EXPLO PRESENTATION

Problem statement :

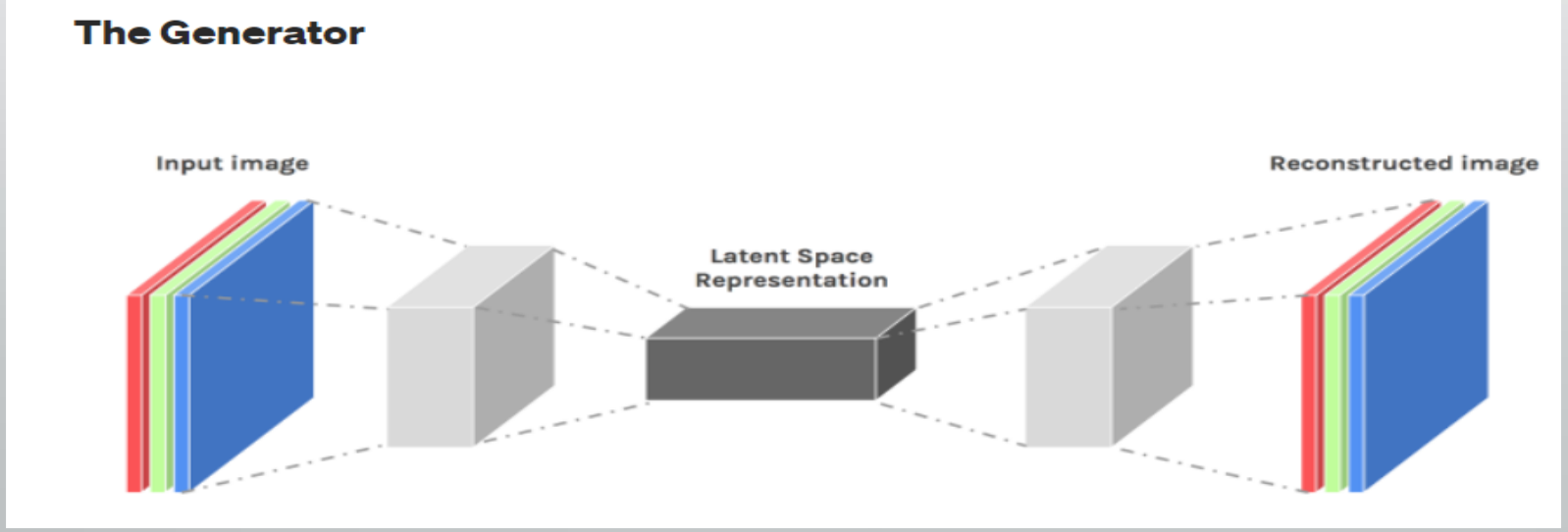
We often come close to some photos that belong to our grandparents and as they are black and white we always wonder what were the actual colours, so to overcome this problem we will create a deep learning model that works along with Gan's to colorize black and white photos.

It's not the first time someone is trying to solve this problem it has been tried multiple time using different methods like using regression models along with convo nets but applying Gan's is still the most efficient way.

solution

One of the most exciting applications of deep learning is colorizing black and white images. This task needed a lot of human input and hardcoding several years ago but now the whole process can be done end-to-end with the power of AI and deep learning. You might think that you need huge amount of data or long training times to train your model from scratch for this task but in the last few weeks we worked on this and tried many different model architectures, loss functions, training strategies, etc. and finally developed an efficient strategy to train such a model, using the latest advances in deep learning, on a rather small dataset and with really short training times.

The model is known as **Generative adversarial network(GAN's)** which uses a L1 loss function which is also known as Least Absolute Deviations along with Adversarial Loss which helps to solve the problem in an unsupervised manner.



How does Gan's work

In a GAN we have a generator and a discriminator model which learn to solve a problem together. In our setting, the generator model takes a grayscale image (1-channel image) and produces a 2-channel image, a channel for *a and another for *b. The discriminator, takes these two produced channels and concatenates them with the input grayscale image and decides whether this new 3-channel image is fake or real. Offcourse the discriminator also needs to see some real images (3-channel images again in Lab color space) that are not produced by the generator and should learn that they are real.
So what about the "condition" we mentioned? Well, that grayscale image which both the generator and discriminator see is the condition that we provide to both models in our GAN and expect that the they take this condition into consideration.
Let's take a look at the math. Consider x as the grayscale image, z as the input noise for the generator, and y as the 2-channel output we want from the generator (it can also represent the 2 color channels of a real image). Also, G is the generator model and D is the discriminator. Then the loss for our conditional GAN will be:

$$\mathcal{L}_{cGAN}(G, D) = \mathbb{E}_{x,y}[\log D(x, y)] + \mathbb{E}_{x,z}[\log(1 - D(x, G(x, z)))]$$

Loss function we optimize

$$\mathcal{L}_{L1}(G) = \mathbb{E}_{x,y,z}[||y - G(x,z)||_1]$$

If we use L1 loss , the model still learns to colorize the images but it will be conservative and most of the time uses colors like "gray" or "brown" because when it doubts which color is the best, it takes the average and uses these colors to reduce the L1 loss as much as possible (it is similar to the blurring effect of L1 or L2 loss in super resolution task). Also, the L1 Loss is preferred over L2 loss (or mean squared error) because it reduces that effect of producing gray-ish images. So, our combined loss function will be:

$$G^* = \arg\min_G \max_D \mathcal{L}_{cGAN}(G,D) + \lambda \mathcal{L}_{L1}(G)$$

where λ is a coefficient to balance the contribution of the two losses to the final loss (of course the discriminator loss does not involve the L1 loss).

Method to create our model

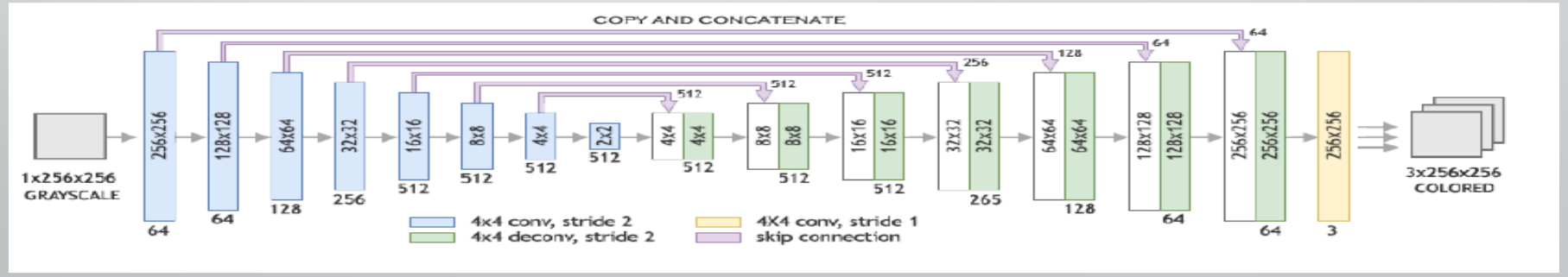
Image colorization is an image-to-image translation problem that maps a high dimensional input to a high dimensional output. It can be seen as pixel-wise regression problem where structure in the input is highly aligned with structure in the output. That means the network needs not only to generate an output with the same spatial dimension as the input, but also to provide color information to each pixel in the grayscale input image. We provide an entirely convolutional model architecture using a regression loss as our baseline and then extend the idea to adversarial nets.
In this work we utilize the L*a*b* color space for the colorization task. This is because L*a*b* color space contains dedicated channel to depict the brightness of the image and the color information is fully encoded in the remaining two channels. As a result, this prevents any sudden variations in both color and brightness through small perturbations in intensity values that are experienced through RGB.

We will create our model in three steps:

1. Creating a baseline network
2. Creating a convolutional Gan layer
3. Training the model

Baseline Network

For our baseline model, we follow the “fully convolutional network" model where the fully connected layers are replaced by convolutional layers which include upsampling instead of pooling operators.
This idea is based on encoder-decoder networks where input is progressively downsampled using a series of contractive encoding layers, and then the process is reversed using a series of expansive decoding layers to reconstruct the input.
This architecture is called U-Net , where skip connections are added between layer i and layer n-i. The architecture of the model is symmetric, with n encoding units and n decoding units. The contracting path consists of 4-4 convolution layers with stride 2 for downsampling, each followed by batch normalization and Leaky-ReLU activation function with the slope of 0.2. The number of channels are doubled after each step. Each unit in the expansive path consists of a 4 - 4 transposed convolutional layer with stride 2 for upsampling, concatenation with the activation map of the mirroring layer in the contracting path, followed by batch normalization and ReLU activation function. The last layer of the network is a 1 -1 convolution which is equivalent to cross-channel parametric pooling layer.
We use tanh function for the last layer as proposed by . The number of channels in the output layer is 3 with L*a*b* color space.

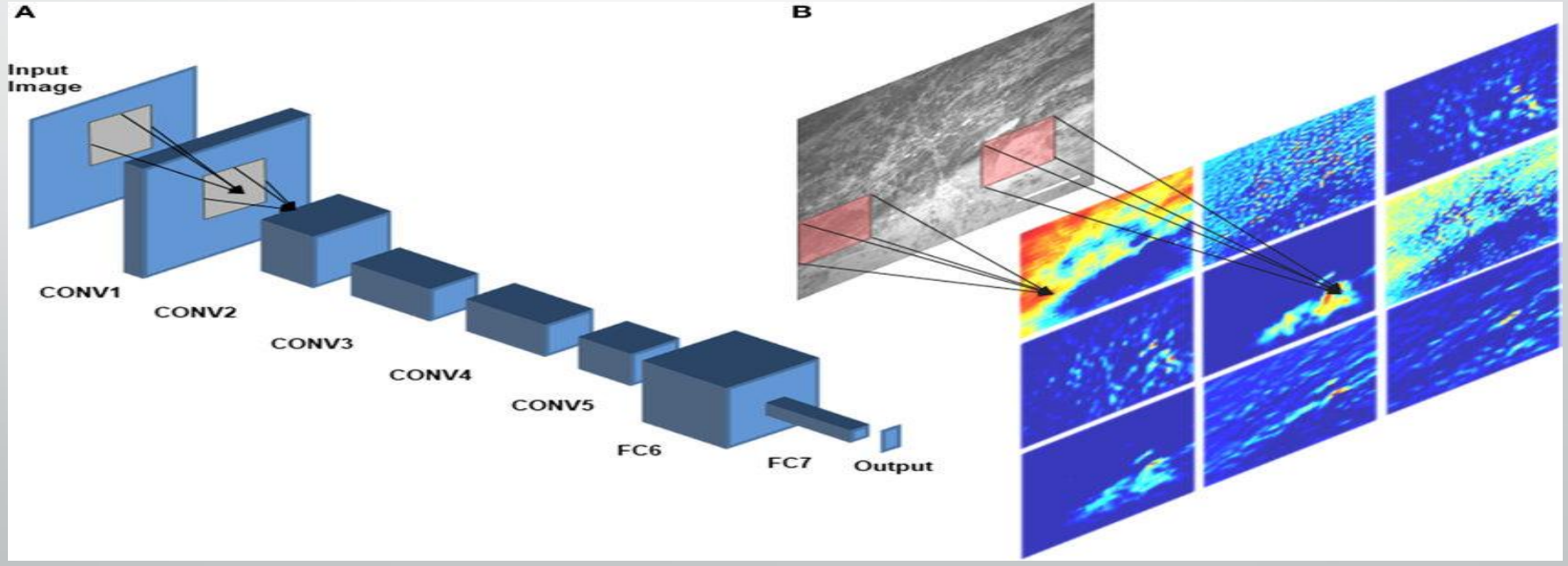


This is our baseline network:

```
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
```

Convolutional GAN

For the generator and discriminator models, we followed Deep Convolutional GANs (DCGAN) guidelines and employed convolutional networks in both generator and discriminator architectures.
The architecture was also modified as a conditional GAN instead of a traditional DCGAN; we also follow guideline in and provide noise only in the form of dropout , applied on several layers of our generator. The architecture of generator G is the same as the baseline. For discriminator D, we use similar architecture as the baselines contractive path: a series of 4 - 4 convolutional layers with stride 2 with the number of channels being doubled after each downsampling. All convolution layers are followed by batch normalization, leaky ReLU activation with slope 0.2. After the last layer, a convolution is applied to map to a 1 dimensional output, followed by a sigmoid function to return a probability value of the input being real or fake. The input of the discriminator is a colored image either coming from the generator or true labels, concatenated with the grayscale image.



Creating the convo Gan layer:

```
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
```

Training The Model

For training our network, we used Adam optimization and weight initialization. We used initial learning rate of 2 *10^34 for both generator and discriminator and manually decayed the learning rate by a factor of 10 whenever the loss function started to plateau. For the hyper-parameter we followed the protocol and chose lambda= 100, which forces the generator to produce images similar to ground truth. GANs have been known to be very difficult to train as it requires finding a Nash equilibrium of a non-convex game with continuous, high dimensional parameters .

We followed a set of constraints and techniques to encourage convergence of our convolutional GAN and make it stable to train :

1. **Alternative Cost Function**
2. **One Sided Label Smoothing**
3. **Batch Normalization**
4. **All Convolutional Net**
5. **Reduced Momentum**
6. **LeakyReLU Activation Function**

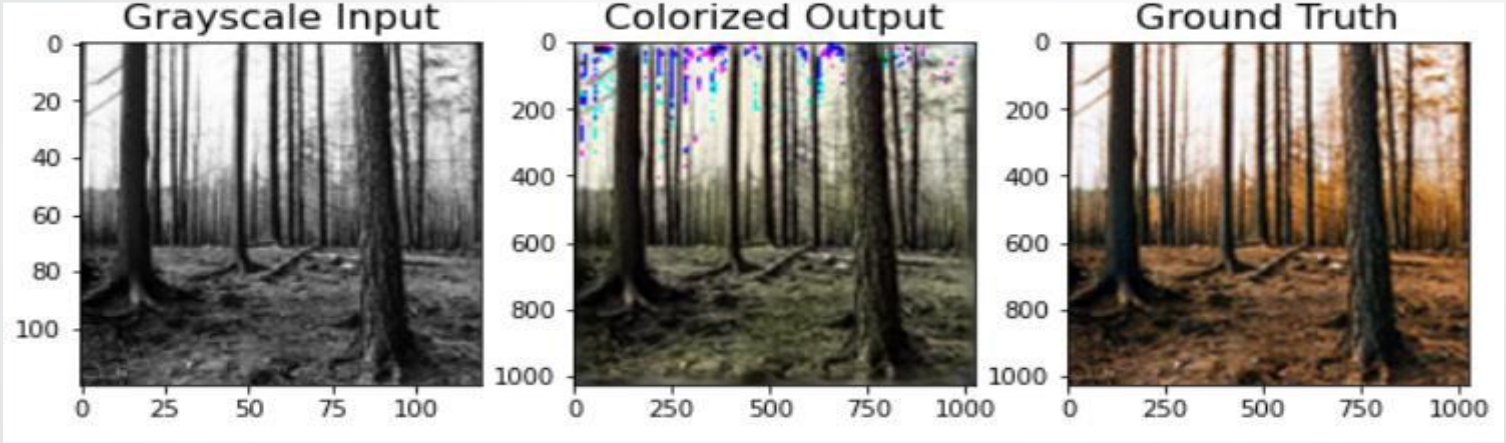
Training Layer:

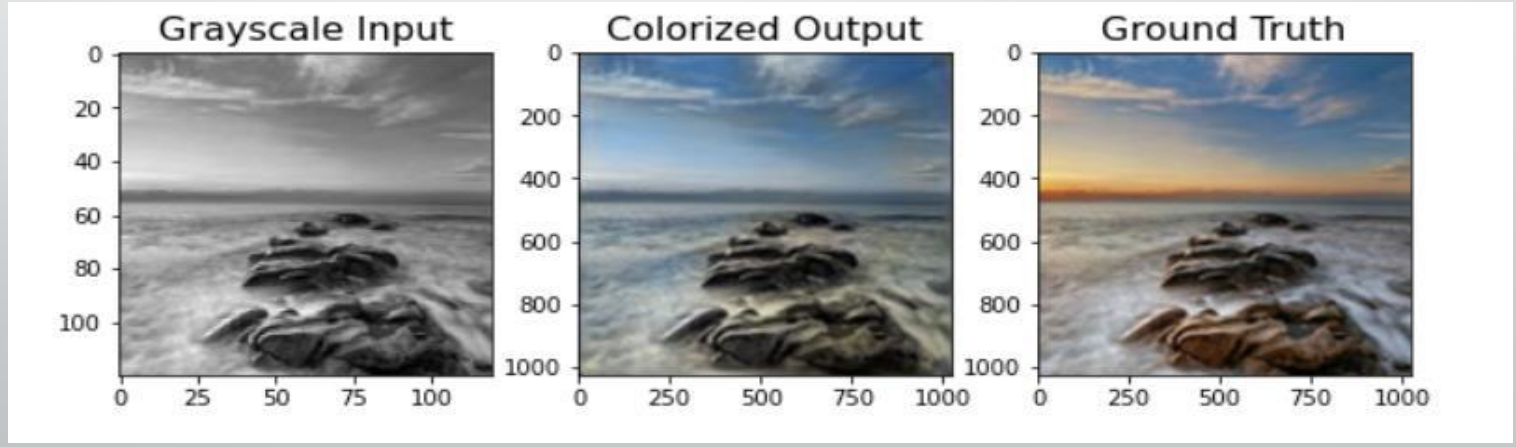
```python
@tf.function
def train_step( input_x , real_y ):

    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))
```

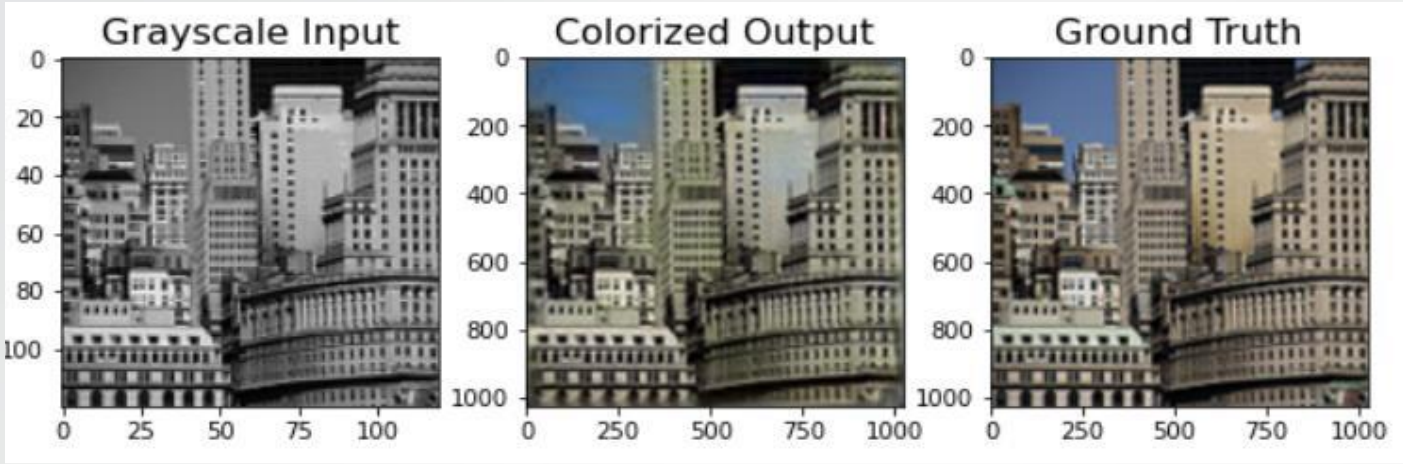
```python
num_epochs = 100

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 )
```

Results

The result is shown in three columns, in the first column we have gray scale image then in the second column we have generated output using the model the in the third column we have original image.







In this study, we were able to automatically colorize grayscale images using GAN, to an acceptable visual degree.

Many of the images generated by U-Net had a brownish hue in the results known as the “Sepia effect" across L*a*b* color space. This is due to the L2 also known as Least Square Errors loss function that was applied to the baseline CNN, which is known to cause a blurring effect.

Mis-colorization was a frequent occurrence with images containing high levels of textured details. This leads us to believe that the model has identified these regions as grass since many images in the training set contained leaves or grass in an open field.

Overall the results look promising and the accuracy seems pretty good.

Thank You