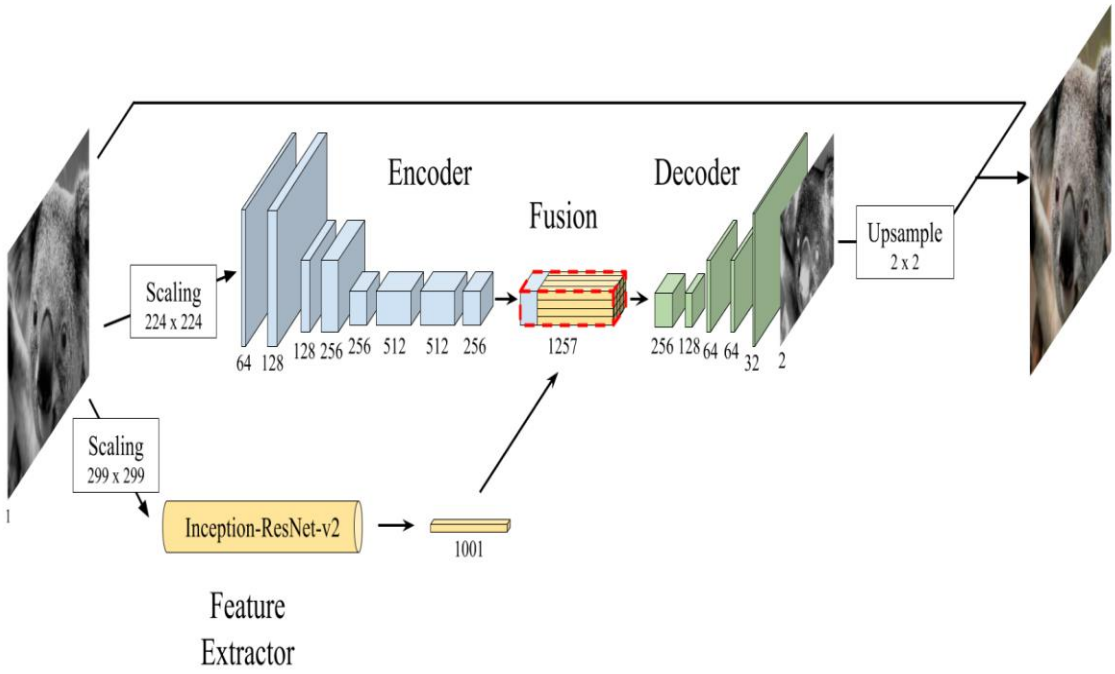
We can observe that in test1 the output image is almost like the original image, it is because the input image is one of the trained images. In test2 since the input image is not one of the trained images the output is not up to the mark. This drawback gives rise to a newer approach, this will be called the Beta version.

BETA VERSION

For the Beta version we took a different approach to tackle the drawbacks of the Alpha version. The beta version has four components, Encoder, Feature extractor, Fusion, Decoder.

We split the network we used in the alpha version 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 a classifier network model, the Inception ResNet v2. This is a neural network trained on 1.2Million images. We extract the information from the classification layer and merge it with the output from the encoder using the Fusion.

By using the learnings from the classifier in the coloring network, the network can now get a better sense of what's in the picture. Thus, enabling the network to match an object representation with a coloring scheme.



ENCODER

Initially the RGB training image is converted to Lab space. The L layer of the Lab space is then passed to the encoder and the Inception ResNet v2 networks.

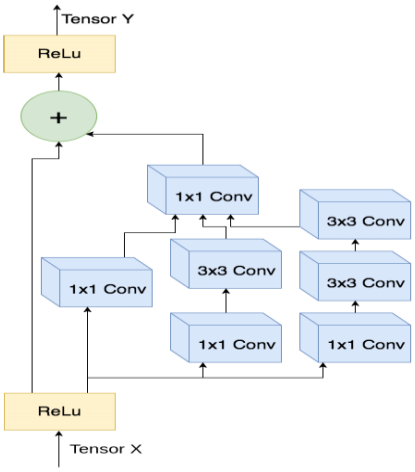
The encoder filters the grayscale image (L layer) in eight convolution neural network layers as shown in the picture above. Each layer of network outputs the corresponding number of filtered images as shown above (64,128,128,256,256,512,512,256). the output of the final layer of the encoder is 256 filtered images.

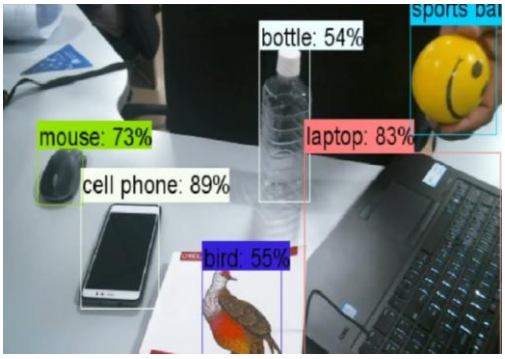
Python script for Encoder:

```
encoder_input = Input(shape=(256, 256, 1,))
encoder_output = Conv2D(64, (3,3), activation='relu', padding='same',
strides=2)(encoder_input)
encoder_output = Conv2D(128, (3,3), activation='relu',
padding='same')(encoder_output)
encoder_output = Conv2D(128, (3,3), activation='relu', padding='same',
strides=2)(encoder_output)
encoder_output = Conv2D(256, (3,3), activation='relu',
padding='same')(encoder_output)
encoder_output = Conv2D(256, (3,3), activation='relu', padding='same',
strides=2)(encoder_output)
encoder_output = Conv2D(512, (3,3), activation='relu',
padding='same')(encoder_output)
encoder_output = Conv2D(512, (3,3), activation='relu',
padding='same')(encoder_output)
encoder_output = Conv2D(256, (3,3), activation='relu',
padding='same')(encoder_output)
```

INCEPTION RESNET V2 NETWORK

The Inception Resnet v2 network is a simple object detection network consisting of a series of convolution layers each with a specific filter resolution and ReLU activation layers at the beginning and the end of the network. The network is trained on 1.2Million images. In parallel to the encoder the Inception ResNet v2 network classifies the objects in the input grayscale image and gives the entire system a better understanding of what is being colored.





Python script to load the Resnet V2 network weights:

```
inception = InceptionResNetV2(weights='imagenet', include_top=True)

inception.graph = tf.get_default_graph()

embed_input = Input(shape=(1000,))
```

FUSION LAYER

The learnings of the Inception Resnet V2 network are combined with filtered images of the encoder network in the fusion layer.

Python script for Fusion layer:

```
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)
```

DECODER

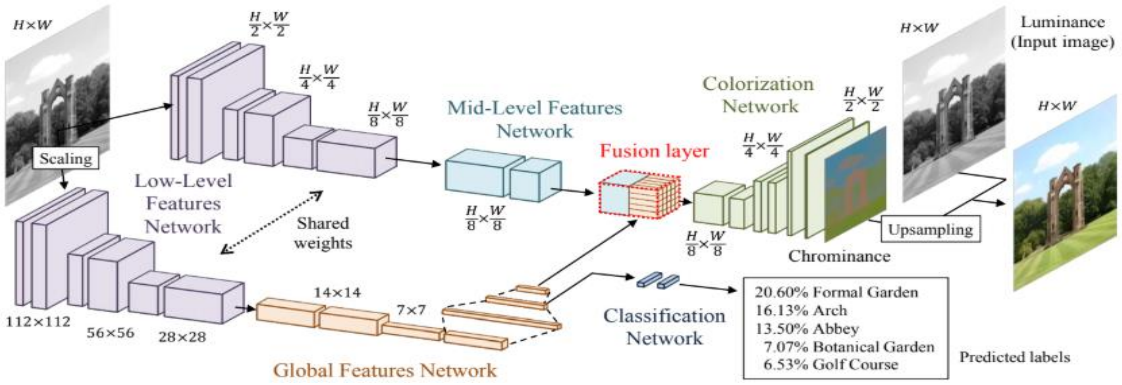
The fused data is then passed to the Decoder where the deciphering happens in six layers as shown in the picture above. The final output of the decoder is the predicted ‘ab’ layers of the Lab space.

The final output from the decoder is combined with the input L layer to obtain the final Lab color image.

Python script for Decoder:

```
decoder_output = Conv2D(128, (3,3), activation='relu', padding='same')(fusion_output)
decoder_output = UpSampling2D((2, 2))(decoder_output)
decoder_output = Conv2D(64, (3,3), activation='relu', padding='same')(decoder_output)
decoder_output = UpSampling2D((2, 2))(decoder_output)
decoder_output = Conv2D(32, (3,3), activation='relu', padding='same')(decoder_output)
decoder_output = Conv2D(16, (3,3), activation='relu', padding='same')(decoder_output)
decoder_output = Conv2D(2, (3, 3), activation='tanh', padding='same')(decoder_output)
decoder_output = UpSampling2D((2, 2))(decoder_output)
```

DETAILED VIEW



The above detailed picture shows that from the input image the Inception Resnet v2 network has collected information, for ex: - 20.60% Formal garden, 16.3% Arch, 13.50% Abbey, 7.07% Botanical garden, and 6.53% Golf course. This information is then used by the decoder to match the color combinations with corresponding object.

TRAINING

The training of the Beta network is similar to the Alpha network. The Lab color image obtained from the decoder Is compared to true Lab space of the training image, the difference between them is the error. based on the error the weights of the network are adjusted.

Python script to train Beta version:

```
model.compile(optimizer='rmsprop', loss='mse')
model.fit_generator(image_a_b_gen(batch_size),epochs=10000,steps_per_epoch=1,callbacks=[tensorboard])
```

RESULTS FOR BETA VERSION

Test1:

training data: 31 color images
input data: grayscale image of one of the training images.
epoch:10000
batch size:10

Input image(256x256)

original image(256x256)

output image(256x256)

Test2:

training data: 31 color images
input data: a random grayscale image (not a part of the training image set).
epoch=10000
batch size=10

Input image(256x256)

original image(256x256)

output image(256x256)

Test3:

training data: 20 color images
input data: a random grayscale image (not a part of the training image set).
epoch=10000
batch size=10

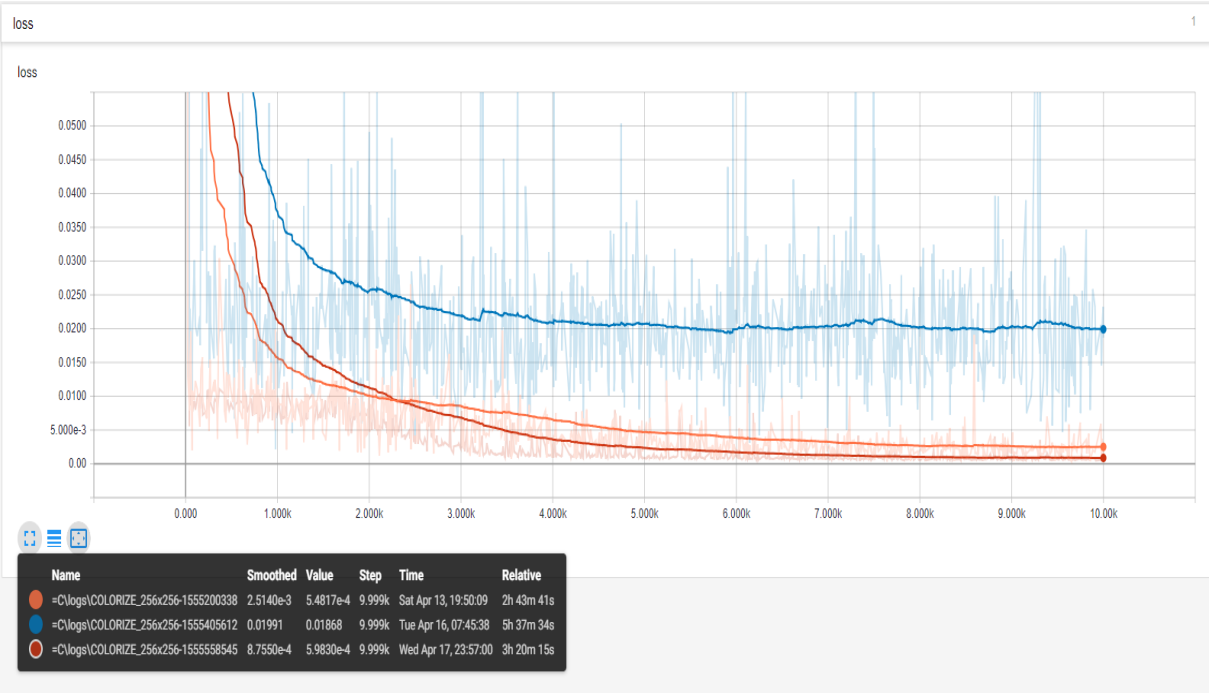
Input image(256x256)

original image(256x256)

output image(256x256)

From the above three tests we can observe that the network works flawlessly for input images that it was trained on. In test2 when a random grayscale input image is fed to the network trained on 31 images, it does a fairly good job of colorization. But when the same test image is passed through a network that is trained on 20 color images the network does a far better job at colorization. This can be further analyzed in detail from a graphical perspective in the next slide.

GRAPHICAL ANALYSIS OF BETA VERSION



The above picture is a graphical plot of loss vs epochs for three different training setups.

30 training images, 10000 epochs, mean loss 0.0025

123 training images, 10000 epochs, mean loss 0.0200

20 training images, 10000 epochs, mean loss 0.0008

It can be inferred from the graphical plot that the overall mean loss is inversely proportional to the number of training images. The network performs better when the overall mean loss is lesser, which can be achieved by training the network with least number of training images. But when a network is trained on a smaller number of training data it learns well but won't learnt everything i.e. that network will find it hard to implement a wide range of colors and objects.

Further research is done to find the perfect balance between the number of training images and the overall mean loss, which will help the network perform better and for a wide range of color images.

MORE VALIDATION IMAGES

training data: 20 color images
input data: a random grayscale image (not a part of the training image set).
epoch=10000
batch size=10

Original image

Output image

Original image

Output image

Software's used

- PyCharm IDE for python programming
- Tensorboard for plotting data

Python libraries used

- "TensorFlow", "Keras" for image processing
- "Skimage" for image resizing
- "Numpy" for arrays
- "os" for file operations
- "time" for timestamping

References

- https://medium.freecodecamp.org/colorize-b-w-photos-with-a-100-line-neural-network-53d9b4449f8d
- https://github.com/emilwallner/Coloring-greyscale-images
- Federico Baldassarre, Diego Gonź alez Morín, Lucas Roď es-Guirao, Deep Koalarization: Image Colorization using CNNs and Inception-Resnet-v2, arXiv:1712.03400v1 [cs.CV] 9 Dec 2017.

Training image source

- https://www.pexels.com/search/fashion/