Image Colorization Using Deep Learning

Using deep neural networks to colorize black & white images.

Project by:

Group 19

Kaushal Lodd BT19CSE052

Saarang Rajguru BT19CSE092

Karan Sujan Valappil BT19CSE048

Abstract

Colorization of black & white pictures is conventionally done manually, requiring hours of manual labor. Furthermore, colorizing many images at once, or colorizing videos manually is very difficult or next to impossible.
With the advent of neural networks and deep learning techniques, the possibility of automatically colorizing images has emerged.
The goal of this project is to provide completely colorized images from simple black & white images using deep learning / neural networks and with high accuracy.

Problem Statement

Taking a black and white image as input, produce a colored image using deep neural networks with a high rate of accuracy.

Dataset

For this project, the dataset requirements are very flexible, since we can use any dataset of images and extract grayscale data from them, effectively creating two images, one grayscale and the other colorized.

However, to keep preprocessing at a minimum and to maintain a uniform result, it is better for the images to be of the same resolution and of a similar color profile, preferably of the same origin.

Hence, the dataset we used is the "*MIT Places Database For Scene Recognition*" released by the Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, USA. It contains over 2.4 million images with category labels, a subset of which we will use for this project.

A preview of the MIT Places Database - Aqueduct

A preview of the MIT Places Database - Army Base

Methodology

Overview

In image colorization problems, our goal is to produce a colored image given a grayscale input image.

Intuitively, this might sound impossible since a given grayscale value can correspond to a number of color values. For this reason, colorization of images was traditionally done with a high degree of human involvement or manual work.

In recent years, this problem has been solved with great success using deep learning and neural networks.

Breaking down the problem statement

Let us first answer the question, how do we represent color? There are two ways of going about this:

1. RGB values: Each pixel has a set of three different values **R**ed, **G**reen and **B**lue which range from 0 to 255. Since every color can be broken down into these three basic colors, this can be an accurate representation of color.

2. LAB, or Lightness, A, B: **L** stands for lightness, and **a** and **b** for the color spectra green–red and blue–yellow.
   We will use this scheme, since it makes it easier for us to separate out the Lightness property, which is the most important in colorization. So, if we have input of RGB values, we will need to pre-process these to obtain LAB values. We can do this using existing libraries easily.

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

# Generate training data
batch_size = 10
def image_a_b_gen(batch_size):
    for batch in datagen.flow(Xtrain, batch_size=batch_size):
        lab_batch = rgb2lab(batch)
        X_batch = lab_batch[:,:,:,0]
        Y_batch = lab_batch[:,:,:,1:] / 128
        yield (X_batch.reshape(X_batch.shape+(1,)), Y_batch)
```

Code used for preprocessing, to convert RGB into LAB.

The Model

The model we use here is a convolutional neural network (CNN).

Every convolutional filter or convolutional layer extracts a certain feature from the image, i.e. downscales it to reveal a particular feature it contains. Hence they can be used to highlight a certain feature, or isolate a certain feature as desired.

For a convolutional neural network, each filter is automatically adjusted to help with the intended outcome.

Then, we apply deconvolutional layers to upscale (increase the spatial resolution) of our features.

```
model = Sequential()
model.add(InputLayer(input_shape=(256, 256, 1)))
model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(64, (3, 3), activation='relu', padding='same', strides=2))
model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(128, (3, 3), activation='relu', padding='same', strides=2))
model.add(Conv2D(256, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(256, (3, 3), activation='relu', padding='same', strides=2))
model.add(Conv2D(512, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(256, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(UpSampling2D((2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(UpSampling2D((2, 2)))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(2, (3, 3), activation='tanh', padding='same'))
model.add(UpSampling2D((2, 2)))
model.compile(optimizer='rmsprop', loss='mse', metrics=['accuracy'])
✓ 0.1s                                                          Python
```

Training

Loss Function: Since we are doing regression, we'll use a mean squared error loss function: we minimize the squared distance between the color value we try to predict, and the true (actual) color value.

Activation Function: To map the predicted values, we use a tanh activation function. For any value we give the tanh function, it will return -1 to 1.
We do this because the true color values range between -128 and 128. By dividing them by 128, they fall within the -1 to 1 interval. This “normalization” enables us to compare the error from our prediction.

We take the input image and run it through our trained neural network. We take all the output values between -1 and 1 and multiply it by 128. This gives us the correct color in the Lab color spectrum.

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.

We run the model across multiple epochs to complete training.

Results

Here we can see the results obtained after running the trained model on our testing dataset.

The results are not perfect and don't look exactly like their original images, but we have a fair degree of accuracy of around 83.4%.