UNSUPERVISED MACHINE LEARNING - SEMESTER II

Chromatic Dreams: Image Colorization with Generative Adversarial Networks

Upasana Udayaraj Yadav

SVKM'S NMIMS Nilkamal School Of Mathematics, Applied Statistics And Analytics , Vile Parle West, Mumbai- 400 056

yadav.upasana0092@gmail.com

PROBLEM STATEMENT - *Traditional methods often struggle to produce realistic and vibrant colors, especially in complex scenes or with subtle details. In this context, leveraging Generative Adversarial Networks (GANs) presents a promising avenue for enhancing the quality and fidelity of colorization results. However, existing GAN-based approaches encounter issues such as color consistency, artifacting, and computational inefficiency. Thus, there is a pressing need to develop novel techniques that effectively address these challenges and push the boundaries of image colorization with GANs, ultimately enabling the creation of lifelike and visually appealing colorized images across various domains and applications.*

DATASET : https://drive.google.com/drive/folders/12hEx4LSPaTEw_96I8nlncJSPY5u1FXn4?usp=sharing

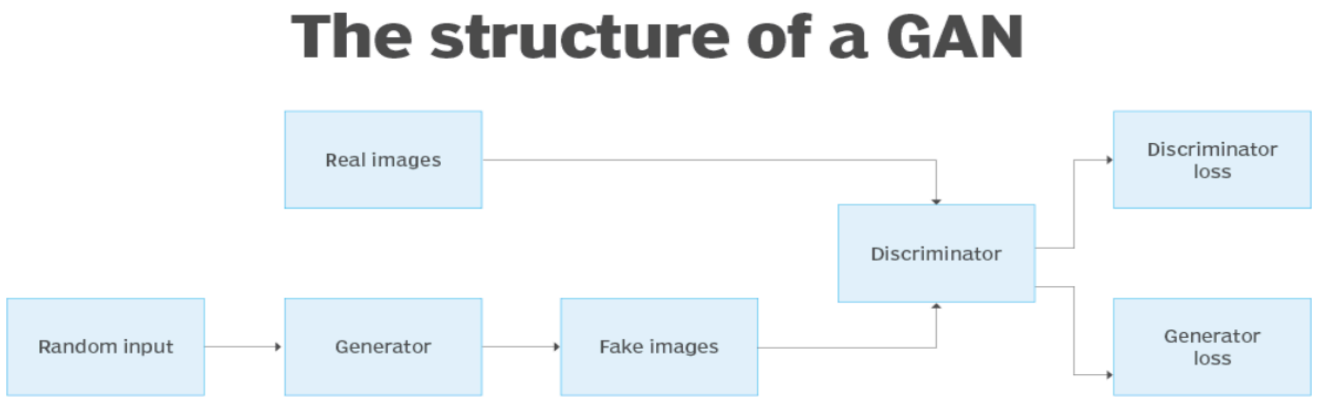
MOTIVATION -

1. Recolorizing and restoring old photos is a painstaking process when done manually through some photo editing software.
2. One solution to this problem is using GANs .
3. Colorizing black and white images with deep learning has become an impressive showcase for the real world application of neural networks in our lives.

PYTHONFILE- : https://colab.research.google.com/drive/1c47yo6VuF7CabDvmZBGfYrOY5tE1_XNV?usp=sharing

WHAT ARE GAN'S -

Generative Adversarial Networks (GANs) are a way to make generative models by having two neural networks compete with each other . GANs are the state-of-the-art machine learning models which can generate new data instances from existing ones. They use a very interesting technique, inspired from Game Theory, to generate realistic samples.



Source : https://www.techtarget.com/searchenterpriseai/definition/generative-adversarial-network-GAN

METHODOLOGY -

1. Dataset Description:
The dataset comprises 3000 RGB images sourced from various domains, including landscapes such as mountains, forests, urban scenes, and more. These images serve as the foundation for the training process.

2. Preprocessing:
RGB images are converted into grayscale to serve as the ground truth labels for the model. This conversion simplifies the task, focusing solely on learning the colorization process without the complexity of full-color images.

3. Training Strategy:
The training strategy alternates between updating the generator once and the discriminator twice for each training step.

4. Discriminator Objective:
The discriminator aims to accurately classify generated images as fake or real and assign high probabilities (closer to 1.0) for images originating from the dataset. This objective ensures effective discrimination between real and fake images, providing meaningful feedback to the generator.

5. Generator Objective:
The generator minimizes its loss by producing images that deceive the discriminator. Generating images indistinguishable from real ones "fools" the discriminator into assigning high probabilities to its outputs, even though they are artificially generated.

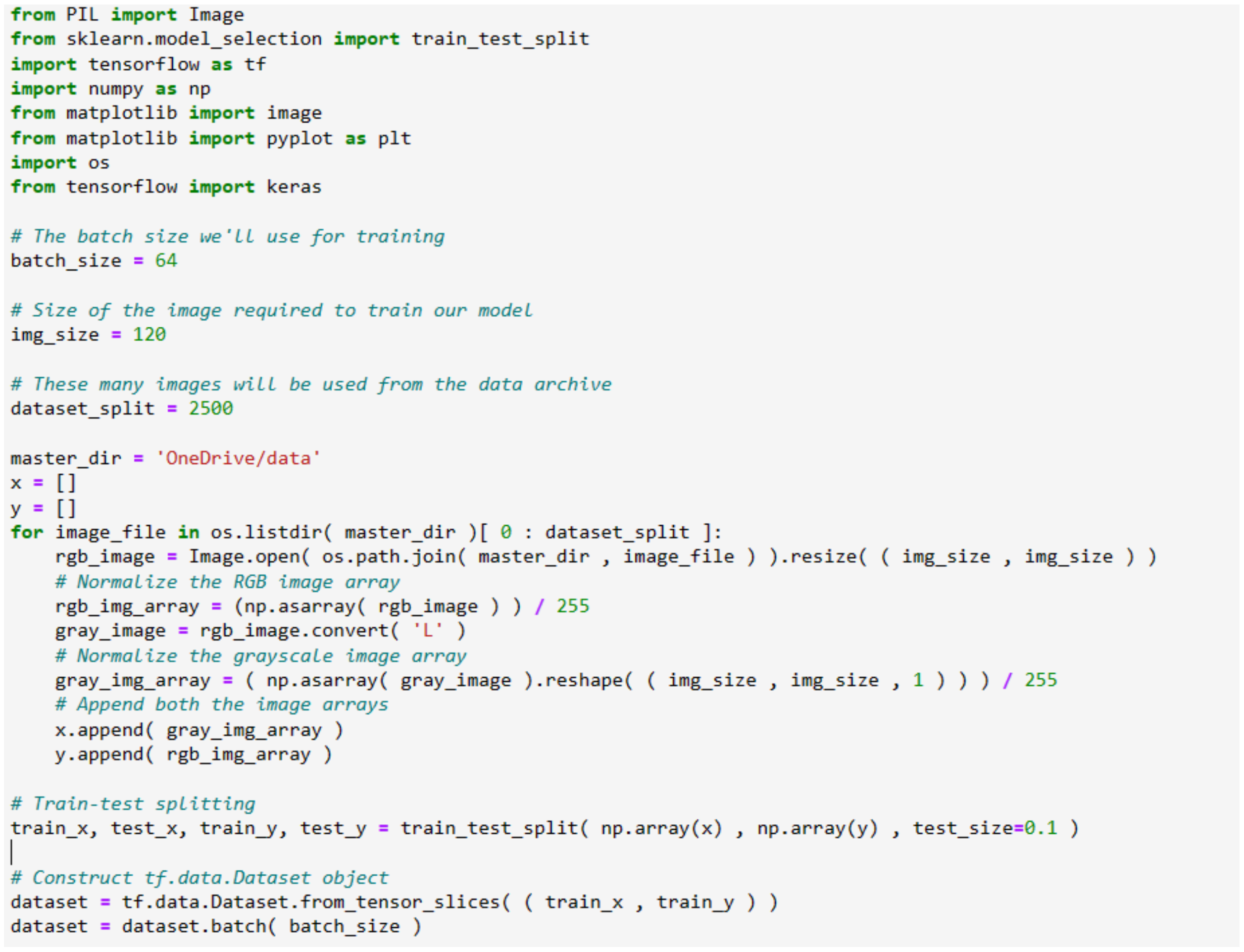
6. Training the Discriminator:
The discriminator is iteratively trained to discern real images from fake ones, outputting probabilities closer to 1.0 for real images and closer to 0.0 for images generated by the generator. This adversarial training process enhances the discriminator's ability to distinguish between real and fake images.

7. Adversarial Training:
Through adversarial interplay between the generator and the discriminator, the goal is to elevate the quality of generated images progressively. As the generator improves, it challenges the discriminator to become more discerning, leading to continuous improvement in both components.

8. Optimization Goal:
The optimization goal is to train a generator capable of producing high-quality colorized images resembling real-world scenes. By refining the generator's ability to deceive the discriminator, the aim is to achieve visually compelling colorization results across diverse domains.

STEPS -

1. Parse the images (RGB images to be precise) one by one, and transform each one to a grayscale image using PIL's `.convert( 'L' )` method. So our dataset will have samples of $(\ grayscale \ image \ , \ RGB \ image \ ) .

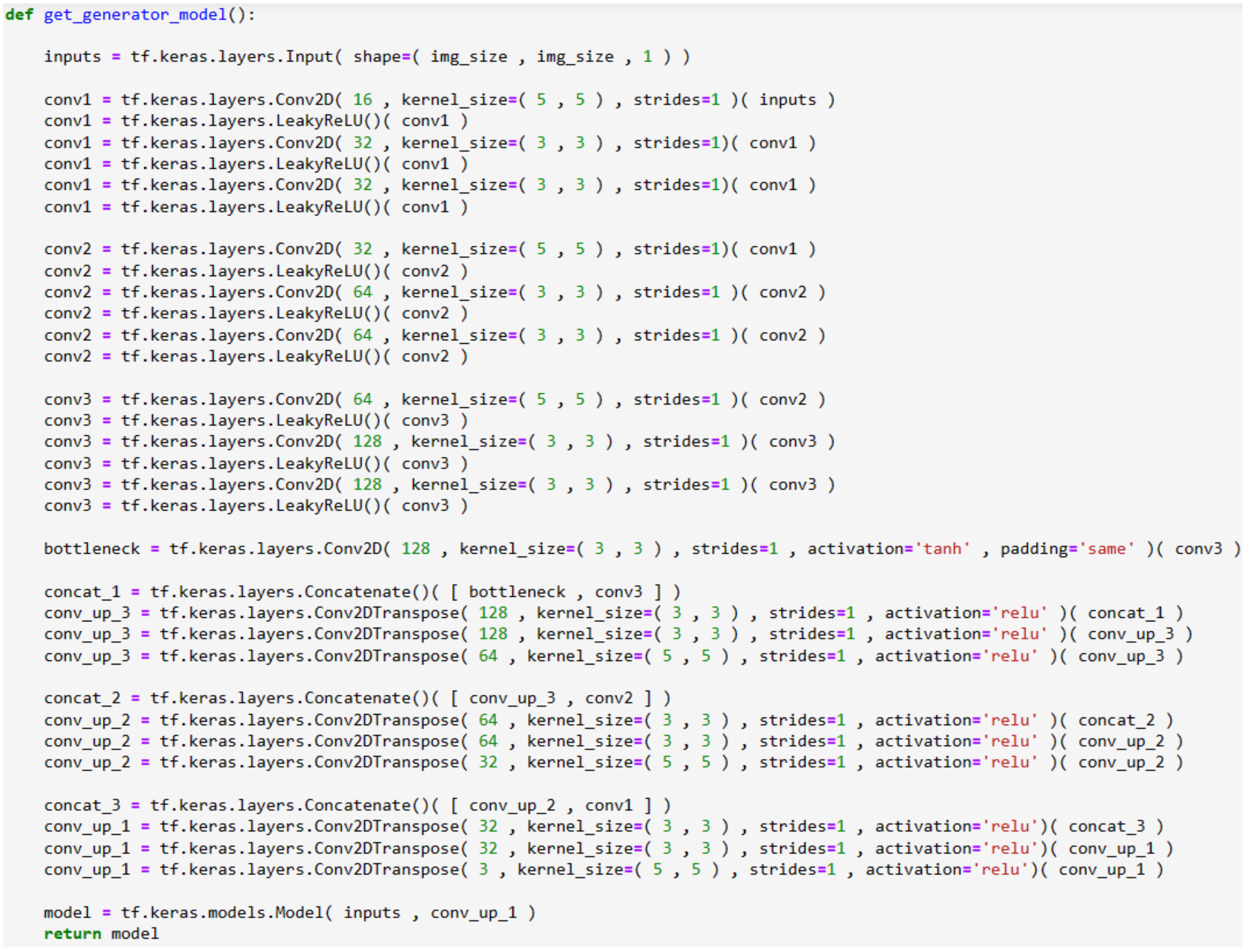


```python
from PIL import Image
from sklearn.model_selection import train_test_split
import tensorflow as tf
import numpy as np
from matplotlib import image
from matplotlib import pyplot as plt
import os
from tensorflow import keras

# The batch size we'll use for training
batch_size = 64

# Size of the image required to train our model
img_size = 120

# These many images will be used from the data archive
dataset_split = 2500

master_dir = 'OneDrive/data'
x = []
y = []
for image_file in os.listdir( master_dir )[ 0 : dataset_split ]:
    rgb_image = Image.open( os.path.join( master_dir , image_file ) ).resize( ( img_size , img_size ) )
    # Normalize the RGB image array
    rgb_img_array = (np.asarray( rgb_image ) ) / 255
    gray_image = rgb_image.convert( 'L' )
    # Normalize the grayscale image array
    gray_img_array = ( np.asarray( gray_image ).reshape( ( img_size , img_size , 1 ) ) ) / 255
    # Append both the image arrays
    x.append( gray_img_array )
    y.append( rgb_img_array )

# Train-test splitting
train_x, test_x, train_y, test_y = train_test_split( np.array(x) , np.array(y) , test_size=0.1 )

# Construct tf.data.Dataset object
dataset = tf.data.Dataset.from_tensor_slices( ( train_x , train_y ) )
dataset = dataset.batch( batch_size )
```

2. THE GAN

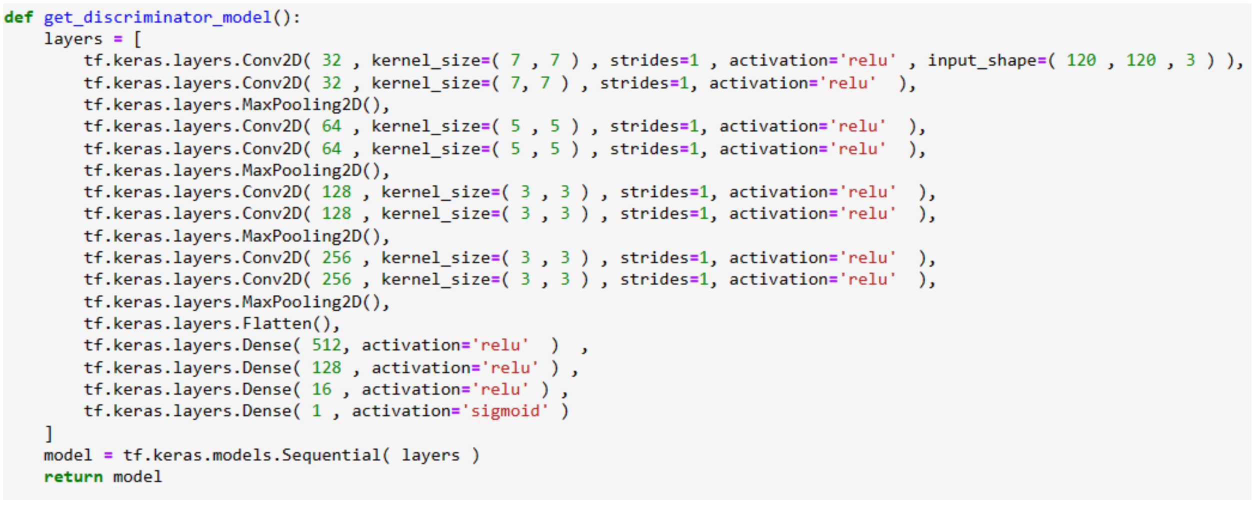
   A) GENERATOR

The generator will have an encoder-decoder structure, similar to the UNet architecture. Additionally, we use Dilated convolutions to have a larger receptive field.I have introduced skip connections in the model so as to have better flow of information from the encoder to the decoder.



```python
def get_generator_model():

    inputs = tf.keras.layers.Input( shape=( img_size , img_size , 1 ) )

    conv1 = tf.keras.layers.Conv2D( 16 , kernel_size=( 5 , 5 ) , strides=1 )( inputs )
    conv1 = tf.keras.layers.LeakyReLU()( conv1 )
    conv1 = tf.keras.layers.Conv2D( 32 , kernel_size=( 3 , 3 ) , strides=1)( conv1 )
    conv1 = tf.keras.layers.LeakyReLU()( conv1 )
    conv1 = tf.keras.layers.Conv2D( 32 , kernel_size=( 3 , 3 ) , strides=1)( conv1 )
    conv1 = tf.keras.layers.LeakyReLU()( conv1 )

    conv2 = tf.keras.layers.Conv2D( 32 , kernel_size=( 5 , 5 ) , strides=1)( conv1 )
    conv2 = tf.keras.layers.LeakyReLU()( conv2 )
    conv2 = tf.keras.layers.Conv2D( 64 , kernel_size=( 3 , 3 ) , strides=1 )( conv2 )
    conv2 = tf.keras.layers.LeakyReLU()( conv2 )
    conv2 = tf.keras.layers.Conv2D( 64 , kernel_size=( 3 , 3 ) , strides=1 )( conv2 )
    conv2 = tf.keras.layers.LeakyReLU()( conv2 )

    conv3 = tf.keras.layers.Conv2D( 64 , kernel_size=( 5 , 5 ) , strides=1 )( conv2 )
    conv3 = tf.keras.layers.LeakyReLU()( conv3 )
    conv3 = tf.keras.layers.Conv2D( 128 , kernel_size=( 3 , 3 ) , strides=1 )( conv3 )
    conv3 = tf.keras.layers.LeakyReLU()( conv3 )
    conv3 = tf.keras.layers.Conv2D( 128 , kernel_size=( 3 , 3 ) , strides=1 )( conv3 )
    conv3 = tf.keras.layers.LeakyReLU()( conv3 )

    bottleneck = tf.keras.layers.Conv2D( 128 , kernel_size=( 3 , 3 ) , strides=1 , activation='tanh' , padding='same' )( conv3 )

    concat_1 = tf.keras.layers.Concatenate()( [ bottleneck , conv3 ] )
    conv_up_3 = tf.keras.layers.Conv2DTranspose( 128 , kernel_size=( 3 , 3 ) , strides=1 , activation='relu' )( concat_1 )
    conv_up_3 = tf.keras.layers.Conv2DTranspose( 128 , kernel_size=( 3 , 3 ) , strides=1 , activation='relu' )( conv_up_3 )
    conv_up_3 = tf.keras.layers.Conv2DTranspose( 64 , kernel_size=( 5 , 5 ) , strides=1 , activation='relu' )( conv_up_3 )

    concat_2 = tf.keras.layers.Concatenate()( [ conv_up_3 , conv2 ] )
    conv_up_2 = tf.keras.layers.Conv2DTranspose( 64 , kernel_size=( 3 , 3 ) , strides=1 , activation='relu' )( concat_2 )
    conv_up_2 = tf.keras.layers.Conv2DTranspose( 64 , kernel_size=( 3 , 3 ) , strides=1 , activation='relu' )( conv_up_2 )
    conv_up_2 = tf.keras.layers.Conv2DTranspose( 32 , kernel_size=( 5 , 5 ) , strides=1 , activation='relu' )( conv_up_2 )

    concat_3 = tf.keras.layers.Concatenate()( [ conv_up_2 , conv1 ] )
    conv_up_1 = tf.keras.layers.Conv2DTranspose( 32 , kernel_size=( 3 , 3 ) , strides=1 , activation='relu')( concat_3 )
    conv_up_1 = tf.keras.layers.Conv2DTranspose( 32 , kernel_size=( 3 , 3 ) , strides=1 , activation='relu')( conv_up_1 )
    conv_up_1 = tf.keras.layers.Conv2DTranspose( 3 , kernel_size=( 5 , 5 ) , strides=1 , activation='relu')( conv_up_1 )

    model = tf.keras.models.Model( inputs , conv_up_1 )
    return model
```

B) DISCRIMINATOR

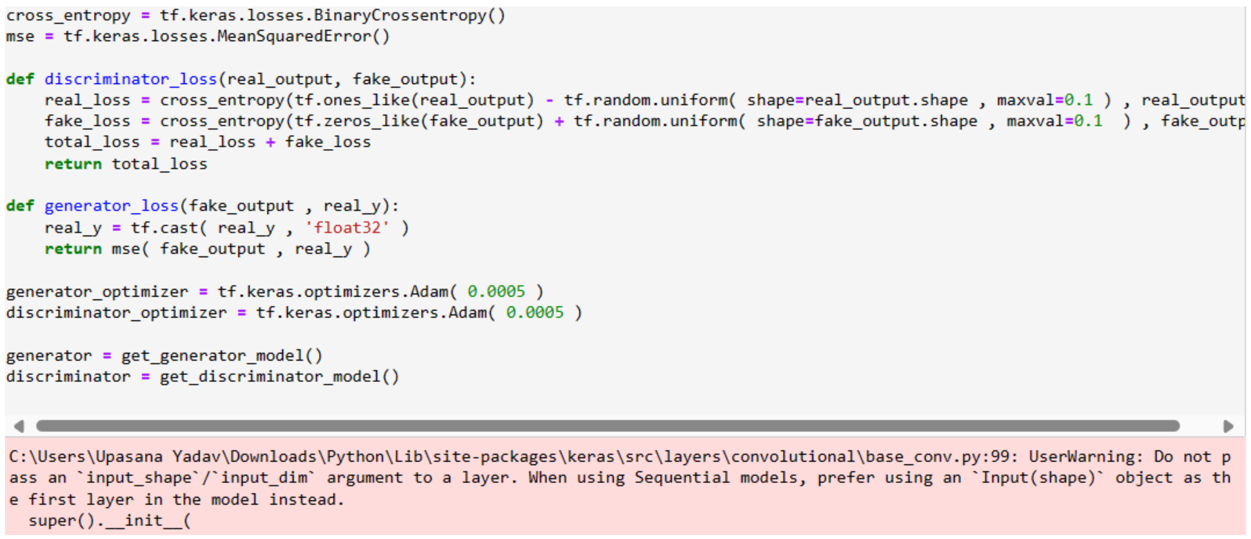


```python
def get_discriminator_model():
    layers = [
        tf.keras.layers.Conv2D( 32 , kernel_size=( 7 , 7 ) , strides=1 , activation='relu' , input_shape=( 120 , 120 , 3 ) ),
        tf.keras.layers.Conv2D( 32 , kernel_size=( 7, 7 ) , strides=1, activation='relu'  ),
        tf.keras.layers.MaxPooling2D(),
        tf.keras.layers.Conv2D( 64 , kernel_size=( 5 , 5 ) , strides=1, activation='relu'  ),
        tf.keras.layers.Conv2D( 64 , kernel_size=( 5 , 5 ) , strides=1, activation='relu'  ),
        tf.keras.layers.MaxPooling2D(),
        tf.keras.layers.Conv2D( 128 , kernel_size=( 3 , 3 ) , strides=1, activation='relu'  ),
        tf.keras.layers.Conv2D( 128 , kernel_size=( 3 , 3 ) , strides=1, activation='relu'  ),
        tf.keras.layers.MaxPooling2D(),
        tf.keras.layers.Conv2D( 256 , kernel_size=( 3 , 3 ) , strides=1, activation='relu'  ),
        tf.keras.layers.Conv2D( 256 , kernel_size=( 3 , 3 ) , strides=1, activation='relu'  ),
        tf.keras.layers.MaxPooling2D(),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense( 512, activation='relu'  )  ,
        tf.keras.layers.Dense( 128 , activation='relu' ) ,
        tf.keras.layers.Dense( 16 , activation='relu' ) ,
        tf.keras.layers.Dense( 1 , activation='sigmoid' )
    ]
    model = tf.keras.models.Sequential( layers )
    return model
```

C) LOSS FUNCTION

For the generator, the L2/MSE loss function is employed to measure the average squared difference between the generated colorized images and the ground truth grayscale images.

Regarding optimization, the Adam optimizer is utilized with a learning rate set at 0.0005. This selection of optimizer allows for efficient convergence by adjusting the learning rates for each parameter individually, while the chosen learning rate of 0.0005 ensures stable and precise updates to the generator's parameters during training.

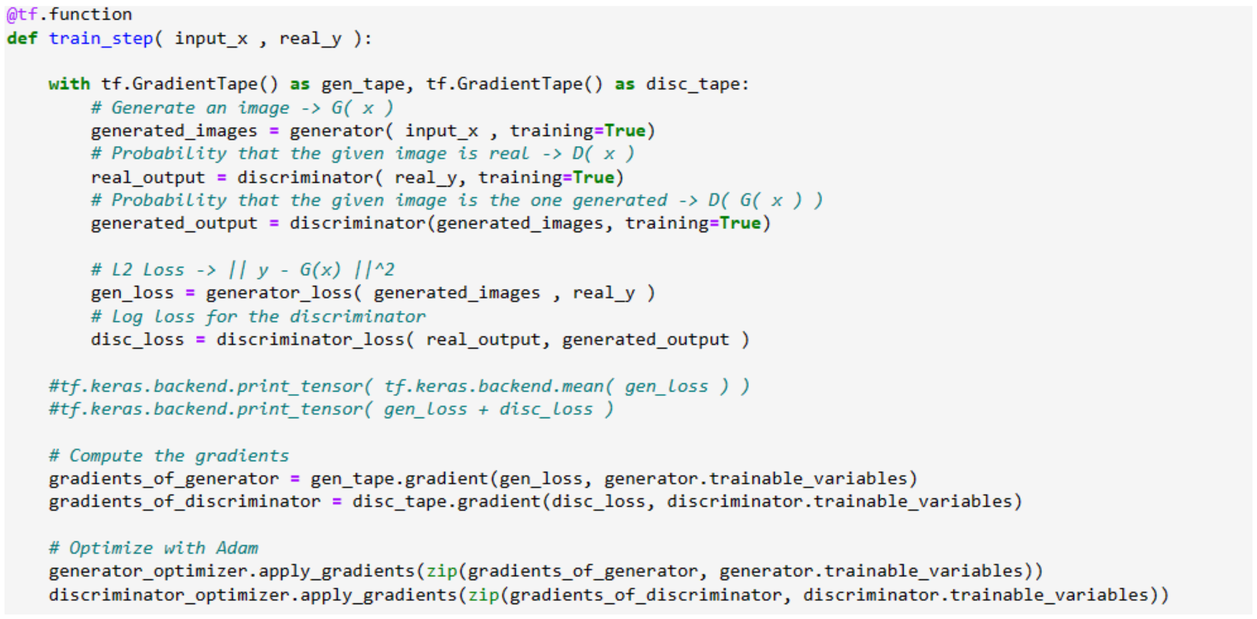


```
cross_entropy = tf.keras.losses.BinaryCrossentropy()
mse = tf.keras.losses.MeanSquaredError()

def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output) - tf.random.uniform( shape=real_output.shape , maxval=0.1 ) , real_output
    fake_loss = cross_entropy(tf.zeros_like(fake_output) + tf.random.uniform( shape=fake_output.shape , maxval=0.1  ) , fake_outp
    total_loss = real_loss + fake_loss
    return total_loss

def generator_loss(fake_output , real_y):
    real_y = tf.cast( real_y , 'float32' )
    return mse( fake_output , real_y )

generator_optimizer = tf.keras.optimizers.Adam( 0.0005 )
discriminator_optimizer = tf.keras.optimizers.Adam( 0.0005 )

generator = get_generator_model()
discriminator = get_discriminator_model()
```

```
C:\Users\Upasana Yadav\Downloads\Python\Lib\site-packages\keras\src\layers\convolutional\base_conv.py:99: UserWarning: Do not p
ass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as th
e first layer in the model instead.
  super().__init__(
```

3. TRAINING THE GAN

- Used Soft and Noisy Labels
- Used the ADAM Optimizer
- Train discriminator
- Avoid Sparse Gradients ReLU



```
@tf.function
def train_step( input_x , real_y ):

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        # Generate an image -> G( x )
        generated_images = generator( input_x , training=True)
        # Probability that the given image is real -> D( x )
        real_output = discriminator( real_y, training=True)
        # Probability that the given image is the one generated -> D( G( x ) )
        generated_output = discriminator(generated_images, training=True)

        # L2 Loss -> || y - G(x) ||^2
        gen_loss = generator_loss( generated_images , real_y )
        # Log loss for the discriminator
        disc_loss = discriminator_loss( real_output, generated_output )

    #tf.keras.backend.print_tensor( tf.keras.backend.mean( gen_loss ) )
    #tf.keras.backend.print_tensor( gen_loss + disc_loss )

    # Compute the gradients
    gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_variables)

    # Optimize with Adam
    generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discriminator.trainable_variables))
```

```
num_epochs = 150

for e in range( num_epochs ):
    print( e )
    for ( x , y ) in dataset:
        # Here ( x , y ) represents a batch from our training dataset.
```

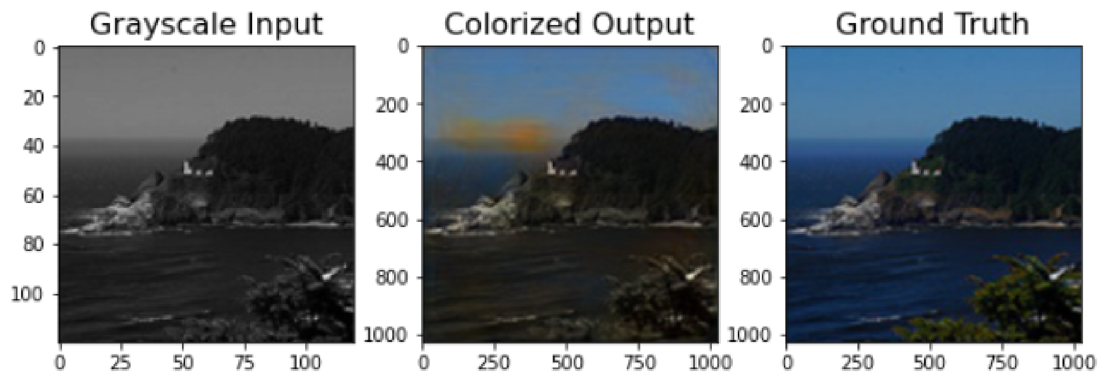
ARCHITECTURE OF GENERATOR MODEL

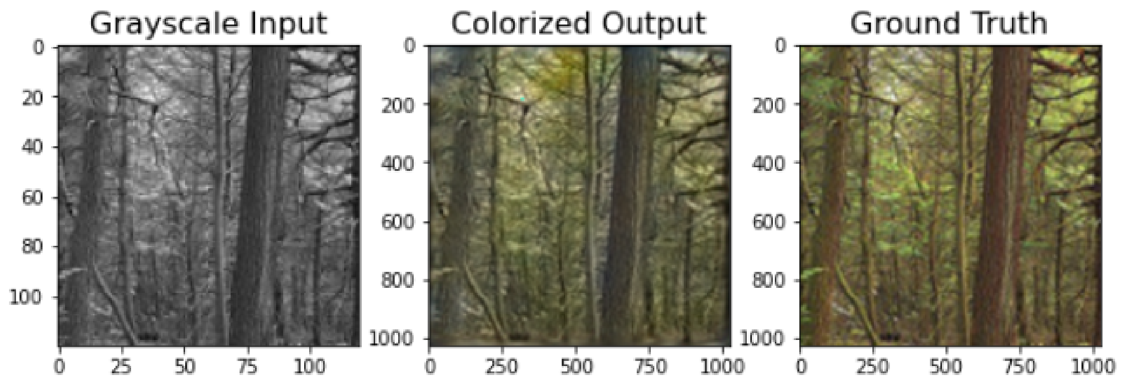
generator.png

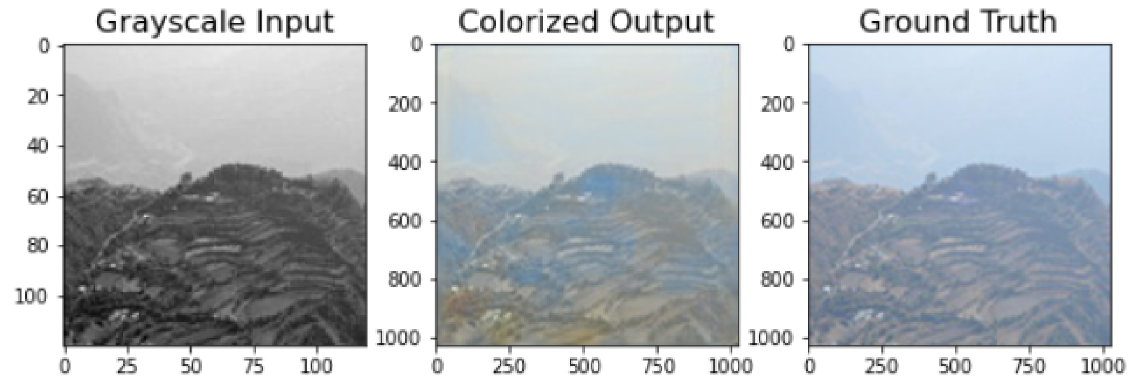
4. RESULTS

```
y = generator( test_x[0 : ] ).numpy()
```

```
for i in range(len(test_x)):
  plt.figure(figsize=(10,10))
  or_image = plt.subplot(3,3,1)
  or_image.set_title('Grayscale Input', fontsize=16)
  plt.imshow( test_x[i].reshape((120,120)) , cmap='gray' )

  in_image = plt.subplot(3,3,2)
  image = Image.fromarray( ( y[i] * 255 ).astype( 'uint8' ) ).resize( ( 1024 , 1024 ) )
  image = np.asarray( image )
  in_image.set_title('Colorized Output', fontsize=16)
  plt.imshow( image )

  ou_image = plt.subplot(3,3,3)
  image = Image.fromarray( ( test_y[i] * 255 ).astype( 'uint8' ) ).resize( ( 1024 , 1024 ) )
  ou_image.set_title('Ground Truth', fontsize=16)
  plt.imshow( image )

  plt.show()
```







CONCLUSION-

In summary, the approach employing Generative Adversarial Networks (GANs) for image colorization demonstrates considerable promise in generating high-fidelity colorized images. The dataset comprising 3000 RGB images, preprocessed into grayscale for labeling, serves as a robust training corpus. The iterative training process, with alternating updates between the generator and discriminator, fosters the gradual enhancement of the generator's proficiency in producing realistic colorizations while continually challenging the discriminator's discernment capabilities. By employing the L2/MSE loss function and the Adam optimizer with a learning rate of 0.0005, the training process ensures stable and efficient convergence. Overall, this methodology presents a significant stride in advancing image colorization techniques, paving the way for the creation of visually captivating and authentic colorized images across diverse domains.

REFERENCES -

1. Zhang, R., Isola, P., & Efros, A. A. (2016). Colorful Image Colorization. European Conference on Computer Vision (ECCV). Retrieved from https://arxiv.org/abs/1603.08511

2. Iizuka, S., Simo-Serra, E., & Ishikawa, H. (2016). Let there be Color!: Joint End-to-end Learning of Global and Local Image Priors for Automatic Image Colorization with Simultaneous Classification. European Conference on Computer Vision (ECCV). Retrieved from https://arxiv.org/abs/1603.08511

3. Isola, P., Zhu, J. Y., Zhou, T., & Efros, A. A. (2017). Image-to-Image Translation with Conditional Adversarial Networks. Computer Vision and Pattern Recognition (CVPR). Retrieved from https://arxiv.org/abs/1611.07004

4. Zhu, J. Y., Park, T., Isola, P., & Efros, A. A. (2017). Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks. Computer Vision and Pattern Recognition (CVPR). Retrieved from https://arxiv.org/abs/1703.10593