Image Denoising
Bharat Pareek 21113039

Introduction:

In this project, we explored the fascinating world of image-to-image translation, focusing specifically on converting low light images to high light images. This technology is incredibly useful for improving the quality of photos taken in poor lighting conditions, making them clearer and more visually appealing.

To achieve this, we used a Pix2Pix Generative Adversarial Network (GAN) enhanced with W-loss (Wasserstein loss) and Gradient Penalty. GANs are a powerful class of deep learning models that can generate high-quality images by learning from examples, and Pix2Pix is a specific type of GAN designed for image-to-image translation tasks.

GANs (Generative Adversarial Networks):

A GAN consists of two neural networks - the Generator and the Discriminator. The Generator creates images, while the Discriminator tries to distinguish between real and generated images. Through this adversarial process, the Generator learns to create increasingly realistic images.

Pix2Pix:

This is a specific GAN architecture designed for paired image-to-image translation tasks. It takes an input image (e.g., a low light image) and translates it to the desired output image (e.g., a high light image).

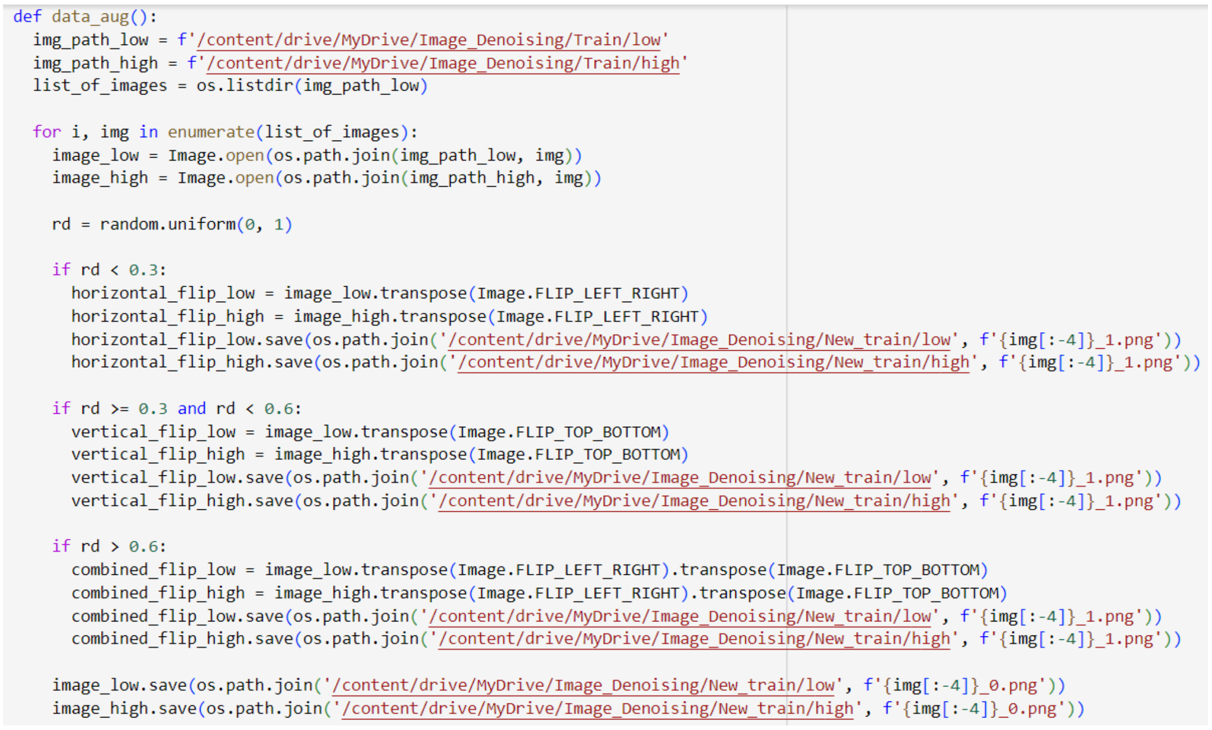
Here's a detailed breakdown of our approach:

Data Preparation:

We had a dataset of 485 low light and high light images. These were then augmented to 1940 images by using flipping, contrast, saturation, hue, cropping etc.

These were then divided into 80%(1552 images) training and 20%(388 images) validation.

For random flipping:



```
def data_aug():
  img_path_low = f'/content/drive/MyDrive/Image_Denoising/Train/low'
  img_path_high = f'/content/drive/MyDrive/Image_Denoising/Train/high'
  list_of_images = os.listdir(img_path_low)

  for i, img in enumerate(list_of_images):
    image_low = Image.open(os.path.join(img_path_low, img))
    image_high = Image.open(os.path.join(img_path_high, img))

    rd = random.uniform(0, 1)

    if rd < 0.3:
      horizontal_flip_low = image_low.transpose(Image.FLIP_LEFT_RIGHT)
      horizontal_flip_high = image_high.transpose(Image.FLIP_LEFT_RIGHT)
      horizontal_flip_low.save(os.path.join('/content/drive/MyDrive/Image_Denoising/New_train/low', f'{img[:-4]}_1.png'))
      horizontal_flip_high.save(os.path.join('/content/drive/MyDrive/Image_Denoising/New_train/high', f'{img[:-4]}_1.png'))

    if rd >= 0.3 and rd < 0.6:
      vertical_flip_low = image_low.transpose(Image.FLIP_TOP_BOTTOM)
      vertical_flip_high = image_high.transpose(Image.FLIP_TOP_BOTTOM)
      vertical_flip_low.save(os.path.join('/content/drive/MyDrive/Image_Denoising/New_train/low', f'{img[:-4]}_1.png'))
      vertical_flip_high.save(os.path.join('/content/drive/MyDrive/Image_Denoising/New_train/high', f'{img[:-4]}_1.png'))

    if rd > 0.6:
      combined_flip_low = image_low.transpose(Image.FLIP_LEFT_RIGHT).transpose(Image.FLIP_TOP_BOTTOM)
      combined_flip_high = image_high.transpose(Image.FLIP_LEFT_RIGHT).transpose(Image.FLIP_TOP_BOTTOM)
      combined_flip_low.save(os.path.join('/content/drive/MyDrive/Image_Denoising/New_train/low', f'{img[:-4]}_1.png'))
      combined_flip_high.save(os.path.join('/content/drive/MyDrive/Image_Denoising/New_train/high', f'{img[:-4]}_1.png'))

    image_low.save(os.path.join('/content/drive/MyDrive/Image_Denoising/New_train/low', f'{img[:-4]}_0.png'))
    image_high.save(os.path.join('/content/drive/MyDrive/Image_Denoising/New_train/high', f'{img[:-4]}_0.png'))
```

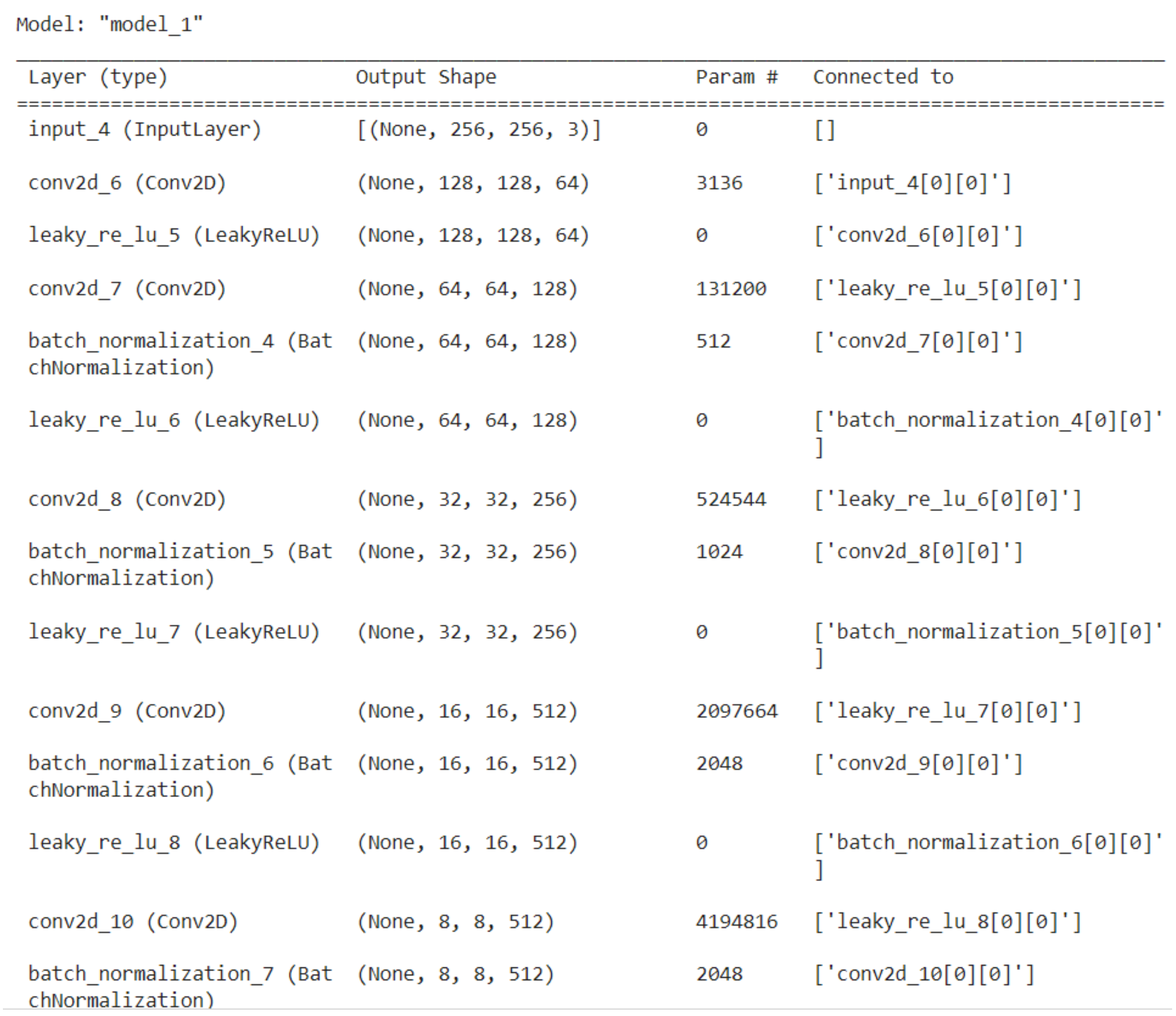
For other changes:

```python
def augment_pair(low_light_image_in, high_light_image_in):

    # Random contrast adjustment
    contrast_factor = tf.random.uniform([], 0.8, 1.2)
    low_light_image = tf.image.adjust_contrast(low_light_image_in, contrast_factor)
    high_light_image = tf.image.adjust_contrast(high_light_image_in, contrast_factor)

    # Random saturation adjustment
    saturation_factor = tf.random.uniform([], 0.8, 1.2)
    low_light_image = tf.image.adjust_saturation(low_light_image, saturation_factor)
    high_light_image = tf.image.adjust_saturation(high_light_image, saturation_factor)

    # Random hue adjustment
    hue_delta = tf.random.uniform([], -0.02, 0.02)
    low_light_image = tf.image.adjust_hue(low_light_image, hue_delta)
    high_light_image = tf.image.adjust_hue(high_light_image, hue_delta)

    # crop
    height = tf.shape(low_light_image)[0]
    width = tf.shape(low_light_image)[1]

    # Calculate the amount to crop
    crop_height = tf.cast(0.1 * tf.cast(height, tf.float32), tf.int32)
    crop_width = tf.cast(0.1 * tf.cast(width, tf.float32), tf.int32)

    # Crop the image
    low_light_image = low_light_image[crop_height:-crop_height, crop_width:-crop_width, :]
    high_light_image = high_light_image[crop_height:-crop_height, crop_width:-crop_width, :]

    # Resize back to original size
    low_light_image = tf.image.resize(low_light_image, [400, 600])
    high_light_image = tf.image.resize(high_light_image, [400, 600])

    return low_light_image, high_light_image
```

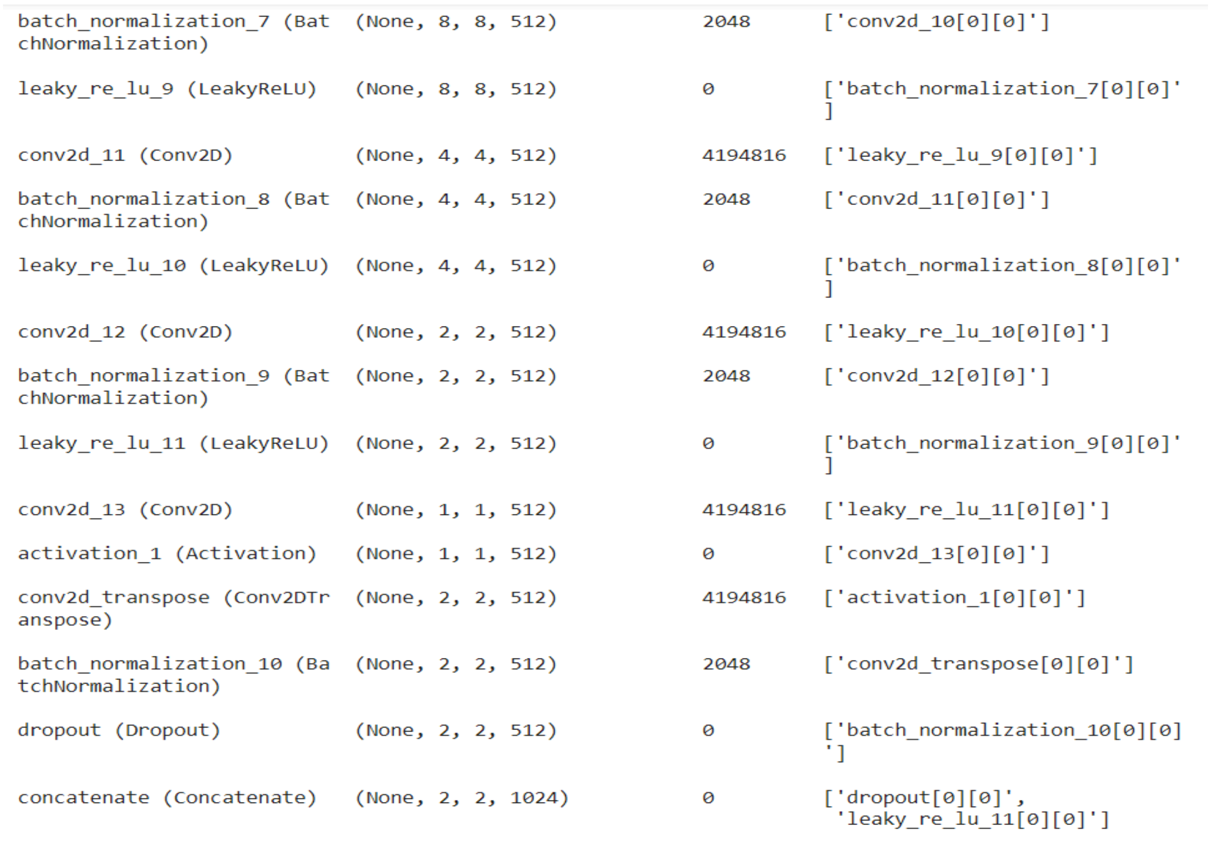
Model Architecture

- **Generator**: The Generator is responsible for transforming low light images into high light images. It is a UNet architecture. It consists of a series of convolutional layers that gradually convert the input image into the desired output.

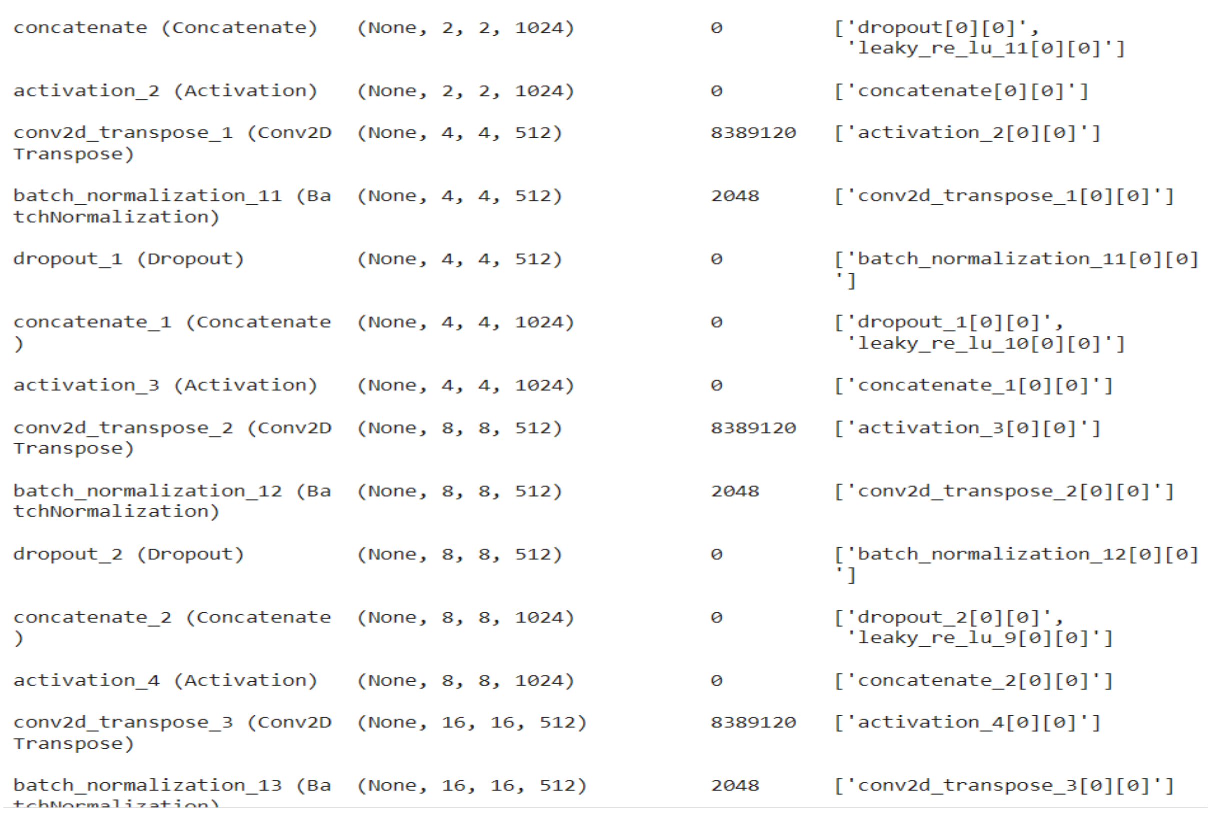
This is a UNet architecture is inspired from **Image-to-Image Translation with Conditional Adversarial Networks** paper



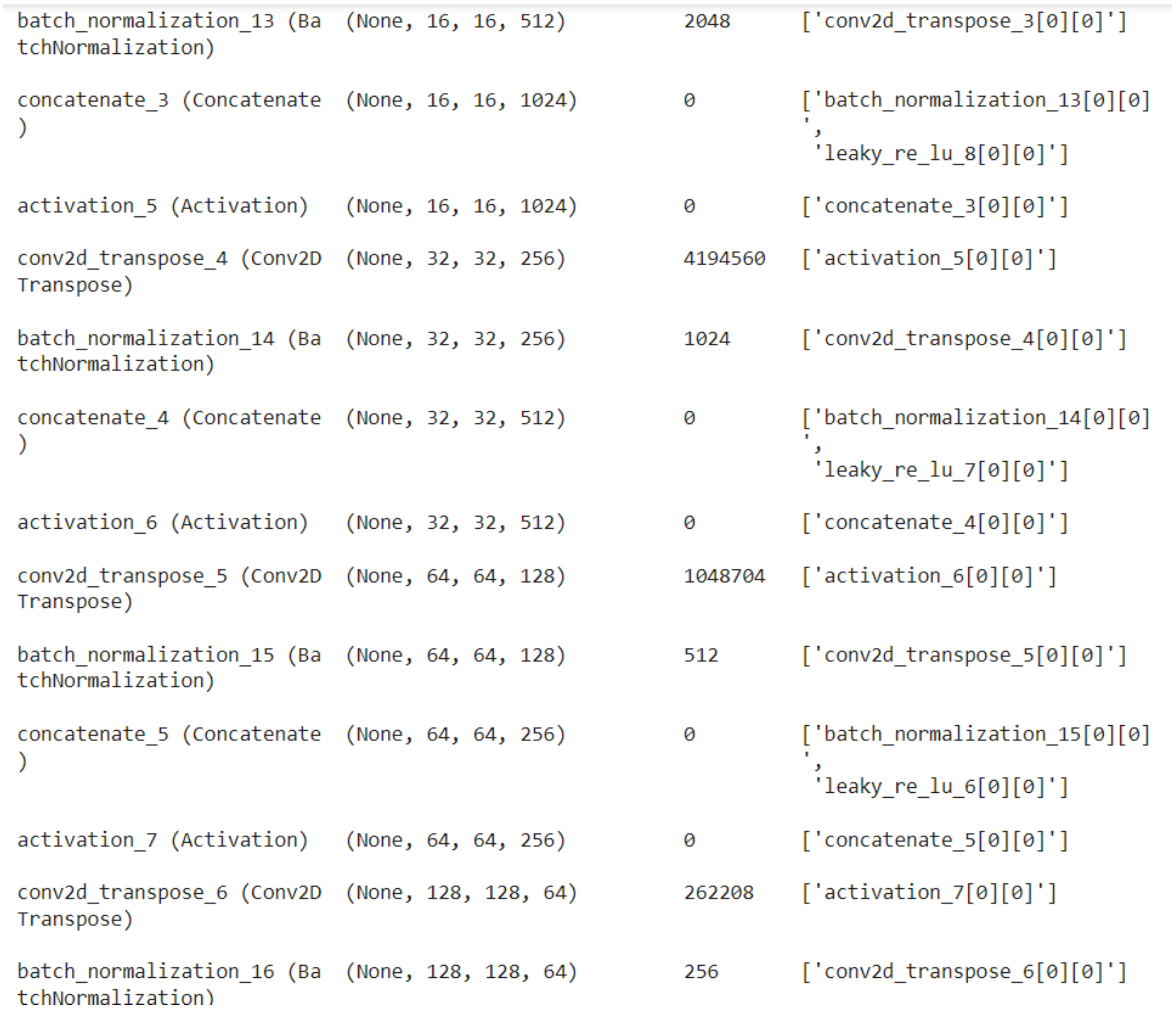
```
Model: "model_1"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_4 (InputLayer)           [(None, 256, 256, 3)]  0         []

 conv2d_6 (Conv2D)              (None, 128, 128, 64)  3136       ['input_4[0][0]']

 leaky_re_lu_5 (LeakyReLU)      (None, 128, 128, 64)  0          ['conv2d_6[0][0]']

 conv2d_7 (Conv2D)              (None, 64, 64, 128)  131200      ['leaky_re_lu_5[0][0]']

 batch_normalization_4 (Bat     (None, 64, 64, 128)  512         ['conv2d_7[0][0]']
 chNormalization)

 leaky_re_lu_6 (LeakyReLU)      (None, 64, 64, 128)  0           ['batch_normalization_4[0][0]'
                                                                 ]

 conv2d_8 (Conv2D)              (None, 32, 32, 256)  524544      ['leaky_re_lu_6[0][0]']

 batch_normalization_5 (Bat     (None, 32, 32, 256)  1024        ['conv2d_8[0][0]']
 chNormalization)

 leaky_re_lu_7 (LeakyReLU)      (None, 32, 32, 256)  0           ['batch_normalization_5[0][0]'
                                                                 ]

 conv2d_9 (Conv2D)              (None, 16, 16, 512)  2097664     ['leaky_re_lu_7[0][0]']

 batch_normalization_6 (Bat     (None, 16, 16, 512)  2048        ['conv2d_9[0][0]']
 chNormalization)

 leaky_re_lu_8 (LeakyReLU)      (None, 16, 16, 512)  0           ['batch_normalization_6[0][0]'
                                                                 ]

 conv2d_10 (Conv2D)             (None, 8, 8, 512)    4194816     ['leaky_re_lu_8[0][0]']

 batch_normalization_7 (Bat     (None, 8, 8, 512)    2048        ['conv2d_10[0][0]']
 chNormalization)
```



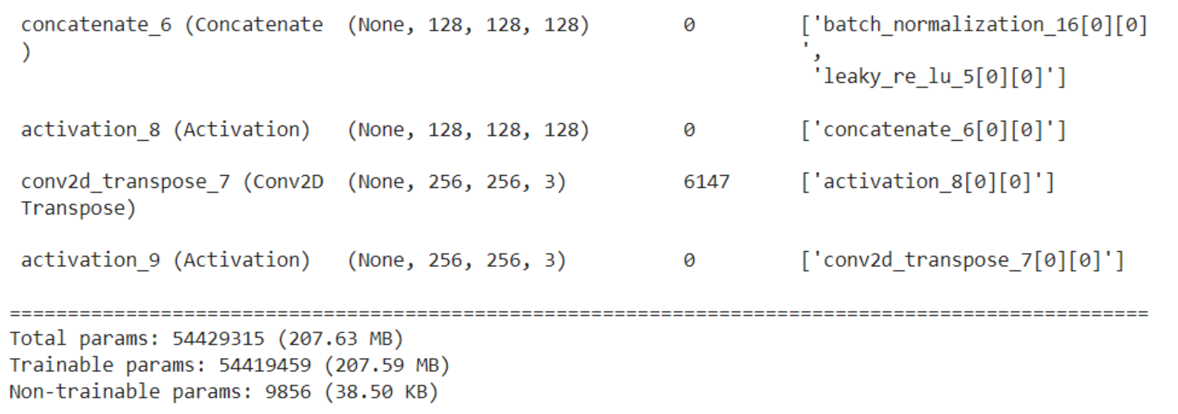
```
 batch_normalization_7 (Bat  (None, 8, 8, 512)           2048      ['conv2d_10[0][0]']
 chNormalization)

 leaky_re_lu_9 (LeakyReLU)   (None, 8, 8, 512)           0         ['batch_normalization_7[0][0]'
                                                                    ]

 conv2d_11 (Conv2D)          (None, 4, 4, 512)           4194816   ['leaky_re_lu_9[0][0]']

 batch_normalization_8 (Bat  (None, 4, 4, 512)           2048      ['conv2d_11[0][0]']
 chNormalization)

 leaky_re_lu_10 (LeakyReLU)  (None, 4, 4, 512)           0         ['batch_normalization_8[0][0]'
                                                                    ]

 conv2d_12 (Conv2D)          (None, 2, 2, 512)           4194816   ['leaky_re_lu_10[0][0]']

 batch_normalization_9 (Bat  (None, 2, 2, 512)           2048      ['conv2d_12[0][0]']
 chNormalization)

 leaky_re_lu_11 (LeakyReLU)  (None, 2, 2, 512)           0         ['batch_normalization_9[0][0]'
                                                                    ]

 conv2d_13 (Conv2D)          (None, 1, 1, 512)           4194816   ['leaky_re_lu_11[0][0]']

 activation_1 (Activation)   (None, 1, 1, 512)           0         ['conv2d_13[0][0]']

 conv2d_transpose (Conv2DTr  (None, 2, 2, 512)           4194816   ['activation_1[0][0]']
 anspose)

 batch_normalization_10 (Ba  (None, 2, 2, 512)           2048      ['conv2d_transpose[0][0]']
 tchNormalization)

 dropout (Dropout)           (None, 2, 2, 512)           0         ['batch_normalization_10[0][0]
                                                                    ']

 concatenate (Concatenate)   (None, 2, 2, 1024)          0         ['dropout[0][0]',
                                                                     'leaky_re_lu_11[0][0]']
```



```
 concatenate (Concatenate)   (None, 2, 2, 1024)          0         ['dropout[0][0]',
                                                                     'leaky_re_lu_11[0][0]']

 activation_2 (Activation)   (None, 2, 2, 1024)          0         ['concatenate[0][0]']

 conv2d_transpose_1 (Conv2D  (None, 4, 4, 512)           8389120   ['activation_2[0][0]']
 Transpose)

 batch_normalization_11 (Ba  (None, 4, 4, 512)           2048      ['conv2d_transpose_1[0][0]']
 tchNormalization)

 dropout_1 (Dropout)         (None, 4, 4, 512)           0         ['batch_normalization_11[0][0]
                                                                    ']

 concatenate_1 (Concatenate  (None, 4, 4, 1024)          0         ['dropout_1[0][0]',
 )                                                                   'leaky_re_lu_10[0][0]']

 activation_3 (Activation)   (None, 4, 4, 1024)          0         ['concatenate_1[0][0]']

 conv2d_transpose_2 (Conv2D  (None, 8, 8, 512)           8389120   ['activation_3[0][0]']
 Transpose)

 batch_normalization_12 (Ba  (None, 8, 8, 512)           2048      ['conv2d_transpose_2[0][0]']
 tchNormalization)

 dropout_2 (Dropout)         (None, 8, 8, 512)           0         ['batch_normalization_12[0][0]
                                                                    ']

 concatenate_2 (Concatenate  (None, 8, 8, 1024)          0         ['dropout_2[0][0]',
 )                                                                   'leaky_re_lu_9[0][0]']

 activation_4 (Activation)   (None, 8, 8, 1024)          0         ['concatenate_2[0][0]']

 conv2d_transpose_3 (Conv2D  (None, 16, 16, 512)         8389120   ['activation_4[0][0]']
 Transpose)

 batch_normalization_13 (Ba  (None, 16, 16, 512)         2048      ['conv2d_transpose_3[0][0]']
 tchNormalization)
```

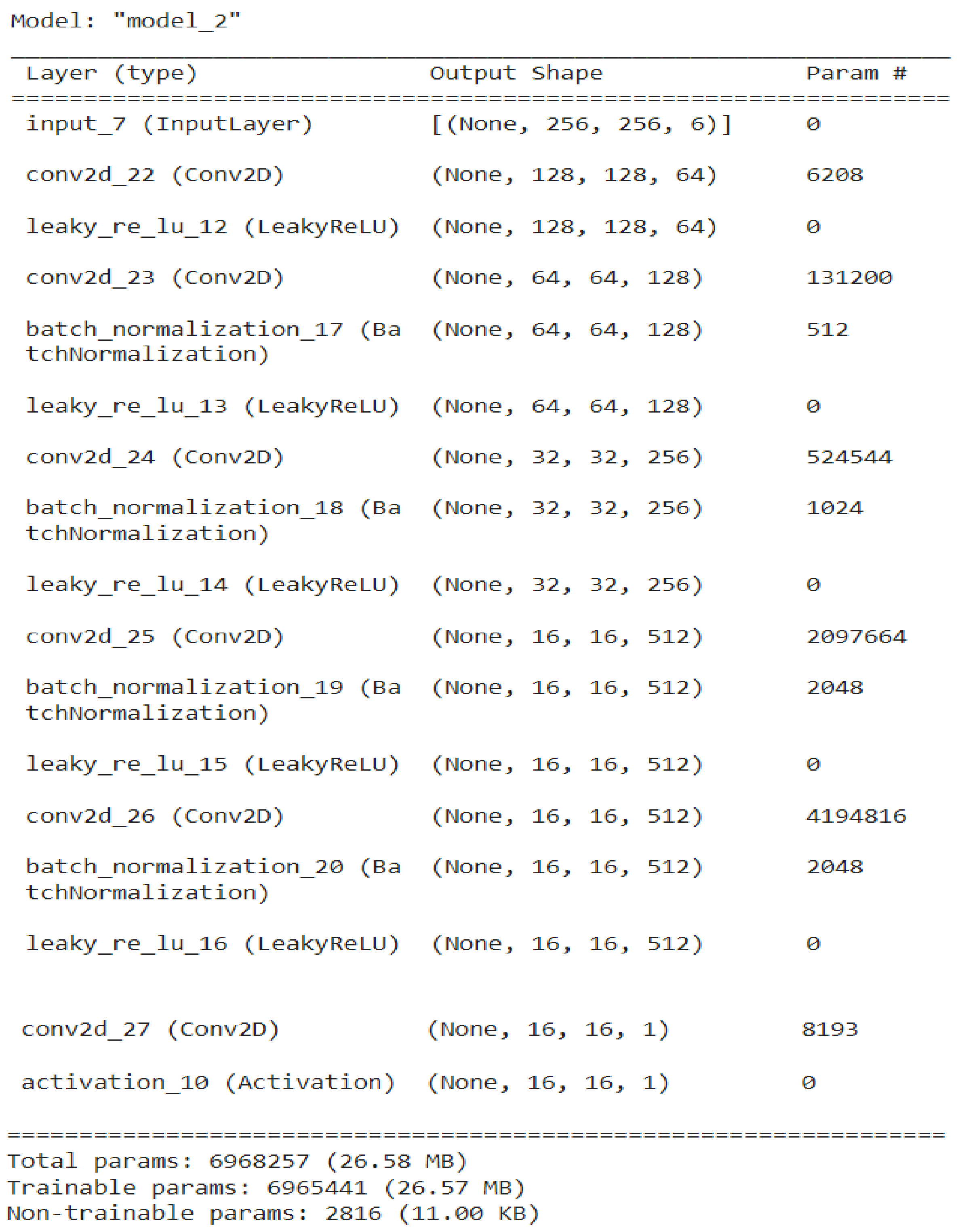


```
 batch_normalization_13 (Ba  (None, 16, 16, 512)          2048      ['conv2d_transpose_3[0][0]']
 tchNormalization)

 concatenate_3 (Concatenate  (None, 16, 16, 1024)         0         ['batch_normalization_13[0][0]
 )                                                                   ',
                                                                      'leaky_re_lu_8[0][0]']

 activation_5 (Activation)   (None, 16, 16, 1024)         0         ['concatenate_3[0][0]']

 conv2d_transpose_4 (Conv2D  (None, 32, 32, 256)          4194560   ['activation_5[0][0]']
 Transpose)

 batch_normalization_14 (Ba  (None, 32, 32, 256)          1024      ['conv2d_transpose_4[0][0]']
 tchNormalization)

 concatenate_4 (Concatenate  (None, 32, 32, 512)          0         ['batch_normalization_14[0][0]
 )                                                                   ',
                                                                      'leaky_re_lu_7[0][0]']

 activation_6 (Activation)   (None, 32, 32, 512)          0         ['concatenate_4[0][0]']

 conv2d_transpose_5 (Conv2D  (None, 64, 64, 128)          1048704   ['activation_6[0][0]']
 Transpose)

 batch_normalization_15 (Ba  (None, 64, 64, 128)          512       ['conv2d_transpose_5[0][0]']
 tchNormalization)

 concatenate_5 (Concatenate  (None, 64, 64, 256)          0         ['batch_normalization_15[0][0]
 )                                                                   ',
                                                                      'leaky_re_lu_6[0][0]']

 activation_7 (Activation)   (None, 64, 64, 256)          0         ['concatenate_5[0][0]']

 conv2d_transpose_6 (Conv2D  (None, 128, 128, 64)         262208    ['activation_7[0][0]']
 Transpose)

 batch_normalization_16 (Ba  (None, 128, 128, 64)         256       ['conv2d_transpose_6[0][0]']
 tchNormalization)
```



```
 concatenate_6 (Concatenate  (None, 128, 128, 128)        0         ['batch_normalization_16[0][0]
 )                                                                   ',
                                                                      'leaky_re_lu_5[0][0]']

 activation_8 (Activation)   (None, 128, 128, 128)        0         ['concatenate_6[0][0]']

 conv2d_transpose_7 (Conv2D  (None, 256, 256, 3)          6147      ['activation_8[0][0]']
 Transpose)

 activation_9 (Activation)   (None, 256, 256, 3)          0         ['conv2d_transpose_7[0][0]']

==================================================================================================
Total params: 54429315 (207.63 MB)
Trainable params: 54419459 (207.59 MB)
Non-trainable params: 9856 (38.50 KB)
```

- **Discriminator**: The Discriminator evaluates the quality of the generated images by comparing them to real high light images. It learns to differentiate between real and generated images, providing feedback to the Generator.



```
Model: "model_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_7 (InputLayer)        [(None, 256, 256, 6)]     0

 conv2d_22 (Conv2D)          (None, 128, 128, 64)      6208

 leaky_re_lu_12 (LeakyReLU)  (None, 128, 128, 64)      0

 conv2d_23 (Conv2D)          (None, 64, 64, 128)       131200

 batch_normalization_17 (Ba  (None, 64, 64, 128)       512
 tchNormalization)

 leaky_re_lu_13 (LeakyReLU)  (None, 64, 64, 128)       0

 conv2d_24 (Conv2D)          (None, 32, 32, 256)       524544

 batch_normalization_18 (Ba  (None, 32, 32, 256)       1024
 tchNormalization)

 leaky_re_lu_14 (LeakyReLU)  (None, 32, 32, 256)       0

 conv2d_25 (Conv2D)          (None, 16, 16, 512)       2097664

 batch_normalization_19 (Ba  (None, 16, 16, 512)       2048
 tchNormalization)

 leaky_re_lu_15 (LeakyReLU)  (None, 16, 16, 512)       0

 conv2d_26 (Conv2D)          (None, 16, 16, 512)       4194816

 batch_normalization_20 (Ba  (None, 16, 16, 512)       2048
 tchNormalization)

 leaky_re_lu_16 (LeakyReLU)  (None, 16, 16, 512)       0


 conv2d_27 (Conv2D)          (None, 16, 16, 1)         8193

 activation_10 (Activation)  (None, 16, 16, 1)         0

=================================================================
Total params: 6968257 (26.58 MB)
Trainable params: 6965441 (26.57 MB)
Non-trainable params: 2816 (11.00 KB)
```

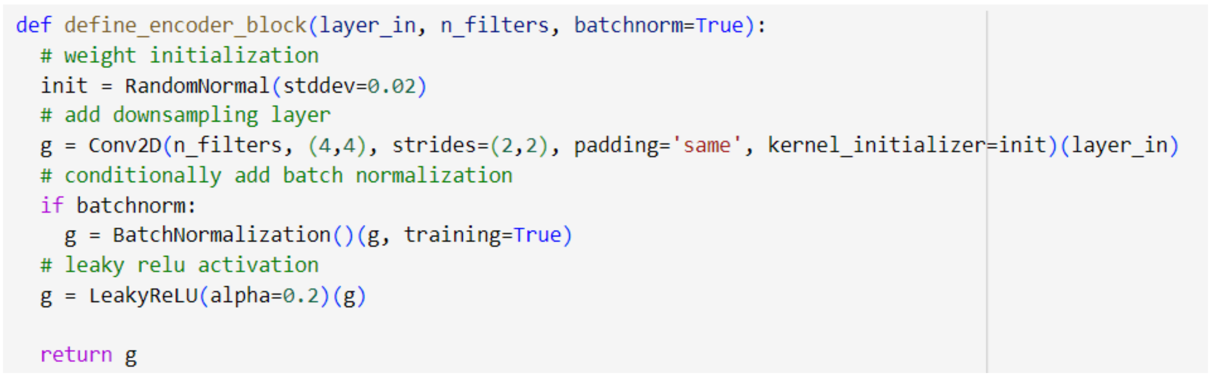
More About architectures:

Generator:

This architecture consists of two main parts: an encoder (contracting path) and a decoder (expanding path), connected by a bottleneck.

Encoder (Contracting Path):

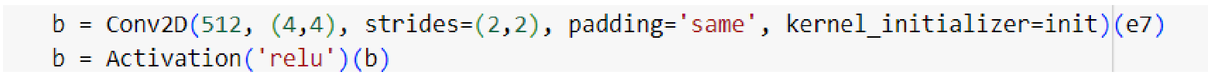
- The encoder compresses the input image into a lower-dimensional representation.
- It consists of repeated application of two 4x4 convolutional layers with a strides of 2, each followed by a Leaky rectified linear unit (LeakyReLU)(to avoid dying Relu problem) activation.
- Each down-sampling step doubles the number of feature channels while halving the spatial dimensions (height and width) of the feature maps.
- We also use BatchNormalization layers in between to stabilise and accelerate the training process by normalising the inputs of each layer, which helps in reducing internal covariate shift.



```
def define_encoder_block(layer_in, n_filters, batchnorm=True):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # add downsampling layer
  g = Conv2D(n_filters, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(layer_in)
  # conditionally add batch normalization
  if batchnorm:
    g = BatchNormalization()(g, training=True)
  # leaky relu activation
  g = LeakyReLU(alpha=0.2)(g)

  return g
```

Bottleneck:

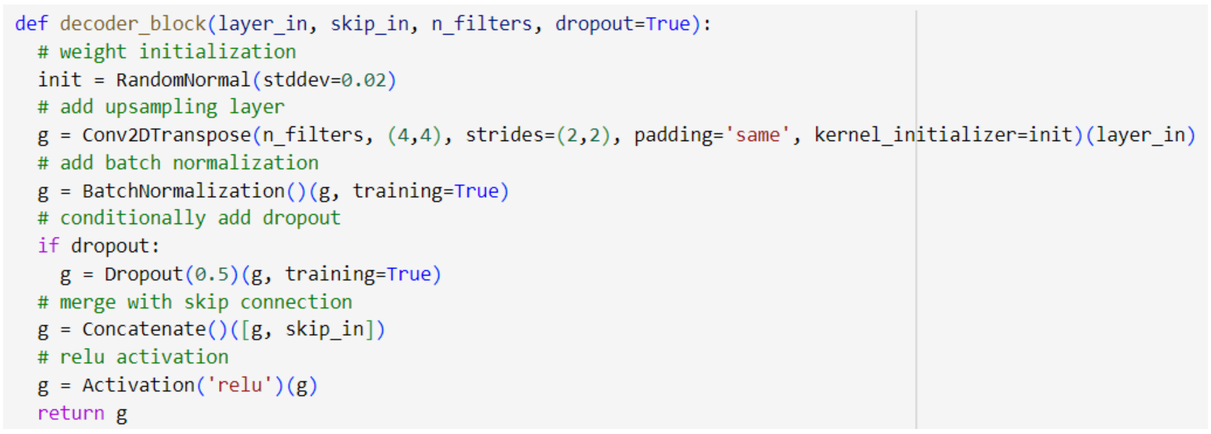
- The bottleneck is the layer at the bottom of the U-shape where the feature maps are at their smallest spatial resolution but contain the most abstract representation of the input image.
- It consists of two 4x4 convolutional layers, with stride of 2, followed by ReLU activations.



```
  b = Conv2D(512, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(e7)
  b = Activation('relu')(b)
```

Decoder (Expanding Path):

- The decoder reconstructs the compressed representation back to the original image size.
- It consists of up-sampling steps, each involving a 4x4 transposed convolution (also known as deconvolution) that doubles the spatial dimensions and halves the number of feature channels and a stride of 2 is used.
- Each up-sampling step is followed by BatchNormalization and a dropout layer(0.5) to avoid overfitting.
- Then there is concatenation with the corresponding feature maps from the encoder path (skip connections), and then ReLU activation.



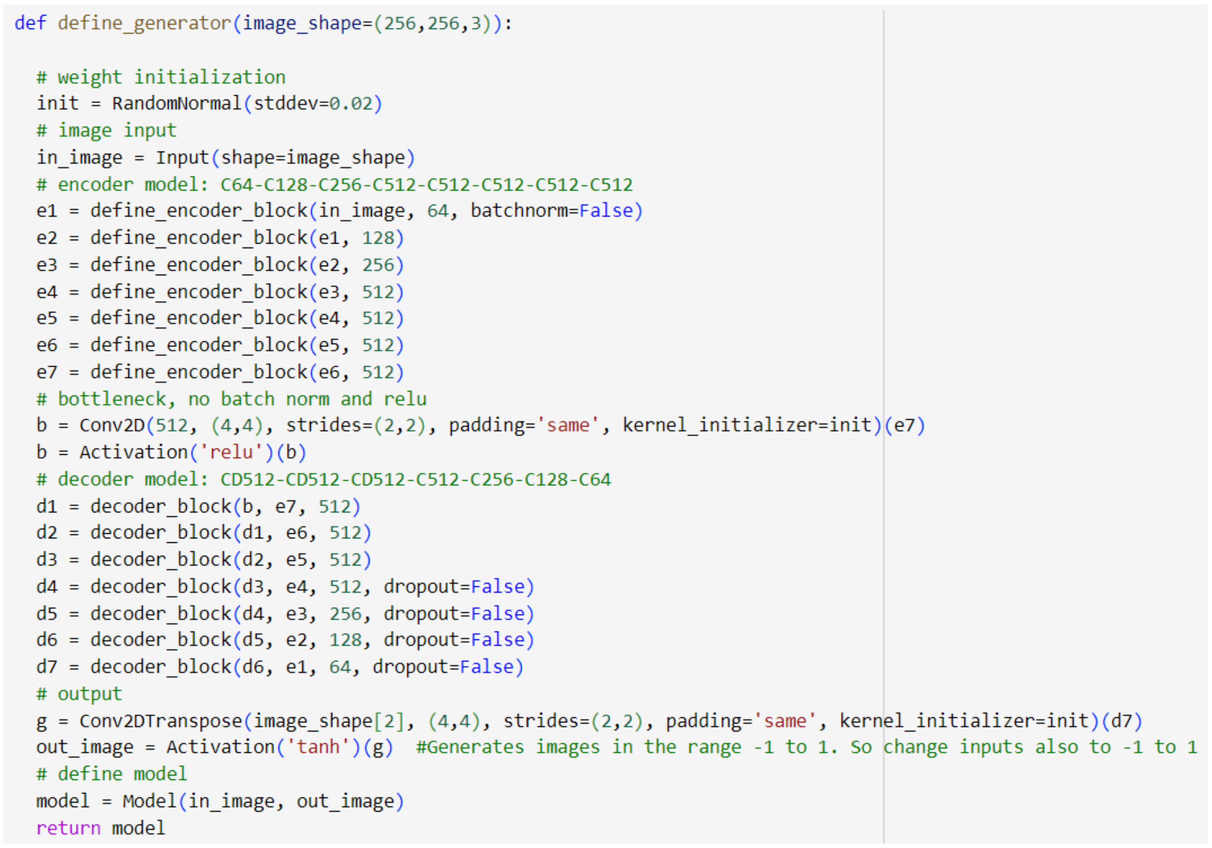
```python
def decoder_block(layer_in, skip_in, n_filters, dropout=True):
  # weight initialization
  init = RandomNormal(stddev=0.02)
  # add upsampling layer
  g = Conv2DTranspose(n_filters, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(layer_in)
  # add batch normalization
  g = BatchNormalization()(g, training=True)
  # conditionally add dropout
  if dropout:
    g = Dropout(0.5)(g, training=True)
  # merge with skip connection
  g = Concatenate()([g, skip_in])
  # relu activation
  g = Activation('relu')(g)
  return g
```

Skip Connections:

- Skip connections directly connect corresponding layers of the encoder and decoder paths. These help preserve spatial information that might be lost during down-sampling.
- These connections concatenate feature maps from the encoder to the corresponding decoder layers, ensuring detailed information is retained throughout the reconstruction process.

The kernels are initialised with Truncated Normal initializer (RandomNormal(stddev = 0.02)).

These Encoder + Bottleneck + Decoder form the Generator



```
def define_generator(image_shape=(256,256,3)):

  # weight initialization
  init = RandomNormal(stddev=0.02)
  # image input
  in_image = Input(shape=image_shape)
  # encoder model: C64-C128-C256-C512-C512-C512-C512-C512
  e1 = define_encoder_block(in_image, 64, batchnorm=False)
  e2 = define_encoder_block(e1, 128)
  e3 = define_encoder_block(e2, 256)
  e4 = define_encoder_block(e3, 512)
  e5 = define_encoder_block(e4, 512)
  e6 = define_encoder_block(e5, 512)
  e7 = define_encoder_block(e6, 512)
  # bottleneck, no batch norm and relu
  b = Conv2D(512, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(e7)
  b = Activation('relu')(b)
  # decoder model: CD512-CD512-CD512-C512-C256-C128-C64
  d1 = decoder_block(b, e7, 512)
  d2 = decoder_block(d1, e6, 512)
  d3 = decoder_block(d2, e5, 512)
  d4 = decoder_block(d3, e4, 512, dropout=False)
  d5 = decoder_block(d4, e3, 256, dropout=False)
  d6 = decoder_block(d5, e2, 128, dropout=False)
  d7 = decoder_block(d6, e1, 64, dropout=False)
  # output
  g = Conv2DTranspose(image_shape[2], (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d7)
  out_image = Activation('tanh')(g)  #Generates images in the range -1 to 1. So change inputs also to -1 to 1
  # define model
  model = Model(in_image, out_image)
  return model
```

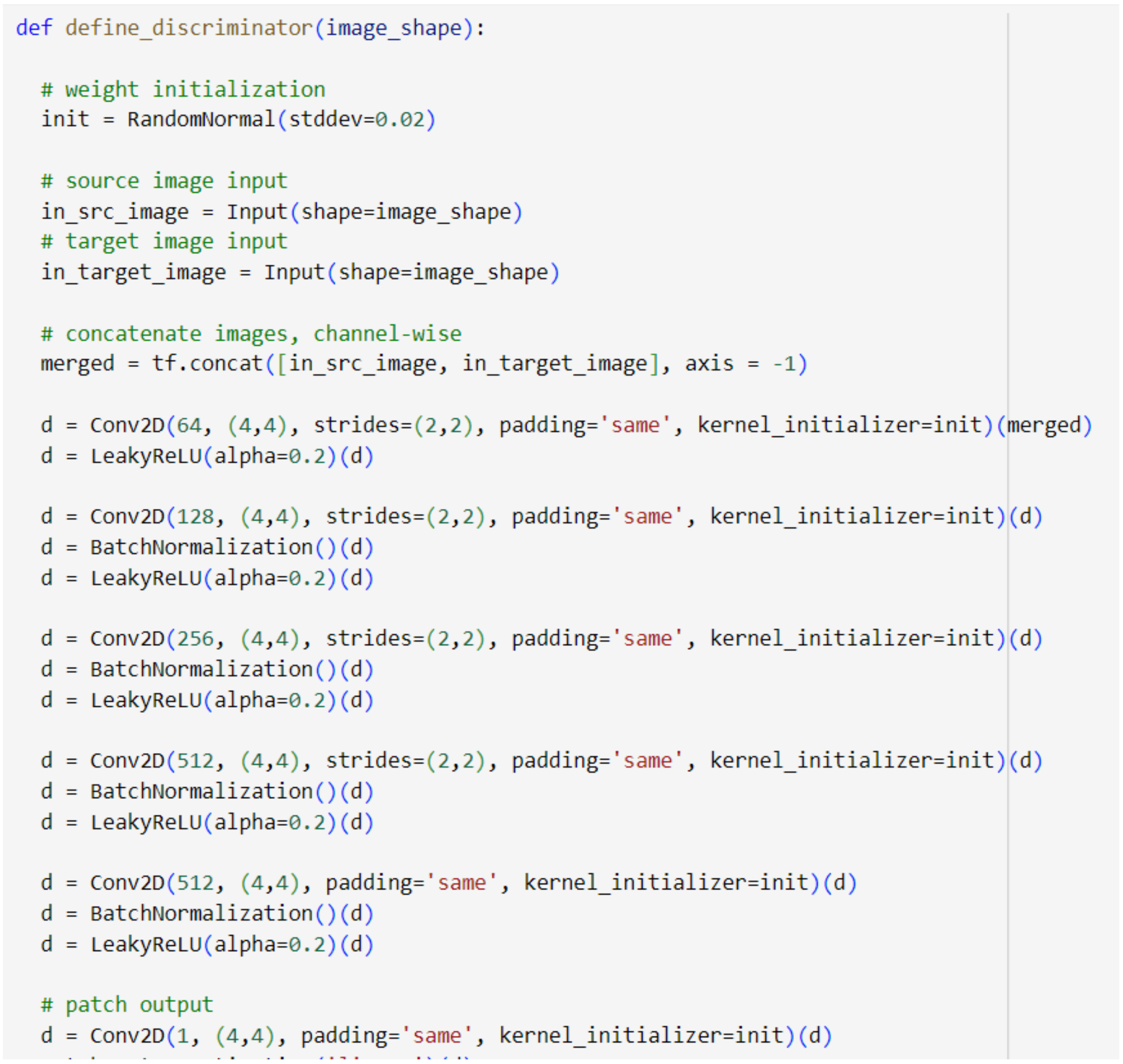
Input is a Low light image of shape (256, 256, 3) and the output shape is also (256, 256, 3). Input image should be scaled between -1 to 1, and the output is also having **tanh** activation so output image is also between -1 to 1.

Discriminator:

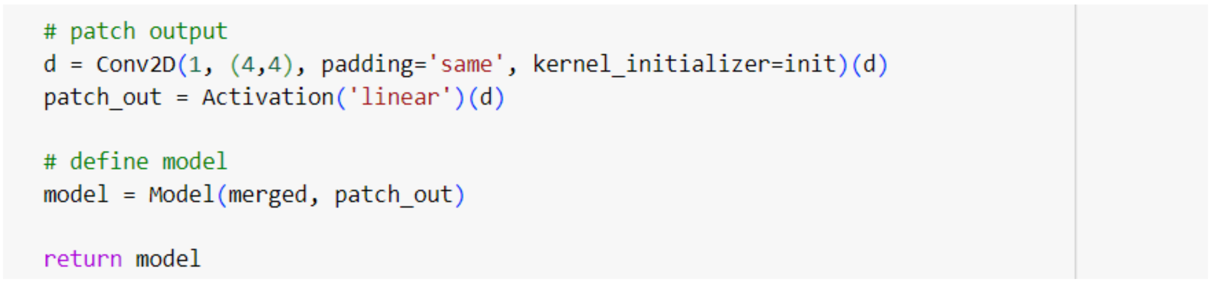
This is also called PatchGAN.

Discriminator aims to distinguish between generated samples from real ones.

The architecture consists of 4x4 convolutional layers, with strides of 2 and BatchNormalization, halving the spatial dimensions and followed by activation function LeakyReLU.



```
def define_discriminator(image_shape):

  # weight initialization
  init = RandomNormal(stddev=0.02)

  # source image input
  in_src_image = Input(shape=image_shape)
  # target image input
  in_target_image = Input(shape=image_shape)

  # concatenate images, channel-wise
  merged = tf.concat([in_src_image, in_target_image], axis = -1)

  d = Conv2D(64, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(merged)
  d = LeakyReLU(alpha=0.2)(d)

  d = Conv2D(128, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
  d = BatchNormalization()(d)
  d = LeakyReLU(alpha=0.2)(d)

  d = Conv2D(256, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
  d = BatchNormalization()(d)
  d = LeakyReLU(alpha=0.2)(d)

  d = Conv2D(512, (4,4), strides=(2,2), padding='same', kernel_initializer=init)(d)
  d = BatchNormalization()(d)
  d = LeakyReLU(alpha=0.2)(d)

  d = Conv2D(512, (4,4), padding='same', kernel_initializer=init)(d)
  d = BatchNormalization()(d)
  d = LeakyReLU(alpha=0.2)(d)

  # patch output
  d = Conv2D(1, (4,4), padding='same', kernel_initializer=init)(d)
```



```
  # patch output
  d = Conv2D(1, (4,4), padding='same', kernel_initializer=init)(d)
  patch_out = Activation('linear')(d)

  # define model
  model = Model(merged, patch_out)

  return model
```

Here also the kernel is initialised with RandomNormal(stddev(0.02)).

The input is a merged image of shape (256, 256, 6).

The job of Discriminator is to distinguish between real and fake generated samples. There are 2 kinds of input. One is a concatenation of low light image with high light image(Which discriminator should understand as Real image) and second kind of input is concatenation between low light image and generated high light image(Which discriminator should understand as fake image).

The output of Discriminator is a Patch of values of shape (16, 16) with no activation. Each value in the grid corresponds to a small patch or region of the input image. This can be understood as discriminator analysing the whole 256, 256 image in a 16,16 patch which gives the discriminator to analyse the input image with a higher focus.

Loss Calculation:

Here we have 2 models Generator and Discriminator and both have different loss calculations.

We see in pix2pix paper they used the BCE loss as Adversarial loss, but the problem with this was:

- Vanishing Gradients
- Mode collapse

While training the training loss of Generator and Discriminator were fluctuating at a certain value and training psnr was also low and was not improving which is a signal of mode collapse. Also if the loss improved and if the discriminator patch values reached near 0, 1 they would suffer gradient vanishing problems.

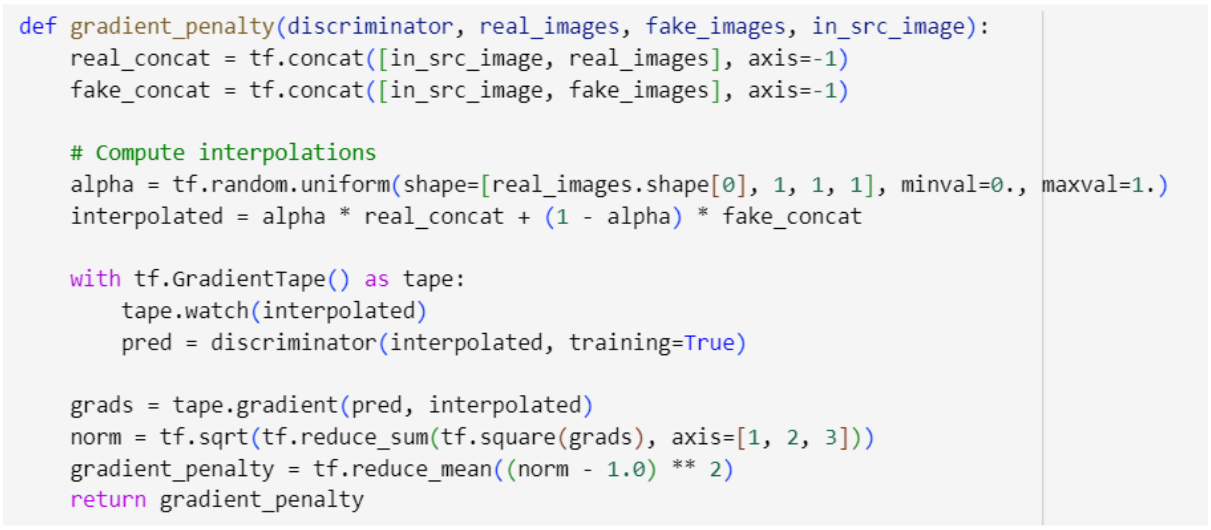
Due to these problems I chose to integrate pix2pix GAN with W loss.

The idea of using W loss instead of BCE loss came from ***Wasserstein GAN*** paper.

Wasserstein loss or the earth movers distance is the approximation of distance between two distributions. Here the two distributions are the real_output and fake_output, where real_output is the discriminator output for high light images and fake_output is the discriminator output for generated images.

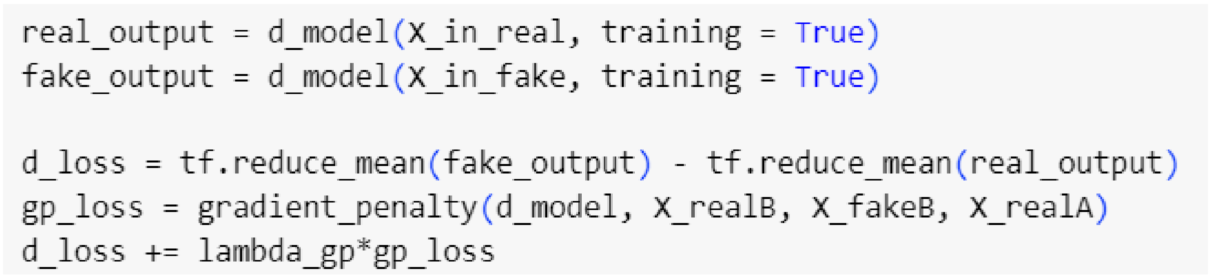
But for W loss to work properly we need to satisfy 1-Lipschitz constraint to ensure the stability and convergence of the training process. Practically it is implemented by applying gradient penalty.

Gradient Penalty: In practice, the Lipschitz constraint is often enforced using a Gradient Penalty (GP). The GP penalises the norm of the gradient of the Discriminator with respect to its input. This penalty encourages the Discriminator to adhere to the Lipschitz condition without explicitly constraining the weights of the neural network.



```python
def gradient_penalty(discriminator, real_images, fake_images, in_src_image):
    real_concat = tf.concat([in_src_image, real_images], axis=-1)
    fake_concat = tf.concat([in_src_image, fake_images], axis=-1)

    # Compute interpolations
    alpha = tf.random.uniform(shape=[real_images.shape[0], 1, 1, 1], minval=0., maxval=1.)
    interpolated = alpha * real_concat + (1 - alpha) * fake_concat

    with tf.GradientTape() as tape:
        tape.watch(interpolated)
        pred = discriminator(interpolated, training=True)

    grads = tape.gradient(pred, interpolated)
    norm = tf.sqrt(tf.reduce_sum(tf.square(grads), axis=[1, 2, 3]))
    gradient_penalty = tf.reduce_mean((norm - 1.0) ** 2)
    return gradient_penalty
```

This Gradient penalty loss is applied only on discriminator.

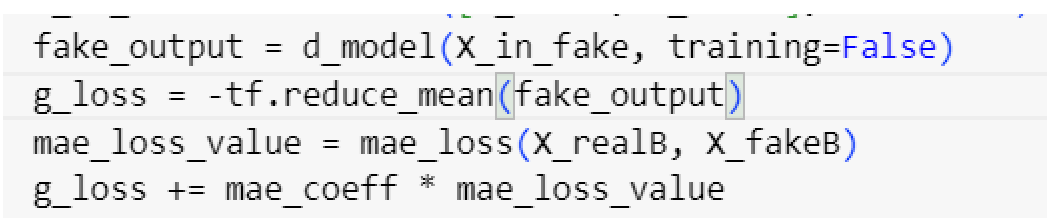


```python
real_output = d_model(X_in_real, training = True)
fake_output = d_model(X_in_fake, training = True)

d_loss = tf.reduce_mean(fake_output) - tf.reduce_mean(real_output)
gp_loss = gradient_penalty(d_model, X_realB, X_fakeB, X_realA)
d_loss += lambda_gp*gp_loss
```

The third line represents the W loss and the last line is Gradient penalty loss with a weightage of lambda_gp, typically taken as 10. As the discriminator gets trained it tries to minimise loss, increasing the distance between distributions of fake_output and real_output.

Now to have a identity match between generated images and the high light image we use MAE loss.

MAE loss: To calculate the difference between the pixel values of the generated image and the high light image. This loss helps in preserving colours in the generated image. This loss is added only with the generator along with W loss and with a weightage of 100.



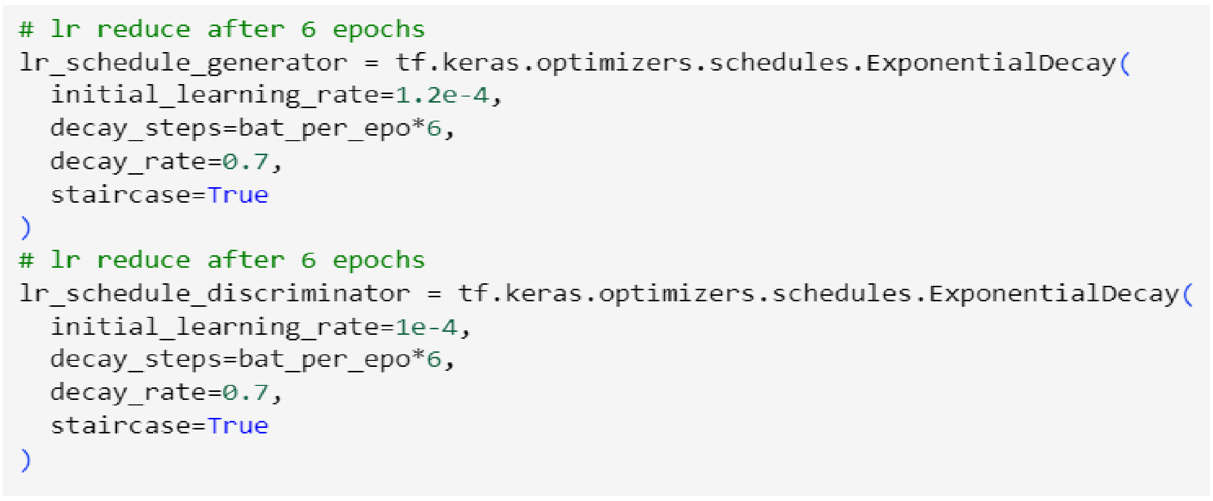
```python
fake_output = d_model(X_in_fake, training=False)
g_loss = -tf.reduce_mean(fake_output)
mae_loss_value = mae_loss(X_realB, X_fakeB)
g_loss += mae_coeff * mae_loss_value
```

The second line shows that as generator loss decreases the mean of fake_output increase catching up the real_output.

Training:

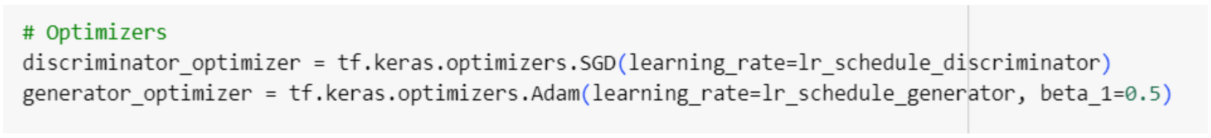
GANs are very difficult to train because the generated and discriminator should improve parallely, if any one gets too strong then training further has no meaning. So to achieve this a lot of hyperparameter optimization was required.

Learning Rates: I started with learning rates of 2e-4 for both Generator and Discriminator but then the learning was imbalanced and Discriminator was getting too strong. Also with this I had to balance batch size, loss functions, optimizers, etc..Finally when I decided to use W loss and MAE loss and by observing the pattern of training losses and psnr metrics I finally reached on **1.2e-4 for Generator** and **1e-4 for Discriminator** and also applied learning rate schedulers. The learning rate was decreased by a factor of 0.7 after every 6 epochs for both.



```
# lr reduce after 6 epochs
lr_schedule_generator = tf.keras.optimizers.schedules.ExponentialDecay(
  initial_learning_rate=1.2e-4,
  decay_steps=bat_per_epo*6,
  decay_rate=0.7,
  staircase=True
)
# lr reduce after 6 epochs
lr_schedule_discriminator = tf.keras.optimizers.schedules.ExponentialDecay(
  initial_learning_rate=1e-4,
  decay_steps=bat_per_epo*6,
  decay_rate=0.7,
  staircase=True
)
```

Optimizers: Initially the optimizer for both was Adam with beta1 = 0.5, but then because the Discriminator was getting overpowered I reduced it to **SGD optimizer for discriminator and Adam optimizer for Generator.**



```
# Optimizers
discriminator_optimizer = tf.keras.optimizers.SGD(learning_rate=lr_schedule_discriminator)
generator_optimizer = tf.keras.optimizers.Adam(learning_rate=lr_schedule_generator, beta_1=0.5)
```

Clip norm: The gradients of Generator were overshooting, and this was causing unstable training of the model, then I tried clipping the norm to 1 and then 2 both worked good, but **clipping norm at 1** was better.

Batch Size: Typically for pix2pix GAN batch size is 1, but for my dataset with W loss it was not working fine then I changed it to **4**, then the training went smoothly.

n_critics: In WGAN paper discriminator is referred to as critic, so this name is used. It is the number of times training discriminator for a single training of generator. Initially I was using BCE loss instead of W loss in which the Discriminator was getting overpowered but by introducing W loss Discriminator was very weak and a single training of discriminator per training of generator was not enough, then by trying different values of n_critic from 1 to 5, the best came out a **n_critic = 5.**

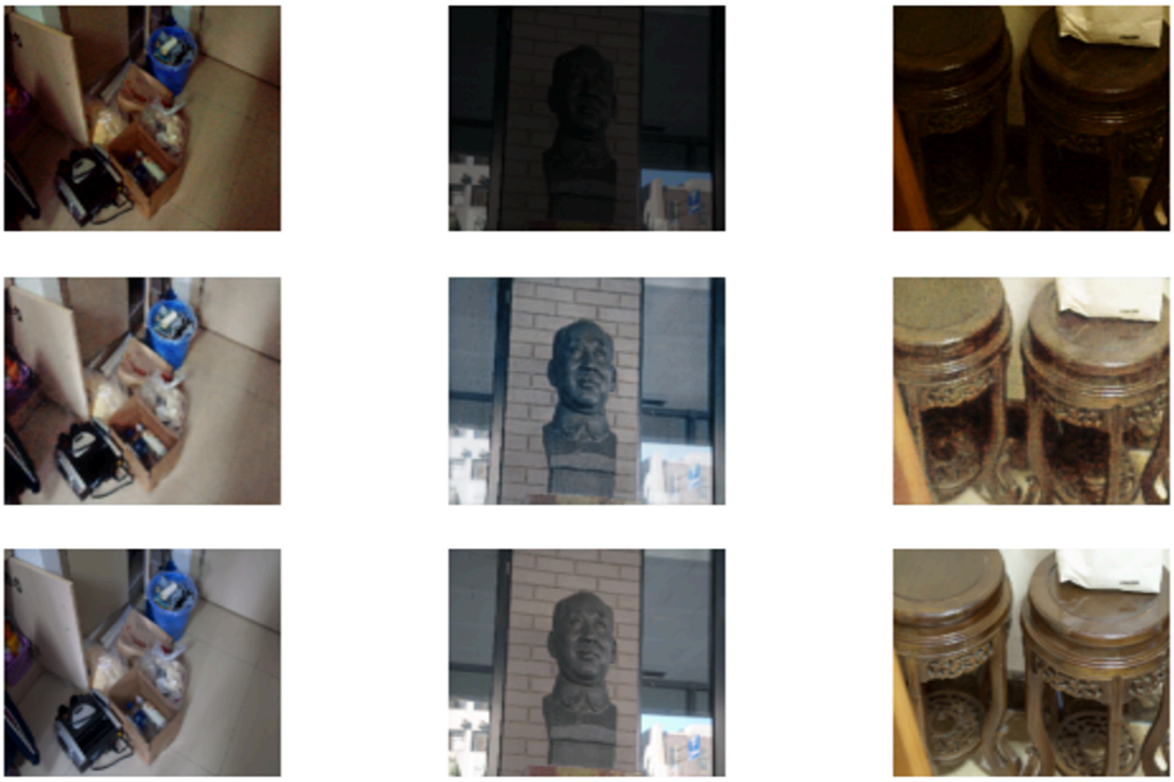
After applying all of these combinations, best combination came out to be with

Learning rate scheduler with initial lr of Discriminator 1e-4 and Generator 1.2e-4 with decreasing factor of 0.7 after every 6 epochs. Adam optimizer for Generator with beta1 = 0.5 and SGD optimizer for Discriminator. Clipping gradients at 1 for both and n_critic = 5 with a batch size of 4.

Results:

I trained on 1552 training images for 34 Epochs.

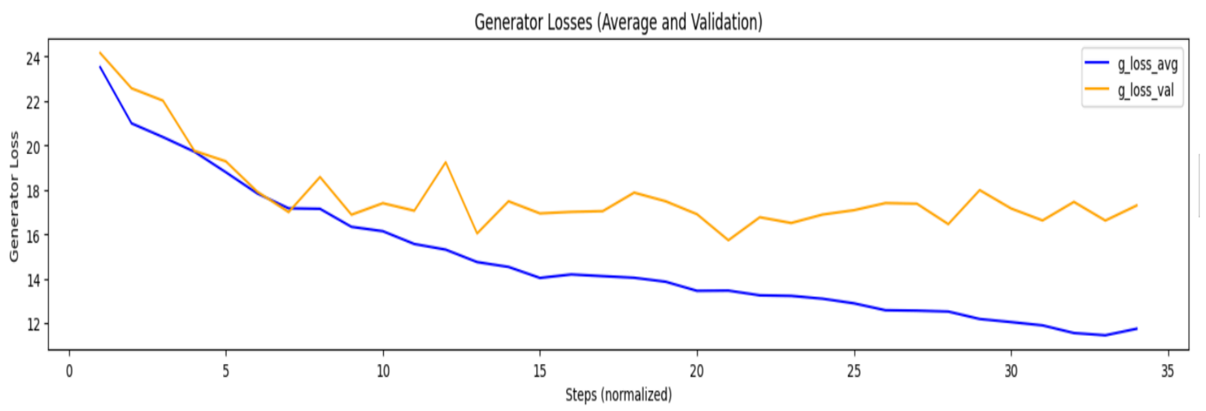
- Reached maximum training **PSNR of 23.29** on 33nd Epoch
- Reached maximum Validation **PSNR of 21.4** on 21st Epoch
- Training **PSNR of 22.15** on 21st Epoch



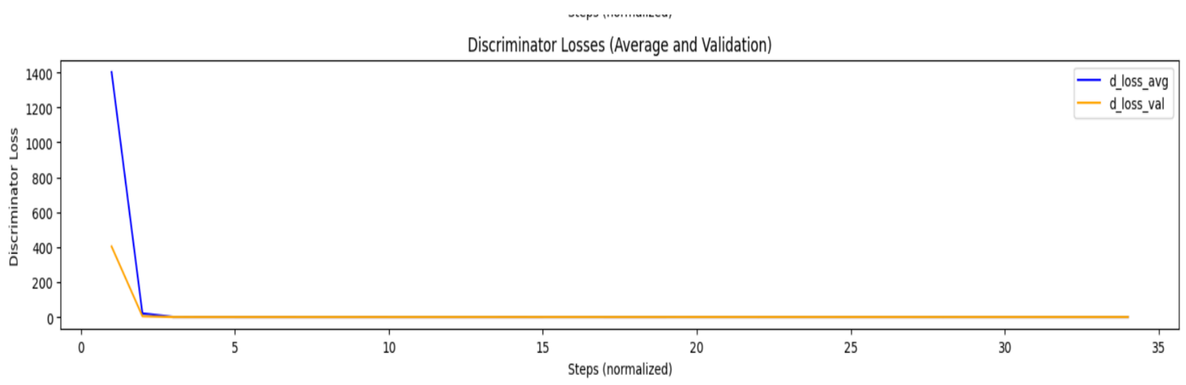
- First one is low light image
- Second one is the Generated Image
- Third is the corresponding High light image

Evaluation:

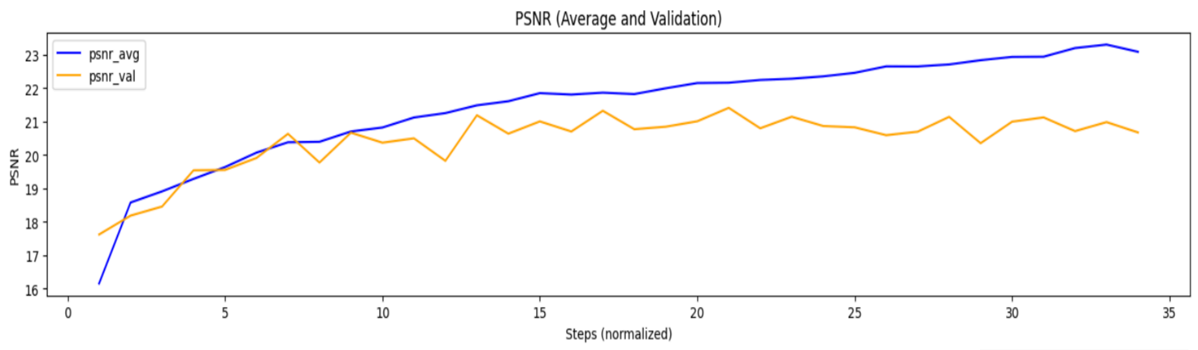
- Generator loss(Training and Validation)



- Discriminator loss(Training and Validation)



- PSNR(Training and Validation)



Generator loss is decreasing, Discriminator loss reached a constant value and PSNR is improving, this shows Stable GAN training. However at the end epochs Generator loss and the PSNR graphs seems to show overfitting.

Conclusion:

The model was trained for 34 Epochs, and the best PSNR value for validation was found on 21st Epoch with a validation PSNR of 21.24