Ranabir Devnath

Image Colorization Using Autoencoder

August 12, 2021

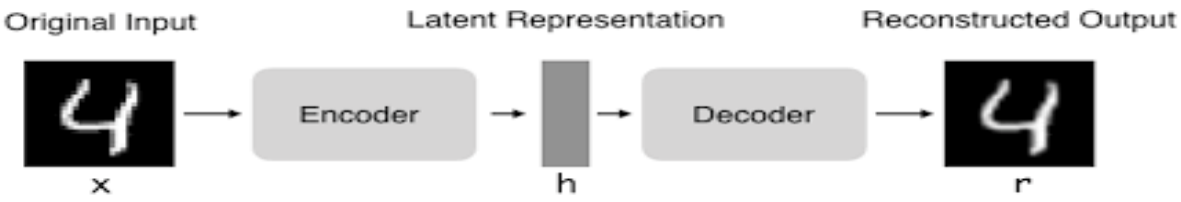
Problem Statement

The main task of this project is to colorize old black and white images. Often we found old images of our family members or other important images. Often we find it is helpful if we can the colorize version of that image or do fun stuff with it. So, basically using machine learning or especially Convolutional Neural Networks from deep learning, we can do this stuff using the concept of AutoEncoder.

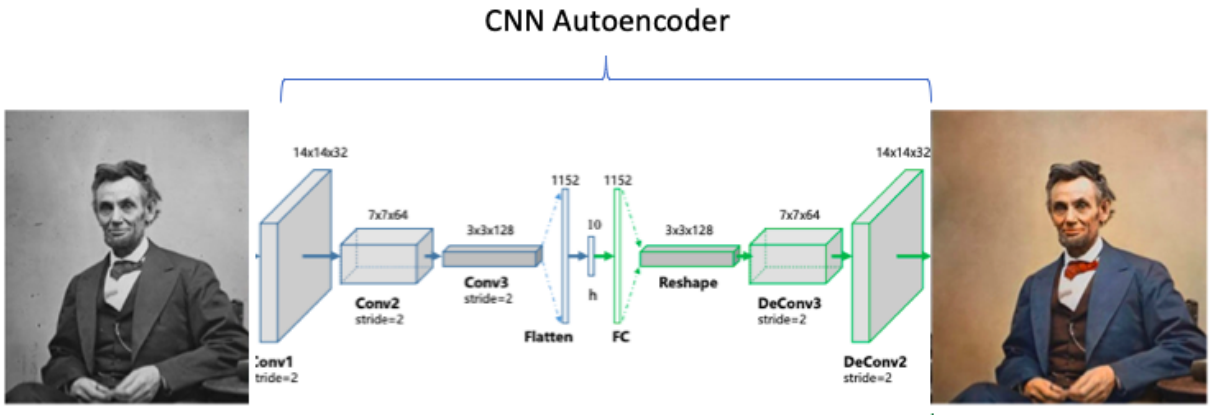
This project is completed using the concept of AutoEncoder and Transfer Learning. A pre-trained model named VGG16 is used for this project. The notebook can be found here.

Preliminaries

The main idea of an autoencoder is that it produces or re-creates an output from the given input. For example, one can train the model to create the numerical number “1” on output by giving it the numerical number “1” as input. An example is given below of colorization using Autoencoder. An example is given below of colorization using Autoencoder.

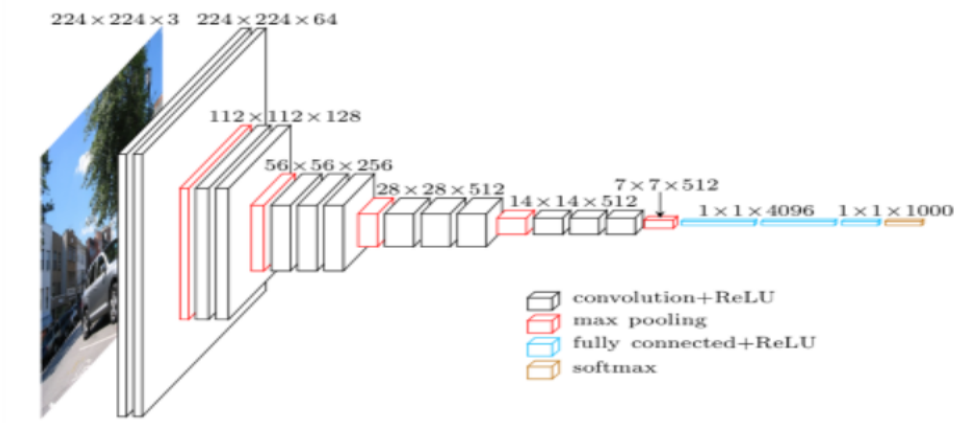


Here is a simple example of Image Colorization using AutoEncoder.



An Autoencoder architecture has 3 components. Encoder, Latent Space, and Decoder. The encoder compresses the input and produces the latent image representation, the decoder then reconstructs the input only using this vectorized representation.

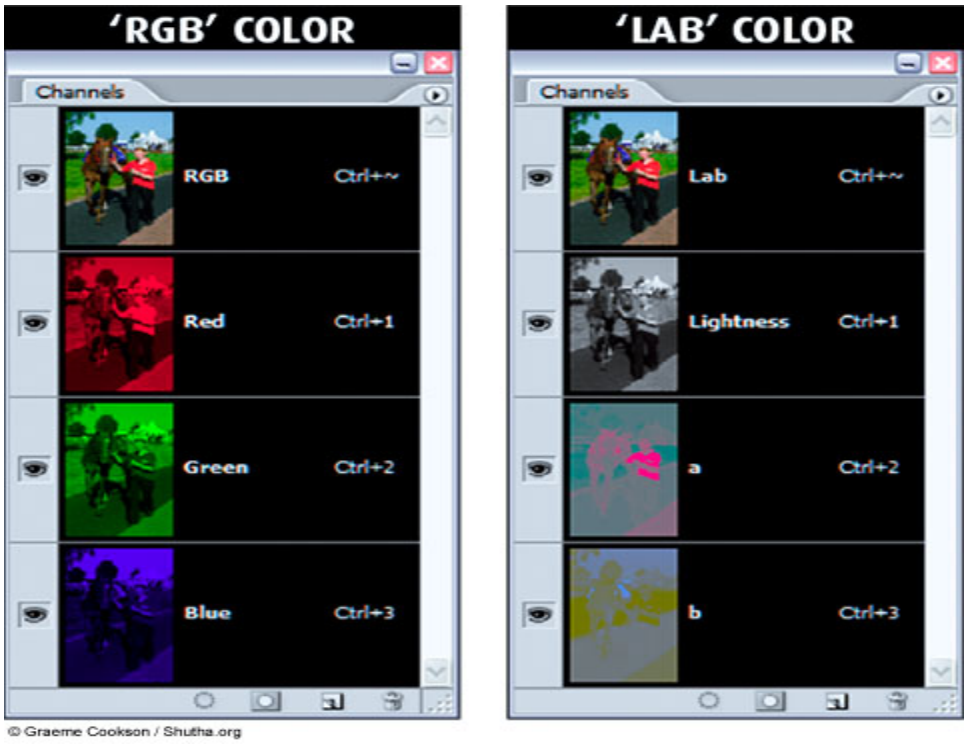
Since it is an image-based project so CNN architecture can be used. So, I used a pre-trained model named VGG16 which was trained on "imagenet" dataset. It has nearly 138 million trainable parameters.



Architecture of VGG16

This pre-trained model will be used as a feature extractor for the encoder part to extracts features from the given input. But since the encoder part is going to be replaced by the

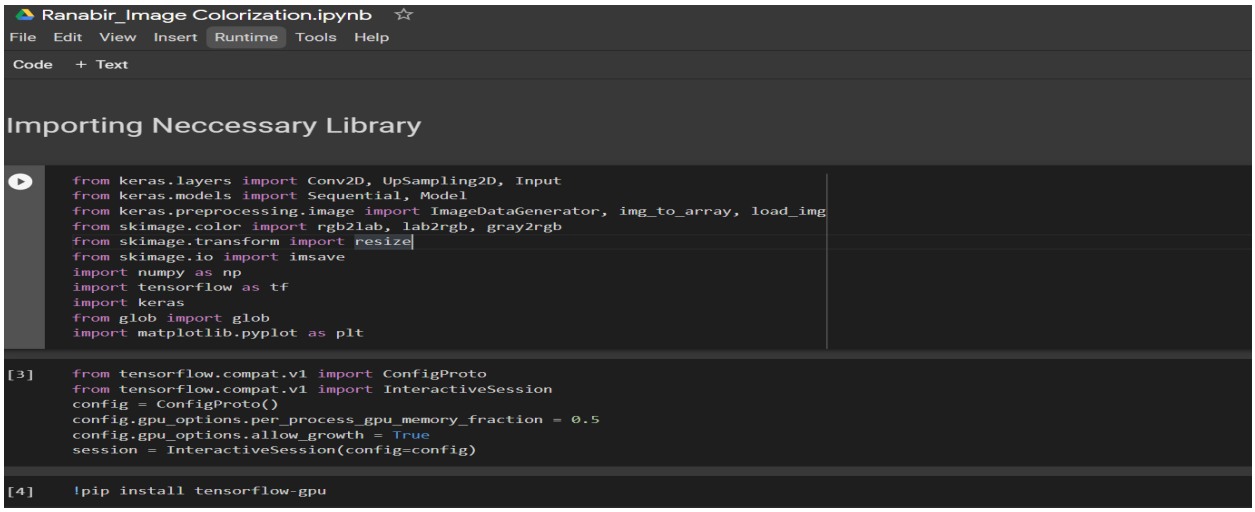
pre-trained VGG, so the classifier part is not needed. So as a result the last dense layers have to be popped up. And also as it has a large number of parameters, training again these parameters will take a lot of time and resources. So, trainable parameters are frozen. An illustration will be given later on in the report.



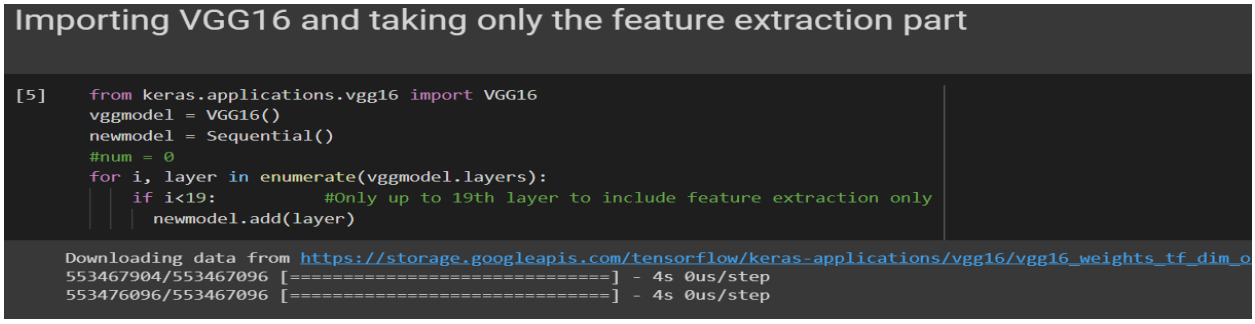
LAB is another format of an image like RGB. An RGB image is easily understood as there are three logical colors. But 'Lab' has a mix of one channel with no color (L), plus two channels with a dual-color combination that has no contrast (a+b). The 'L' channel, or Lightness, is the easiest to understand as it is a Greyscale. It has no color value at all; it just contains the contrast between the lightest and darkest points in the image. Our training data is in RGB format, so we can use its extracted "L" channel for training the data. As a result, we don't need extra steps to convert "RGB" images to "GRAY".

These are the basic concepts needed to understand this project. Further demonstration of this project is described below.

Project Demonstration



These are the necessary libraries needed to run the code.



Here VGG16 is loaded. Since I will use this model as a feature extractor, I iterated on each layer except the last dense layers so, I added 19 layers to my model. The dimension of the last layer volume is “7x7x512”. That latent space volume will be used as a feature vector to be input to the decoder and the decoder is going to learn the mapping from the latent space vector to “a” and “b” channels.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
```

Here is the model summary.

```
[7]   #We don't want to train these layers again, so False.
      for layer in newmodel.layers:
        layer.trainable=False
```

I don’t want the layers of VGG16 with its original weights without changing them, so the trainable parameter in each layer is false because I don’t want to train them again.

```
[29]  #Normalize images - divide by 255
      train_datagen = ImageDataGenerator(rescale=1. / 255)

      train = train_datagen.flow_from_directory(path, target_size=(224, 224), batch_size=32, class_mode=None)
      test = train_datagen.flow_from_directory(test_path, target_size=(224, 224), batch_size=32, class_mode=None)
```

```
Found 3000 images belonging to 1 classes.
Found 20 images belonging to 1 classes.
```

So, here all the images are loaded from the drive and converted into 224x224 dimensions.

Converting RGB image to lab image for feature extraction

```
[10]  X =[]
      Y =[]
      for img in train[0]:
        try:
            lab = rgb2lab(img)
            X.append(lab[:,:,0])
            Y.append(lab[:,:,1:] / 128) #A and B values range from -127 to 128,
            #so we divide the values by 128 to restrict values to between -1 and 1.
        except:
            print('error')
      X = np.array(X)
      Y = np.array(Y)
      X = X.reshape(X.shape+(1,)) #dimensions to be the same for X and Y
      print(X.shape)
      print(Y.shape)
```

```
(32, 224, 224, 1)
(32, 224, 224, 2)
```

By iterating on each image, I converted them RGB to Lab. Think of the LAB image format as a grey image in L channel and all color info stored in A and B channels. The input to the network will be the L channel, so we assign L channel to X vector. And assign A and B to Y.

Now all is set, it is time to construct the model.

```
[12]  #Encoder
      vggfeatures = []
      for i, sample in enumerate(X):
        sample = gray2rgb(sample)
        sample = sample.reshape((1,224,224,3))
        prediction = newmodel.predict(sample)
        prediction = prediction.reshape((7,7,512))
        vggfeatures.append(prediction)
      vggfeatures = np.array(vggfeatures)
      print(vggfeatures.shape)
```

```
(32, 7, 7, 512)
```

So, here the encoder part is ready and the output dimension of the VGG model is you can see 7x7x512. This we now go through a decoder.

```
[13]  #Decoder
      model = Sequential()

      model.add(Conv2D(256, (3,3), activation='relu', padding='same', input_shape=(7,7,512)))
      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(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'))
      model.add(UpSampling2D((2, 2)))
      model.summary()
```

Here is the Decoder part of the network. 'RELU' activation is used and sequential model of course is used. Now let's compile the model.

```
[14]  #Model Compile
      model.compile(optimizer='Adam', loss='mse' , metrics=['accuracy'])
```

As you can see, the "ADAM" optimizer, "MSE" loss function is used. And as evaluation metrics, "ACCURACY" is used. Now, all set for training, let's go for training.

```
model.fit(vggfeatures, Y, verbose=1, epochs=600, batch_size=128)
```

```
Epoch 592/600
1/1 [==============================] - 1s 967ms/step - loss: 0.0024 - accuracy: 0.8041
Epoch 593/600
1/1 [==============================] - 1s 943ms/step - loss: 0.0024 - accuracy: 0.8122
Epoch 594/600
1/1 [==============================] - 1s 900ms/step - loss: 0.0023 - accuracy: 0.8112
Epoch 595/600
1/1 [==============================] - 1s 909ms/step - loss: 0.0024 - accuracy: 0.8059
Epoch 596/600
1/1 [==============================] - 1s 924ms/step - loss: 0.0024 - accuracy: 0.8125
Epoch 597/600
1/1 [==============================] - 1s 933ms/step - loss: 0.0023 - accuracy: 0.8107
Epoch 598/600
1/1 [==============================] - 1s 933ms/step - loss: 0.0023 - accuracy: 0.8082
Epoch 599/600
1/1 [==============================] - 1s 940ms/step - loss: 0.0024 - accuracy: 0.8125
Epoch 600/600
1/1 [==============================] - 1s 902ms/step - loss: 0.0023 - accuracy: 0.8103
<keras.callbacks.History at 0x7fec276127d0>
```

The model iterated up to 600 and got an accuracy between 75-80%.

```
[35]  # saving the model
      from tensorflow.keras.models import load_model
      model.save('/content/drive/MyDrive/Work/Quantum AI Course/Assignments/CNN/Dataset/small/colorization_epoch-600_acc-81.h5')
```

The model is also saved for future use. Now it's testing time.

```
test_path = '/content/model_train/0a0aa9e4a5d037e5.jpg'
i = 0
# for idx, image in enumerate(test_path):
test = img_to_array(load_img(test_path))
test = resize(test, (224,224), anti_aliasing=True)
test*= 1.0/255
lab = rgb2lab(test)
l = lab[:,:,0]
L = gray2rgb(l)
L = L.reshape((1,224,224,3))
vggpred = newmodel.predict(L)
ab = model.predict(vggpred)
ab = ab*128
cur = np.zeros((224, 224, 3))
cur[:,:,0] = l
cur[:,:,1:] = ab
imsave('/content/model_result/result'+ 'vgg' +".jpg", lab2rgb(cur))
i+=1
```

Here's how the testing can be done. Here's some result from the model.

Input

Output

Input

Output

Input

Output

Input

Output

Though the result is not that good it can be improved by tuning the hyperparameters.