DeepColor: Automatic Colorization of Gray scale Images

Abstract

Image colorization refers to assigning color to pixels of the black-and- white or gray scale images. Colors help gain a deeper understanding and help determine salient features associated with the image data. The task of Image Colorization has gained significant attention among the computer vision community over the last decade, and is often posed as a difficult problem due to the high degree of freedom associated with assigning the color pixels. This project aims to colorize gray-scale images using a neural network, using embedding's from Google's Inception ResNet v2 [2]. We are recovering color information using only the luminous intensity information (CIE Lab space). The project is inspired by the works "Deep Koalarization" [4], "Colorful Image Colorization" [7] and "Automatic Colorization of Grayscale Images" [6]. The performance of the proposed model has been evaluated using loss and accuracy. The proposed model is able to generate generalization accuracy of 56.8752%.

1. Introduction

Image colorization is a regression task, where we assign a color value to the target gray-scale image. Colorization of gray-scale images can add much more to the semantics of the image, and can help in enhancing historical image data, colorize black and white movies, improve the performance of surveillance tasks, to name a few. The colorization techniques are broadly divided into two types, scribble-based and example based colorization techniques. The scribble-based technique requires the user to provide a significant amount of scribble for the gray scale image data, and thus is extremely painstaking. The example-based colorization technique requires an example/reference image to be passed for every image, and transfers the color information from the reference image to the target image. However, it is not always easy to find a reference image for an input image, and there are no guarantees that the reference image selected would lead to a desirable target image.

Deep learning techniques for image data have flourished over the past decade and the results obtained have become state-of-the-art. Deep learning has also been used generously to solve the problem of image colorization, from earlier methods which applied brute force to more recent methods using Generative Adversarial Networks (GAN) [5].

In this project, we aim to explore the performance of neural networks for a task which is extremely painstaking and takes a long time for humans to do manually. Our network takes in a gray-scale image as input and produces a colored image as output. The data set employed for the exploration contains L, a and b components of the image, which are in the form of numpy files.

Our major contribution in this project is (a) enhancing the performance of the task of choice, which is colorization of gray scale images, by employing parameter optimization, and (b) carrying out a detailed analysis on the results obtained.

2. Related Works

Our major inspiration for this project was from "Deep Koalarization: Image Colorization using CNNs and Inception-Resnet-v2". Baldassarre et al. [4] extracts high-level features from the images using Inception-ResNet v2 and combines them with a Convolutional Neural Network, which was trained from scratch. The model architecture consisted of four major components, (1) Encoder to extract the mid-level features, (2) Inception-ResNet-v2 Feature extractor to extract high-level features, (3) Fusion layer to fuse these two features together, and (4) Decoder to predict the output using these features.

Zhang et al. [7] also utilized an encoder and decoder Convolutional Neural Network architecture for the task for gray-scale image colorization. However, interestingly, they have restructured this problem as a classification problem and have used bins as a color state, and have not used Mean Square Error (MSE) loss. They have even accounted for an even color distribution and have taken active measures do that their network learns not only the common colors, but the rarer colors as well, and therefore have multiplied a weight vector for every pixel which weighted dull colors with a greater loss, to the output of the softmax loss.

Sousa et al. [6] applied example-based approach, and implemented colorization of gray-scale images by using machine learning methods and a color map selected from a similar color image. They made the color space discrete and converted the problem into a classification problem. The classification was performed using support vector machines, one SVM per bin of the discrete color space, to perform one VS all classification.

An et al. [3] utilized VGG-16 Convolutional Neural Network for the task of colorizing gray scale images. The problem was treated as a classification task and cross-entropy loss was used. Their approach produced plausible colored images.

3. Approach

We are using images of size H x W in the CIE L*a*b color space. The CIE lab space contains three channels: L for luminance/ lightness, a and b are the color components representing green-red and blue-yellow respectively. The model uses the luminance component $X_L \in \mathbb{R}^{h\times w\times 1}$ to estimate the other two components to generate a colored version $X \in \mathbb{R}^{h\times w\times 3}$, that is a colored image with 3 channels for red, green and blue. Figure 1 shows the CIE Lab color space.

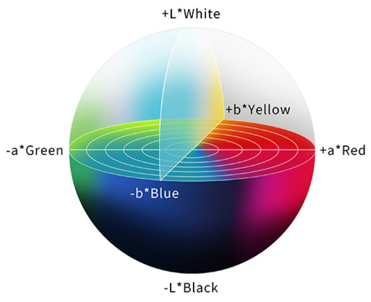


Figure 1. CIE Lab color space.

Therefore, we obtain the colored components of the image X_a and X_b using X_L, and then use these components together to obtain the colored image X. This can be shown mathematically, assuming a function $\mathcal{F}$ which does the following

$$\mathcal{F} : X_L \to (X_a, X_b)$$

where X_a and X_b represent the a* and b* components of the image. We get the colored image X using all the three image components

$$X = (X_L, X_a, X_b)$$

The architecture is based on Convolutional Neural Networks (CNN) as they are independent of the input size of the image. CNNs are a special type of artificial neural network specifically designed to work with pixel data. Figure 2 shows the Convolutional filter.

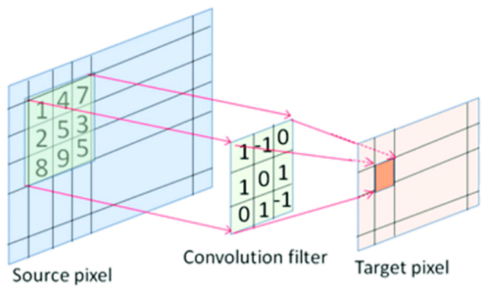


Figure 2. Convolutional filter.

We have used weights of Google's Inception-ResNet v2 for this experiment. It has been trained on more than a million images on the ImageNet dataset, and has 1000 classes. It is a 164 layers deep network, and has an input size of 299 x 299. Figure 3 shows the compressed version of the Inception-Resnet v2.

We have used a model on top of Inception-Resnet v2 to train our model to do image colorization. We generate embeddings by passing images through the Inception-Resnet v2 to further improve the performance of the system by concatenating these images with the encoder output in the fusion layer.

We tried variations of layers of encoders, fusion and decoders layers to optimize the test accuracy. We further tried to tweak the hyper-parameters like learning rate, batch size, regularization to train the model. We further plotted graphs of training accuracy, training loss for various use cases as well as calculated test and validation accuracy for them to achieve the best performance of all the cases.

We plot the model layers along with data of various use cases as follows along with learning rate, regularization strength and batch size.

3.1. Architecture

We are proposing a CNN structure consisting of an encoder, a decoder and a fusion layer which We have tested several architectures for this task. These are as follows.

Encoder: An encoder is a network (FC, CNN, RNN, etc) that takes the input, and output a feature map/vector/tensor. These feature vector hold the information, the features, that represents the input.

Fusion: Data fusion is the process of integrating multiple data sources to produce more consistent, accurate, and useful information than that provided by any individual data source. Fusion layers are added to combine 2 features encoder output and the embedded input which are created by passing input images through InceptionResnetV2.

Decoder: The decoder is again a network (usually the same network structure as encoder but in opposite orientation) that takes the feature vector from the encoder, and

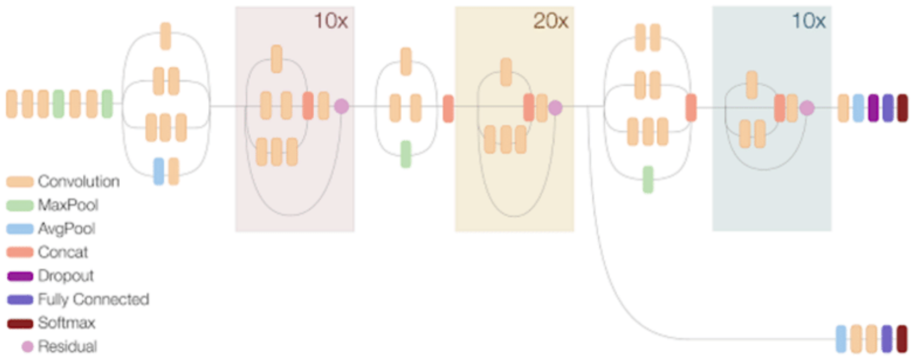
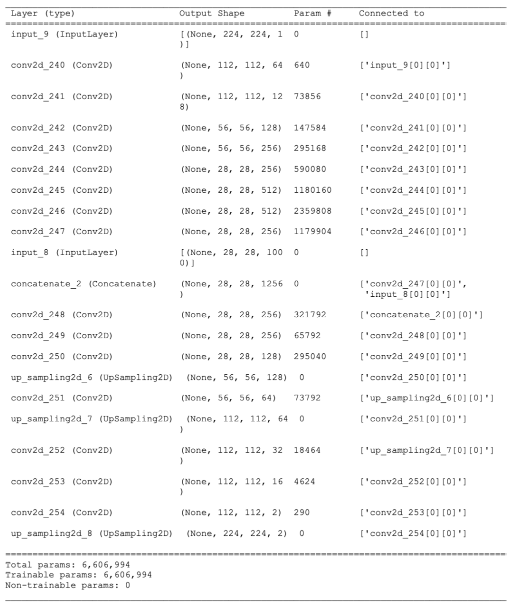
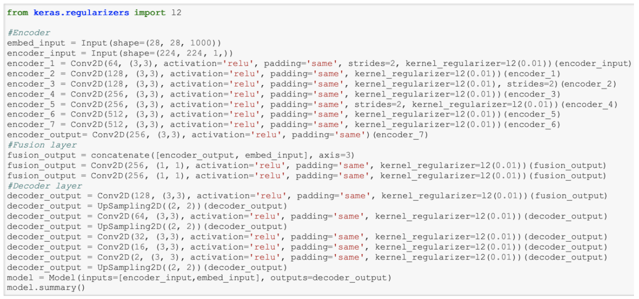


Figure 3. Inception-ResNet v2 compressed.



```
________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
================================================================================================
 input_9 (InputLayer)           [(None, 224, 224, 1  0           []
                                )]

 conv2d_240 (Conv2D)            (None, 112, 112, 64  640         ['input_9[0][0]']
                                )

 conv2d_241 (Conv2D)            (None, 112, 112, 12  73856       ['conv2d_240[0][0]']
                                8)

 conv2d_242 (Conv2D)            (None, 56, 56, 128)  147584      ['conv2d_241[0][0]']

 conv2d_243 (Conv2D)            (None, 56, 56, 256)  295168      ['conv2d_242[0][0]']

 conv2d_244 (Conv2D)            (None, 28, 28, 256)  590080      ['conv2d_243[0][0]']

 conv2d_245 (Conv2D)            (None, 28, 28, 512)  1180160     ['conv2d_244[0][0]']

 conv2d_246 (Conv2D)            (None, 28, 28, 512)  2359808     ['conv2d_245[0][0]']

 conv2d_247 (Conv2D)            (None, 28, 28, 256)  1179904     ['conv2d_246[0][0]']

 input_8 (InputLayer)           [(None, 28, 28, 100  0           []
                                0)]

 concatenate_2 (Concatenate)    (None, 28, 28, 1256  0           ['conv2d_247[0][0]',
                                )                                 'input_8[0][0]']

 conv2d_248 (Conv2D)            (None, 28, 28, 256)  321792      ['concatenate_2[0][0]']

 conv2d_249 (Conv2D)            (None, 28, 28, 256)  65792       ['conv2d_248[0][0]']

 conv2d_250 (Conv2D)            (None, 28, 28, 128)  295040      ['conv2d_249[0][0]']

 up_sampling2d_6 (UpSampling2D)  (None, 56, 56, 128)  0          ['conv2d_250[0][0]']

 conv2d_251 (Conv2D)            (None, 56, 56, 64)   73792       ['up_sampling2d_6[0][0]']

 up_sampling2d_7 (UpSampling2D)  (None, 112, 112, 64  0          ['conv2d_251[0][0]']
                                )

 conv2d_252 (Conv2D)            (None, 112, 112, 32  18464       ['up_sampling2d_7[0][0]']
                                )

 conv2d_253 (Conv2D)            (None, 112, 112, 16  4624        ['conv2d_252[0][0]']
                                )

 conv2d_254 (Conv2D)            (None, 112, 112, 2)  290         ['conv2d_253[0][0]']

 up_sampling2d_8 (UpSampling2D)  (None, 224, 224, 2)  0          ['conv2d_254[0][0]']

================================================================================================
Total params: 6,606,994
Trainable params: 6,606,994
Non-trainable params: 0
________________________________________________________________________________________________
```

Figure 4. Base Architecture.



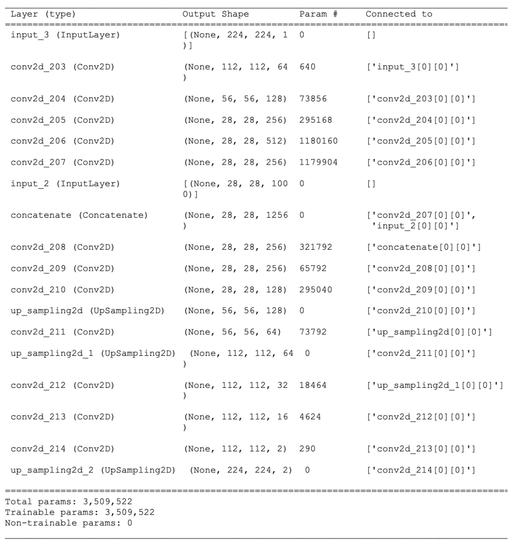
```
from keras.regularizers import l2

#Encoder
embed_input = Input(shape=(28, 28, 1000))
encoder_input = Input(shape=(224, 224, 1,))
encoder_1 = Conv2D(64, (3,3), activation='relu', padding='same', strides=2, kernel_regularizer=l2(0.01))(encoder_input)
encoder_2 = Conv2D(128, (3,3), activation='relu', padding='same', kernel_regularizer=l2(0.01))(encoder_1)
encoder_3 = Conv2D(128, (3,3), activation='relu', padding='same', kernel_regularizer=l2(0.01), strides=2)(encoder_2)
encoder_4 = Conv2D(256, (3,3), activation='relu', padding='same', kernel_regularizer=l2(0.01))(encoder_3)
encoder_5 = Conv2D(256, (3,3), activation='relu', padding='same', strides=2, kernel_regularizer=l2(0.01))(encoder_4)
encoder_6 = Conv2D(512, (3,3), activation='relu', padding='same', kernel_regularizer=l2(0.01))(encoder_5)
encoder_7 = Conv2D(512, (3,3), activation='relu', padding='same', kernel_regularizer=l2(0.01))(encoder_6)
encoder_output= Conv2D(256, (3,3), activation='relu', padding='same')(encoder_7)
#Fusion layer
fusion_output = concatenate([encoder_output, embed_input], axis=3)
fusion_output = Conv2D(256, (1, 1), activation='relu', padding='same', kernel_regularizer=l2(0.01))(fusion_output)
fusion_output = Conv2D(256, (1, 1), activation='relu', padding='same', kernel_regularizer=l2(0.01))(fusion_output)
#Decoder layer
decoder_output = Conv2D(128, (3,3), activation='relu', padding='same', kernel_regularizer=l2(0.01))(fusion_output)
decoder_output = UpSampling2D((2, 2))(decoder_output)
decoder_output = Conv2D(64, (3,3), activation='relu', padding='same', kernel_regularizer=l2(0.01))(decoder_output)
decoder_output = UpSampling2D((2, 2))(decoder_output)
decoder_output = Conv2D(32, (3,3), activation='relu', padding='same', kernel_regularizer=l2(0.01))(decoder_output)
decoder_output = Conv2D(16, (3,3), activation='relu', padding='same', kernel_regularizer=l2(0.01))(decoder_output)
decoder_output = Conv2D(2, (3, 3), activation='relu', padding='same', kernel_regularizer=l2(0.01))(decoder_output)
decoder_output = UpSampling2D((2, 2))(decoder_output)
model = Model(inputs=[encoder_input,embed_input], outputs=decoder_output)
model.summary()
```

Figure 5. Code snippet for base architecture.

gives the best closest match to the intended output.

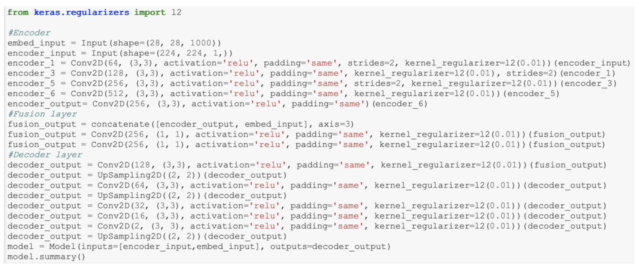
We have implemented the following architectures.

We have implemented the architecture shown in figure



```
________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
================================================================================================
 input_3 (InputLayer)           [(None, 224, 224, 1  0           []
                                )]

 conv2d_203 (Conv2D)            (None, 112, 112, 64  640         ['input_3[0][0]']
                                )

 conv2d_204 (Conv2D)            (None, 56, 56, 128)  73856       ['conv2d_203[0][0]']

 conv2d_205 (Conv2D)            (None, 28, 28, 256)  295168      ['conv2d_204[0][0]']

 conv2d_206 (Conv2D)            (None, 28, 28, 512)  1180160     ['conv2d_205[0][0]']

 conv2d_207 (Conv2D)            (None, 28, 28, 256)  1179904     ['conv2d_206[0][0]']

 input_2 (InputLayer)           [(None, 28, 28, 100  0           []
                                0)]

 concatenate (Concatenate)      (None, 28, 28, 1256  0           ['conv2d_207[0][0]',
                                )                                 'input_2[0][0]']

 conv2d_208 (Conv2D)            (None, 28, 28, 256)  321792      ['concatenate[0][0]']

 conv2d_209 (Conv2D)            (None, 28, 28, 256)  65792       ['conv2d_208[0][0]']

 conv2d_210 (Conv2D)            (None, 28, 28, 128)  295040      ['conv2d_209[0][0]']

 up_sampling2d (UpSampling2D)   (None, 56, 56, 128)  0           ['conv2d_210[0][0]']

 conv2d_211 (Conv2D)            (None, 56, 56, 64)   73792       ['up_sampling2d[0][0]']

 up_sampling2d_1 (UpSampling2D)  (None, 112, 112, 64  0          ['conv2d_211[0][0]']
                                )

 conv2d_212 (Conv2D)            (None, 112, 112, 32  18464       ['up_sampling2d_1[0][0]']
                                )

 conv2d_213 (Conv2D)            (None, 112, 112, 16  4624        ['conv2d_212[0][0]']
                                )

 conv2d_214 (Conv2D)            (None, 112, 112, 2)  290         ['conv2d_213[0][0]']

 up_sampling2d_2 (UpSampling2D)  (None, 224, 224, 2)  0          ['conv2d_214[0][0]']

================================================================================================
Total params: 3,509,522
Trainable params: 3,509,522
Non-trainable params: 0
________________________________________________________________________________________________
```

Figure 6. Architecture 1.

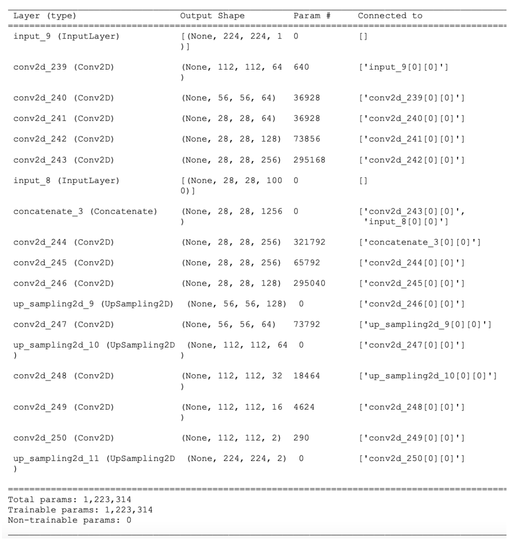


```
from keras.regularizers import l2

#Encoder
embed_input = Input(shape=(28, 28, 1000))
encoder_input = Input(shape=(224, 224, 1,))
encoder_1 = Conv2D(64, (3,3), activation='relu', padding='same', strides=2, kernel_regularizer=l2(0.01))(encoder_input)
encoder_3 = Conv2D(128, (3,3), activation='relu', padding='same', kernel_regularizer=l2(0.01), strides=2)(encoder_1)
encoder_5 = Conv2D(256, (3,3), activation='relu', padding='same', strides=2, kernel_regularizer=l2(0.01))(encoder_3)
encoder_6 = Conv2D(512, (3,3), activation='relu', padding='same', kernel_regularizer=l2(0.01))(encoder_5)
encoder_output= Conv2D(256, (3,3), activation='relu', padding='same')(encoder_6)
#Fusion layer
fusion_output = concatenate([encoder_output, embed_input], axis=3)
fusion_output = Conv2D(256, (1, 1), activation='relu', padding='same', kernel_regularizer=l2(0.01))(fusion_output)
fusion_output = Conv2D(256, (1, 1), activation='relu', padding='same', kernel_regularizer=l2(0.01))(fusion_output)
#Decoder layer
decoder_output = Conv2D(128, (3,3), activation='relu', padding='same', kernel_regularizer=l2(0.01))(fusion_output)
decoder_output = UpSampling2D((2, 2))(decoder_output)
decoder_output = Conv2D(64, (3,3), activation='relu', padding='same', kernel_regularizer=l2(0.01))(decoder_output)
decoder_output = UpSampling2D((2, 2))(decoder_output)
decoder_output = Conv2D(32, (3,3), activation='relu', padding='same', kernel_regularizer=l2(0.01))(decoder_output)
decoder_output = Conv2D(16, (3,3), activation='relu', padding='same', kernel_regularizer=l2(0.01))(decoder_output)
decoder_output = Conv2D(2, (3, 3), activation='relu', padding='same', kernel_regularizer=l2(0.01))(decoder_output)
decoder_output = UpSampling2D((2, 2))(decoder_output)
model = Model(inputs=[encoder_input,embed_input], outputs=decoder_output)
model.summary()
```

Figure 7. Code snippet for architecture 1.

4 and 5 as the base architecture and have performed major experiments on it. The number and types of layers used are given in figure 5. Architecture 1, 2 and 3 are the modifications which were made to the base architecture.

The number of layers and filters have been varied in order to experiment with the proposed approach. Figure 6 depicts the Architecture 1 and figure 7 shows the code snippet for this architecture, similarly figure 8 shows the Architecture 2 and figure 9 shows the code snippet for this architecture 2, and figure 10 shows the Architecture 3 and figure 11 shows the code snippet for this architecture 3.



```
____________________________________________________________________________________
 Layer (type)                Output Shape         Param #     Connected to
====================================================================================
 input_9 (InputLayer)        [(None, 224, 224, 1  0           []
                             )]

 conv2d_239 (Conv2D)         (None, 112, 112, 64  640         ['input_9[0][0]']
                             )

 conv2d_240 (Conv2D)         (None, 56, 56, 64)   36928       ['conv2d_239[0][0]']

 conv2d_241 (Conv2D)         (None, 28, 28, 64)   36928       ['conv2d_240[0][0]']

 conv2d_242 (Conv2D)         (None, 28, 28, 128)  73856       ['conv2d_241[0][0]']

 conv2d_243 (Conv2D)         (None, 28, 28, 256)  295168      ['conv2d_242[0][0]']

 input_8 (InputLayer)        [(None, 28, 28, 100  0           []
                             0)]

 concatenate_3 (Concatenate)  (None, 28, 28, 1256  0          ['conv2d_243[0][0]',
                             )                                 'input_8[0][0]']

 conv2d_244 (Conv2D)         (None, 28, 28, 256)  321792      ['concatenate_3[0][0]']

 conv2d_245 (Conv2D)         (None, 28, 28, 256)  65792       ['conv2d_244[0][0]']

 conv2d_246 (Conv2D)         (None, 28, 28, 128)  295040      ['conv2d_245[0][0]']

 up_sampling2d_9 (UpSampling2D)  (None, 56, 56, 128)  0       ['conv2d_246[0][0]']

 conv2d_247 (Conv2D)         (None, 56, 56, 64)   73792       ['up_sampling2d_9[0][0]']

 up_sampling2d_10 (UpSampling2D  (None, 112, 112, 64  0       ['conv2d_247[0][0]']
 )                           )

 conv2d_248 (Conv2D)         (None, 112, 112, 32  18464       ['up_sampling2d_10[0][0]']
                             )

 conv2d_249 (Conv2D)         (None, 112, 112, 16  4624        ['conv2d_248[0][0]']
                             )

 conv2d_250 (Conv2D)         (None, 112, 112, 2)  290         ['conv2d_249[0][0]']

 up_sampling2d_11 (UpSampling2D  (None, 224, 224, 2)  0       ['conv2d_250[0][0]']
 )

====================================================================================
Total params: 1,223,314
Trainable params: 1,223,314
Non-trainable params: 0
____________________________________________________________________________________
```

Figure 8. Architecture 2.

3.2. Workflow

The workflow of the project can be explained in the following steps:

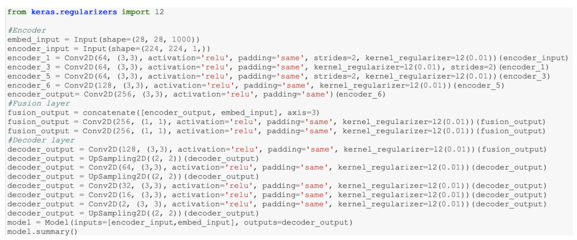
- Mapping of L component to obtain a and b image components.
- Combining L, a and b components to obtain the colored image.
- Splitting the dataset into training, validation and test sets.
- Loading pre-trained Inception-Resnet v2's weights.
- Generating input embeddings.
- Running the proposed model.
- Parameter optimization.
- Evaluation and results formulation.

4. Experiments

This section elaborates on the experiments performed and the components involved by following the workflow provided in the previous section.

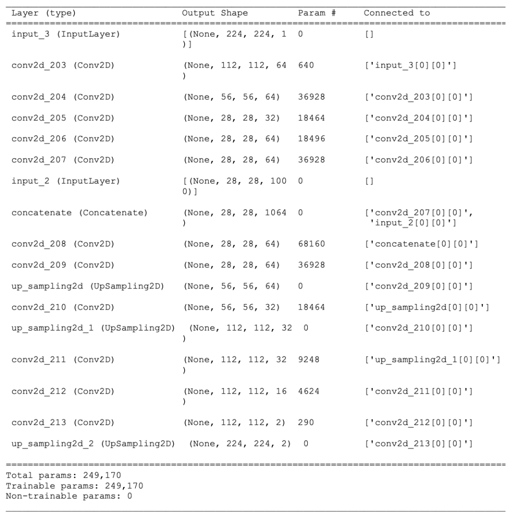
4.1. Dataset

We have implemented a part of this project using the "Image Colorization" dataset [1], which is a gray scale conversion of the colored images dataset. It is a publically



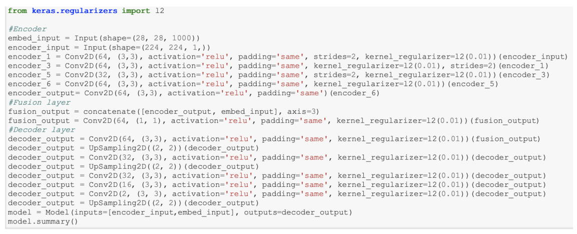
```python
from keras.regularizers import l2

#Encoder
embed_input = Input(shape=(28, 28, 1000))
encoder_input = Input(shape=(224, 224, 1,))
encoder_1 = Conv2D(64, (3,3), activation='relu', padding='same', strides=2, kernel_regularizer=l2(0.01))(encoder_input)
encoder_3 = Conv2D(64, (3,3), activation='relu', padding='same', kernel_regularizer=l2(0.01), strides=2)(encoder_1)
encoder_5 = Conv2D(64, (3,3), activation='relu', padding='same', strides=2, kernel_regularizer=l2(0.01))(encoder_3)
encoder_6 = Conv2D(128, (3,3), activation='relu', padding='same', kernel_regularizer=l2(0.01))(encoder_5)
encoder_output= Conv2D(256, (3,3), activation='relu', padding='same')(encoder_6)
#Fusion layer
fusion_output = concatenate([encoder_output, embed_input], axis=3)
fusion_output = Conv2D(256, (1, 1), activation='relu', padding='same', kernel_regularizer=l2(0.01))(fusion_output)
fusion_output = Conv2D(256, (1, 1), activation='relu', padding='same', kernel_regularizer=l2(0.01))(fusion_output)
#Decoder layer
decoder_output = Conv2D(128, (3,3), activation='relu', padding='same', kernel_regularizer=l2(0.01))(fusion_output)
decoder_output = UpSampling2D((2, 2))(decoder_output)
decoder_output = Conv2D(64, (3,3), activation='relu', padding='same', kernel_regularizer=l2(0.01))(decoder_output)
decoder_output = UpSampling2D((2, 2))(decoder_output)
decoder_output = Conv2D(32, (3,3), activation='relu', padding='same', kernel_regularizer=l2(0.01))(decoder_output)
decoder_output = Conv2D(16, (3,3), activation='relu', padding='same', kernel_regularizer=l2(0.01))(decoder_output)
decoder_output = Conv2D(2, (3, 3), activation='relu', padding='same', kernel_regularizer=l2(0.01))(decoder_output)
decoder_output = UpSampling2D((2, 2))(decoder_output)
model = Model(inputs=[encoder_input,embed_input], outputs=decoder_output)
model.summary()
```

Figure 9. Code snippet for architecture 2.



```
____________________________________________________________________________________
 Layer (type)                Output Shape         Param #     Connected to
====================================================================================
 input_3 (InputLayer)        [(None, 224, 224, 1  0           []
                             )]

 conv2d_203 (Conv2D)         (None, 112, 112, 64  640         ['input_3[0][0]']
                             )

 conv2d_204 (Conv2D)         (None, 56, 56, 64)   36928       ['conv2d_203[0][0]']

 conv2d_205 (Conv2D)         (None, 28, 28, 32)   18464       ['conv2d_204[0][0]']

 conv2d_206 (Conv2D)         (None, 28, 28, 64)   18496       ['conv2d_205[0][0]']

 conv2d_207 (Conv2D)         (None, 28, 28, 64)   36928       ['conv2d_206[0][0]']

 input_2 (InputLayer)        [(None, 28, 28, 100  0           []
                             0)]

 concatenate (Concatenate)   (None, 28, 28, 1064  0           ['conv2d_207[0][0]',
                             )                                 'input_2[0][0]']

 conv2d_208 (Conv2D)         (None, 28, 28, 64)   68160       ['concatenate[0][0]']

 conv2d_209 (Conv2D)         (None, 28, 28, 64)   36928       ['conv2d_208[0][0]']

 up_sampling2d (UpSampling2D)  (None, 56, 56, 64)  0          ['conv2d_209[0][0]']

 conv2d_210 (Conv2D)         (None, 56, 56, 32)   18464       ['up_sampling2d[0][0]']

 up_sampling2d_1 (UpSampling2D)  (None, 112, 112, 32  0       ['conv2d_210[0][0]']
                             )

 conv2d_211 (Conv2D)         (None, 112, 112, 32  9248        ['up_sampling2d_1[0][0]']
                             )

 conv2d_212 (Conv2D)         (None, 112, 112, 16  4624        ['conv2d_211[0][0]']
                             )

 conv2d_213 (Conv2D)         (None, 112, 112, 2)  290         ['conv2d_212[0][0]']

 up_sampling2d_2 (UpSampling2D)  (None, 224, 224, 2)  0       ['conv2d_213[0][0]']

====================================================================================
Total params: 249,170
Trainable params: 249,170
Non-trainable params: 0
____________________________________________________________________________________
```

Figure 10. Architecture 3.



```python
from keras.regularizers import l2

#Encoder
embed_input = Input(shape=(28, 28, 1000))
encoder_input = Input(shape=(224, 224, 1,))
encoder_1 = Conv2D(64, (3,3), activation='relu', padding='same', strides=2, kernel_regularizer=l2(0.01))(encoder_input)
encoder_3 = Conv2D(64, (3,3), activation='relu', padding='same', kernel_regularizer=l2(0.01), strides=2)(encoder_1)
encoder_5 = Conv2D(32, (3,3), activation='relu', padding='same', strides=2, kernel_regularizer=l2(0.01))(encoder_3)
encoder_6 = Conv2D(64, (3,3), activation='relu', padding='same', kernel_regularizer=l2(0.01))(encoder_5)
encoder_output= Conv2D(64, (3,3), activation='relu', padding='same')(encoder_6)
#Fusion layer
fusion_output = concatenate([encoder_output, embed_input], axis=3)
fusion_output = Conv2D(64, (1, 1), activation='relu', padding='same', kernel_regularizer=l2(0.01))(fusion_output)
#Decoder layer
decoder_output = Conv2D(64, (3,3), activation='relu', padding='same', kernel_regularizer=l2(0.01))(fusion_output)
decoder_output = UpSampling2D((2, 2))(decoder_output)
decoder_output = Conv2D(32, (3,3), activation='relu', padding='same', kernel_regularizer=l2(0.01))(decoder_output)
decoder_output = UpSampling2D((2, 2))(decoder_output)
decoder_output = Conv2D(32, (3,3), activation='relu', padding='same', kernel_regularizer=l2(0.01))(decoder_output)
decoder_output = Conv2D(16, (3,3), activation='relu', padding='same', kernel_regularizer=l2(0.01))(decoder_output)
decoder_output = Conv2D(2, (3, 3), activation='relu', padding='same', kernel_regularizer=l2(0.01))(decoder_output)
decoder_output = UpSampling2D((2, 2))(decoder_output)
model = Model(inputs=[encoder_input,embed_input], outputs=decoder_output)
model.summary()
```

Figure 11. Code snippet for architecture 3.

available dataset, and consists of 2.9 GB of data consisting of 25K sets of images. We have randomly sampled our data to form (20 percent) test, (60 percent) train and (20 percent) validation datasets, and have used Google Colab for our computation of code. The dataset contains L component (Luminance values), a and b components in NumPy files. The dataset has been visualized in the figure below:

4.2. Training

We have conducted several experiments by varying parameters like number and types of layers, regularization parameter, learning rate, etc,. The model has been trained on 2500 epochs, using Adam optimizer and Mean Square Error loss. Parameter optimization has been performed on the validation set to obtain the best possible results and therefore, various models have been run and analyzed. The variations mainly involve varying the types and number of layers in the network, varying the learning rate from 0.1 to 0.0001, varying the batch size and varying the regularization constant for l2 regularization from 0.1 to 0.001. The project has been implemented in Keras and TensorFlow in Python 3.7 using Google Colab's GPU.

5. Results and Discussions

After training and parameter optimization, we fed the test images to the model and it performed fairly okay, with highest generalization accuracy of 56.87 %. Plots were constructed using matplotlib so as to cogently display the results obtained. Some of the plots which gave striking results are shown below:

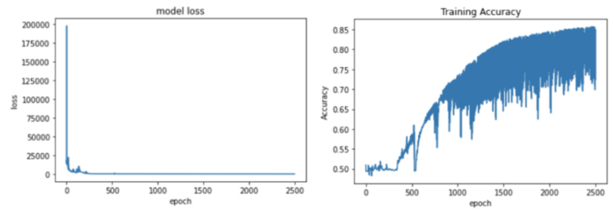


Figure 12. Training loss and training accuracy on architecture 1 with learning rate 0.01, l2 regularization 0.01, batch size=5

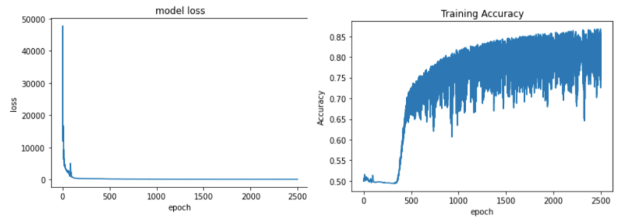


Figure 13. Training loss and training accuracy with learning rate 0.01 on architecture 2 with l2 regularization 0.01, batch size=5

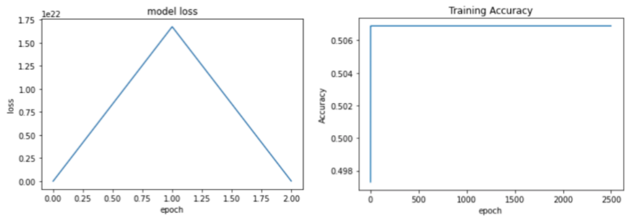


Figure 14. Training loss and training accuracy on base model with batch size = 50, l2 regularization 0.01, learning rate = 0.01

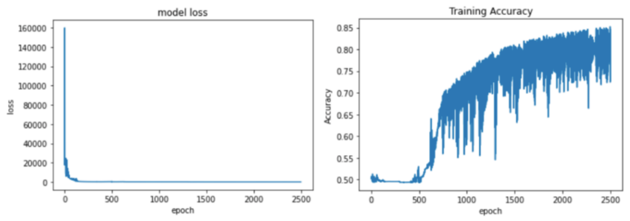


Figure 15. Training loss and training accuracy on base model with learning rate 0.001 on base model, l2 regularization 0.01, batch size=5

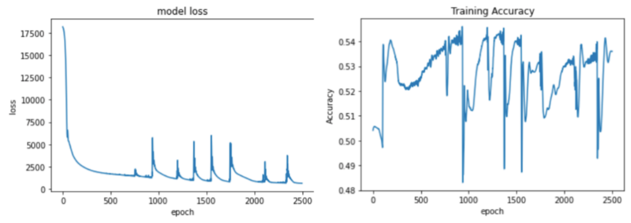


Figure 16. Training loss and training accuracy on base model with learning rate = 0.00001

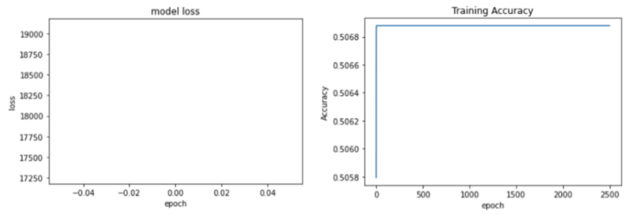


Figure 17. Training loss and accuracy on base model with learning rate = 0.1, batch size = 5 regularization =0.01

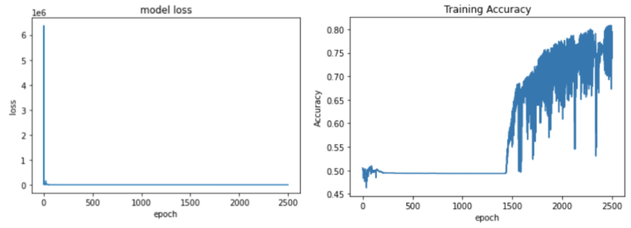


Figure 18. Training loss and training accuracy on base model with learning rate = 0.001, batch size = 5 and l2 regularization = 0.1

In figure 12, the learning rate is 0.001 the learning here is optimum as can be seen in the loss graph as well as training accuracy which increases slowly. The test accuracy achieved here was 0.5496931 while the validation accuracy was 0.567035. In figure 13, the learning rate is 0.01 while the architecture 2 is used with validation accuracy of 0.5714356 and test accuracy of 0.5394033. In figure 14, the batch size is increased from 5 to 50 so no learning hap-

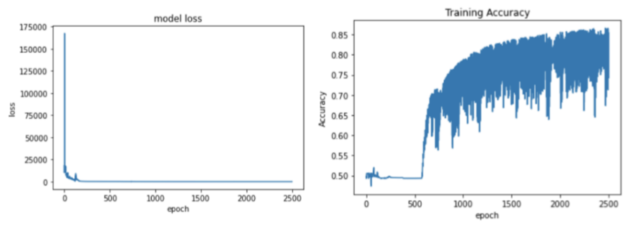


Figure 19. Training loss and training accuracy on base model with learning rate = 0.01, batch size = 5 and l2 regularization = 0.01

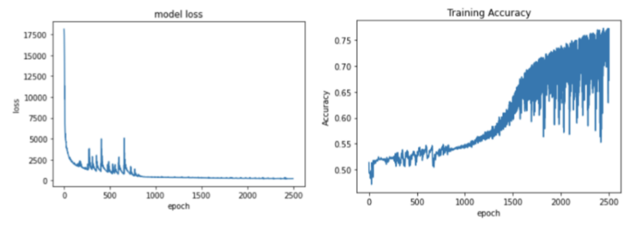


Figure 20. Training loss and training accuracy on base model with learning rate = 0.0001, batch size = 5 and l2 regularization = 0.01

pens as we increased the batch size drastically. In figure 15, the learning rate is 0.001 as the learning rate is less the graph is slow to go up but achieves a validation accuracy of 0.5881328 and test accuracy of 0.55679. In figure 16, the learning rate is 0.00001 and hence the learning is very slow and achieves the validation accuracy 0.52240014 and test accuracy accuracy of 0.5577577. In figure 17, the learning rate is 0.1 hence the loss is nan due to high learning rate and validation accuracy remains at 0.49233896 and test accuracy of 0.47044206. In figure 18, the learning rate is 0.001 with regularization as 0.1 with validation accuracy of 0.57672495 and test accuracy of 0.54632795. In figure 19, the base model has been used with learning rate of 0.01, the l2 regularization was 0.01, and batch size was 5 where the validation accuracy attained is 0.584938 and the test accuracy attained is 0.55964506. In Figure 20, the base model has been used with learning rate of 0.0001, batch size of 5 and l2 regularization of 0.01 achieves a 0.57347536 validation accuracy and 0.568752 test accuracy.

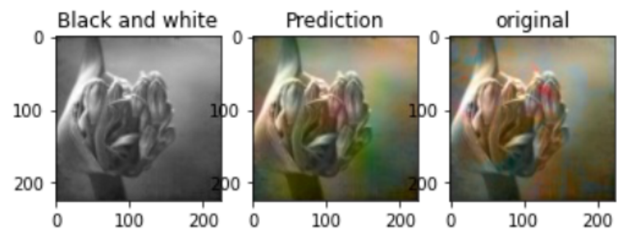


Figure 21. Predicted images on base model with lr = 0.0001 and regularization parameter = 0.1. where lr = learning rate

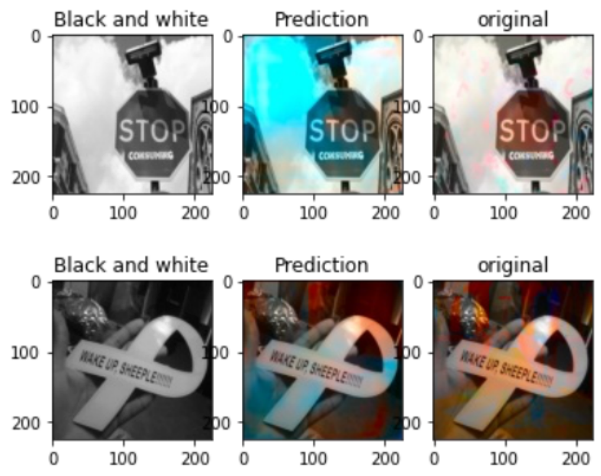


Figure 22. Predicted images on base model lr = 0.0001

Figures 21-26 contains the results generated by the base model. Figures 22, 23, 24 ans 26 shows the output produced by the base model on batch size = 5 and learning rates = 0.00001, 0.001, 0.1 and 0.01. Figure 21 shows results of the base model with l2 regularization = 0.1, batch size = 5 and learning rate = 0.0001, and figure 25 shows the results of the base model with l2 regularization = 0.1, batch size = 5 and learning rate = 0.01.

It can be seen that the best results are shown in figure 23. That is, the base model with batch size = 5 and learning rate = 0.001, with l2 regularization = 0.01. The results given in figure 25 are a little plausible but do not match the original image completely, and they were produced by the base model with learning rate = 0.01 and regularization parameter of 0.1. The results depicted by figure 21 are by the base model with learning rate = 0.00001 and since the learning rate is so low, the model is not able to learn and thus produce poor results. Figure 24 shows the results obtained by the base model with learning rate = 0.1. As the learning rate is too high for this particular task it causes the model to converge too quickly to a sub-optimal solution.

6. Conclusion and Future Works

This project is an attempt to implement automatic colorization of gray scale images, which still remains a difficult task due to its multi modal nature. Our approach is successfully able to colorize high level components like fores and sky, but struggles a little with lower level components. Better resources for computation and a larger dataset should be able to deal with this issue.

From our experiments, we found that the best performance was achieved using the base model with learning rate of 0.0001, using Adam optimizer, batch size of 5 and regularization of 0.01. We conclude that we have been able to

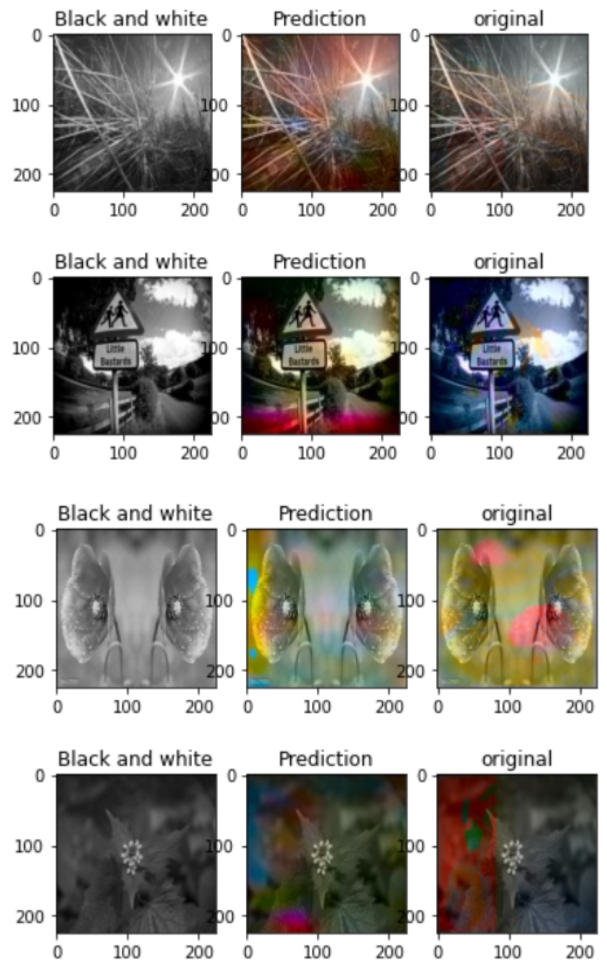


Figure 23. Predicted images on base model for lr = 0.001

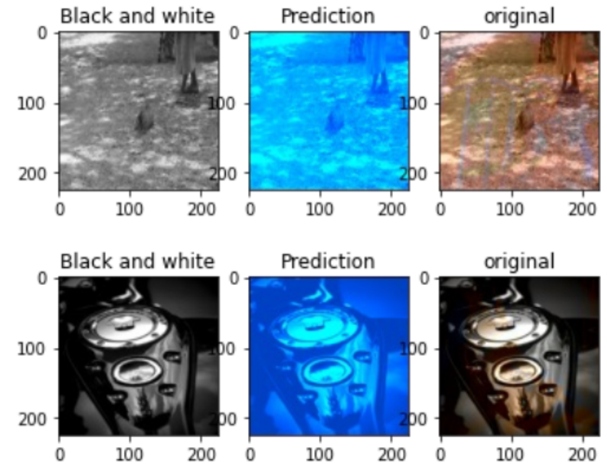


Figure 24. Predicted images on base model learning rate = 0.1

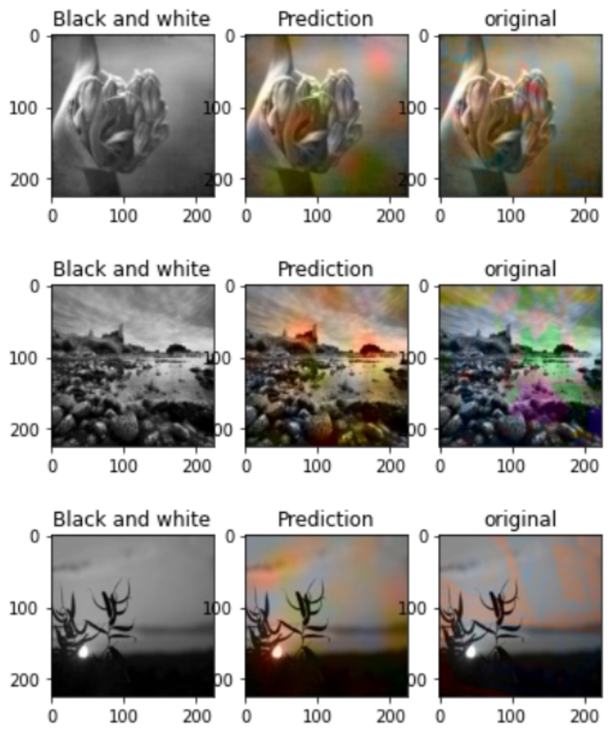


Figure 25. Predicted images on base model with lr = 0.01, regularization parameter = 0.1

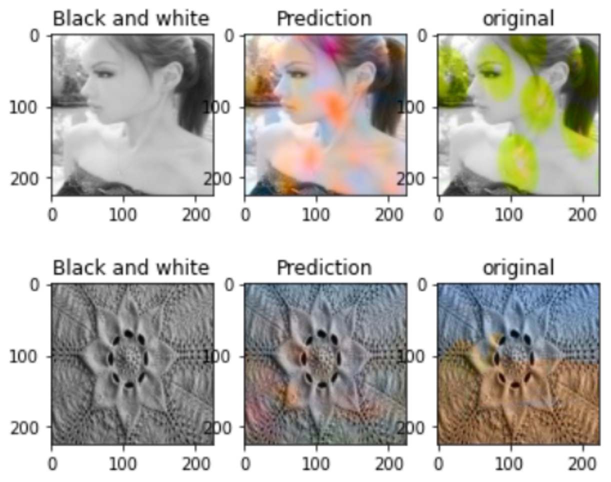


Figure 26. Predicted images on base model with lr = 0.01

achieve an accuracy of approximately 56.8752 percent.

Various other networks could also be used, for instance the use of variational autoencoders for image generation and then colorizing those images could also help improve performance.

Image generation remains a tough task as multiple values/ colors exist for a single datapoint. We believe that some form of human intervention is required to achieve near-perfect colorized images, this task remains an important task and future works could help reduce the hours of human supervision required.

References

[1] Image colorization dataset. https://www.kaggle.com/shravankumar9892/image-colorization. Accessed: 2021-12-01. 4

[2] Alex Alemi et al. Improving inception and image classification in tensorflow. *Google Research Blog*, 2016. 1

[3] Jiancheng An, Koffi Gagnon Kpeyiton, and Qingnan Shi. Grayscale images colorization with convolutional neural networks. *Soft Computing*, 24(7):4751–4758, 2020. 2

[4] Federico Baldassarre, Diego González Morín, and Lucas Rodés-Guirao. Deep koalarization: Image colorization using cnns and inception-resnet-v2. *arXiv preprint arXiv:1712.03400*, 2017. 1

[5] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *Communications of the ACM*, 63(11):139–144, 2020. 1

[6] Ustin Sousa, Rasoul Kabirzadeh, and Patrick Blaes. Automatic colorization of grayscale images. *Department of Electrical Engineering, Stanford University*, 2013. 1

[7] Richard Zhang, Phillip Isola, and Alexei A Efros. Colorful image colorization. In *European conference on computer vision*, pages 649–666. Springer, 2016. 1