```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remoun
```

Open in Colab

2. The GAN

In this section, we'll create our GAN model step-by-step with Keras. First, we'll implement the generator then the discriminator and finally the loss functions required by both of them.

A. Generator

Our generator (represented as G) will take in grayscale image x and produce a RGB image $G(x)$. Note, x will be a tensor of shape $(\ batch\ size\ ,\ 120\ ,\ 120\ ,\ 1\ )$ and the output $G(x)$ will have a shape $(\ batch\ size\ ,\ 120\ ,\ 120\ ,\ 3\ )$

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

```
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' )( conv

    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

The discriminator model, represented as D, will take in the *real image* y (from the training data) and the *generated image* $G(x)$ (from the generator) to output two probabilities.

- We train the discriminator in such a manner that is able to differentiate the *real images* and the generated *images*. So, we train the model such that y produces a output of 1.0 and $G(x)$ produces an output of 0.0.
- Note that instead of using hard labels like 1.0 and 0.0, we use soft labels which are close to 1 and 0. So for a hard label of 1.0, the soft label will be $(1 - \epsilon)$ where ϵ is picked uniformly from $(0, 0.1]$

```
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 Functions

We'll now implement the loss functions for our GAN model. As you might know that we have two loss functions, one for the generator and another for the discriminator.

- For our generator, we'll use the L2/MSE loss function.
- For optimization, we use the Adam optimizer with a learning rate of 0.0005

```
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_c
    fake_loss = cross_entropy(tf.zeros_like(fake_output) + tf.random.uniform( shape=fake_output.shape , maxval=0.1  ) , fake
    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()
```

3. Training The GAN

So finally, we'll train our GAN on the dataset, we prepared earlier.

```
@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 )

    # 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))
```

Run the cell below, to start the training

```
num_epochs = 50

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

```
0
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(10, 120, 120, 1)
1
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
```

```
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
(64, 120, 120, 1)
```

4. Results

We plotted the input, output and the original images respectively, from a part of the dataset to find out the results.

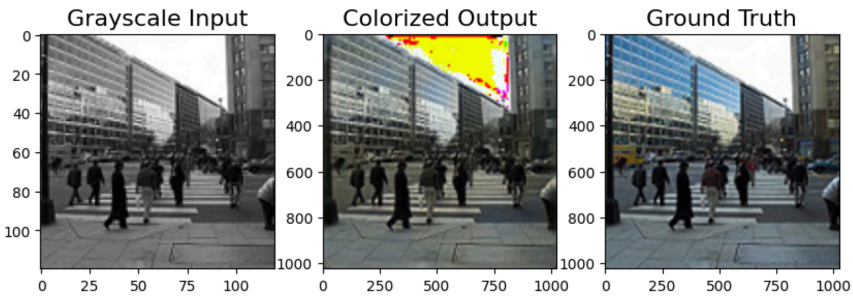
```
generator.save('gen.h5')
discriminator.save('dis.h5')
```

```
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` wil
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics` wil
```

```
from keras.models import load_model
generator = load_model('gen.h5')
y = generator( test_x[0 : ] ).numpy()
```

```
WARNING:tensorflow:No training configuration found in the save file, so the model was *not* compiled. Compile it manuall
```

```
# 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()

# FOR SINGLE IMAGE RESULTS
i = 25
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()
```



Therefore, the overwhelming reslts depicts the power of GANs and the disruption which can be broght through them.

```
Start coding or generate with AI.
```