INSTITUTO TECNOLÓGICO
DE
BUENOS AIRES

Ingeniería Electrónica

(22.45) - Redes Neuronales

TRABAJO PRÁCTICO FINAL

Colorización de imagenes mediante la implementación de modelo Pix2Pix

---

Alumno:
Díaz, Ian Cruz

Legajo:
57515

Contenido

1. Introducción

A lo largo de los años, la tecnología fue avanzando para resolver cada vez problemas mas avanzados. Ciertos avances tecnológicos permitieron al ser humano adaptarse a las condiciones del ambiente y mejorar sus capacidades frente a otras especies. Una de las virtudes que tenemos los humanos, es la capacidad de adaptarnos y generar nuevas herramientas a partir de herramientas preexistentes, generando así una mejora continua y resultando en tecnologías cada vez mas avanzadas. Para esto, resulta vital el hecho de poder ver hacia el pasado, y observar con la mayor definición posible detalles del pasado. Para esto, desde siempre existieron fotografías o vídeos, sin embargo, cuanto mas hacia el pasado se mire, menos definición en general obtendremos, y en cierto punto de nuestra historia, resulta ya muy difícil encontrar fotografías o vídeos tomados en color, ya que en el pasado predominaba el blanco y negro para plasmar recuerdos. Es por esto, que este trabajo se propone superar la barrera del blanco y negro, generando una herramienta que permita, mediante *CNNs* (Redes Neuronales Convolucionales por sus siglas en ingles), transformar imagenes en ByN a color.

2. Modelo

Para poder resolver este problema se utilizará un modelo comúnmente denominado *Pix2Pix*. El analisis de este modelo se realizará a continuación.

2.1. Pix2Pix

El modelo *Pix2Pix* [6], es un tipo de modelo de *Deep Learning* que se basa en las arquitecturas *GAN* (Generative Adversarial Network) [5][8]. Estas arquitecturas funcionan mediante la implementación de 2 redes neuronales, un a **Red Generadora**, que deberá a prender a resolver la tarea de generar el contenido que nosotros queramos, y la **Red Discriminadora**, y su tarea será la de intentar discriminar a si una imagen de entrada a la red fue una imagen original de un dataset, o si fue una imagen generada por la red generadora. De esta manera, el sistema se convierte en adversario de la manera en que las redes van compitiendo entre sí y de esta manera, la red generadora aprende a crear imágenes lo suficientemente realistas como para que un humano no pueda distinguirlas del dataset original.
La Red generadora como se dijo es una red que dada una entrada aleatoria, deberá ser capaz de generar imágenes realistas, esto se hace mediante el entrenamiento de una red deconvolucional. Estas redes tiene la característica de que dada una entrada acotada, puede ir generando valores de manera tal que genere cada unos de los valores de los píxeles de una imagen. Es como si fuera (en oposición a las redes convolucionales que comprimen la información) generando información a partir de unos valores que contienen la imagen 'comprimida'.
Sin embargo, como se dijo, esta red generadora tiene un problema, la salida de la red generadora sabemos que depende 100 % de la entrada obviamente, y esta entrada es aleatoria, esto quiere decir que siempre para una misma entrada, la salida será la misma, pero estas entradas son completamente aleatorias, como ruido blanco, entonces ¿Como podemos hacer para generar una salida deseada si no podemos saber a priori el valor de entrada necesario? Para resolver este problema surgen las **cGAN** (Conditional Generative Adversarial Networks), y la idea de estas redes son generar un vector de *etiquetas* que especifique el tipo de salida que se quiere, combinarlo de alguna manera con el vector de entrada aleatorio (ya sea concatenándolo, sumándolo, etc.), y entrenar a la red con estas etiquetas. De esta manera, cuando la red ya se encuentre entrenada, uno solo deberá introducir a la entrada un vector de etiquetas deseado y un vector aleatorio, y la red será capaz de generar una salida adecuada a lo que nosotros querríamos[1][8].
De esta manera se puede extrapolar el problema y en lugar de generar una imagen a partir de unas etiquetas acotadas, se podria pensar al vector de etiquetas como

otra imagen, es decir generar una imagen a partir de otra imagen, lo que convierte la arquitectura en un **Encoder-Decoder** o **AutoEncoder**, que se basa en una red neuronal convolucional seguida de una red neuronal deconvolucional. De ahí viene el nombre de *Pix2Pix* es decir, de imagen a imagen.
Sin embargo, falta un detalle, como nos encontramos frente a una arquitectura del tipo reloj de arena (Encoder-Decoder), la información de la imagen original debe pasar por un cuello de botella para llegar al otro lado de la red, y en muchos casos esto es suficiente, sin embargo, en nuestro caso, esto no resulta suficiente, por esto se agrega las llamadas *Skip Connections*[4] [7] [8], estas conexiones entre capas que no son adyacentes permite que la información necesaria sea capaz de atravesar la red sin el inconveniente de perder la información que fue comprimida para pasar por el cuello de botella. Un tipo de red famosa por tener estas conexiones es la **UNet** como se puede ver en la Figura 2.1.

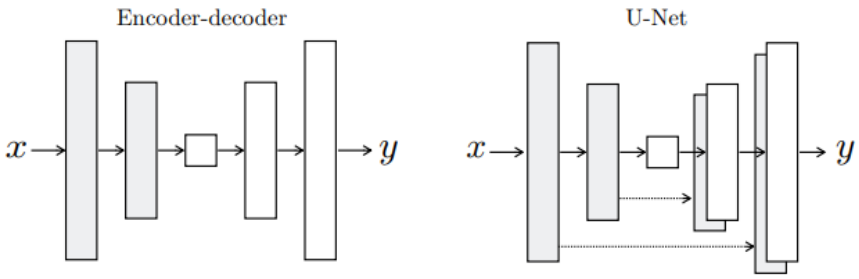


FIGURA 2.1: Redes Autoencoder

2.2. Creación del modelo

El modelo, por consecuencia de lo mencionado anteriormente, se lo definió como se menciona en el apéndice del paper [6] [9]. Para eso se crearon varias funciones de **Tensorflow** con el fin de lograr el modelo deseado.

```
def downsample(filters, batch_norm=True):

  result = Sequential()
  # Inicializamos las variables del modelo con valores de una variable
    aleatoria gaussiana con media 0 y varianza 0.02
  initializer = tf.random_normal_initializer(0, 0.02)

  #Conv
  result.add(Conv2D(filters=filters,strides=2, kernel_size=4, padding=
   'same',
```

```
                    kernel_initializer=initializer, use_bias=not
    batch_norm))
  #Batch
  if batch_norm:
    result.add(BatchNormalization())
  #Activation LeakyRelu
  result.add(LeakyReLU())
  return result
```

Primero se crea la función **downsample** que implementa una capa de convolución de *stride* = 2, es decir, que reduce la imagen a la mitad del tamaño del inicial, y recibe una cantidad de filtros como parámetro, además se puede notar que la activación no es una *ReLU* común y corriente, sino que es una **LeakyReLU**, esta función se diferencia de la ReLU en que en los valores negativos, su derivada no es cero, esto se puede ver en la Figura 2.2.

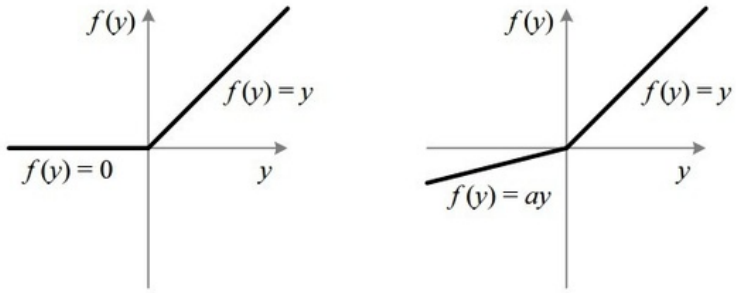


FIGURA 2.2: Comparación entre la activación ReLU y LeakyReLU

A si mismo, se genera la función **upsample** que es análoga a la función *downsample* pero en lugar de utilizarla como reducción de información para el encoder, se le utiliza como producción de información para el decoder. Esto se puede ver en la pórcion de código a continuación.

```
def upsample(filters, dropout=True):
  result = Sequential()
  initializer = tf.random_normal_initializer(0, 0.02)

  #Conv
  result.add(Conv2DTranspose(filters=filters,strides=2, kernel_size=4,
    padding='same',
                             kernel_initializer=initializer, use_bias=
    False))
  #Batch
  if dropout:
    result.add(Dropout(0.5))
  #Activation LeakyRelu
```

```
  result.add(ReLU())
  return result
```

Podemos observar además que a la capa de upsample también se le agrega una cantidad de dropout del 50 % que vino definida por el paper[6] [9].

2.2.1. Generador

Consecuentemente, la red neuronal del generador ser la genera de la siguiente manera:

```
def Generator():

  initializer = tf.random_normal_initializer(0, 0.02)

  inputs = Input(shape=[None, None, 3]) # (b, 256, 256, 64)

  down_stack = [
      downsample(64, batch_norm=False), # (b, 128, 128, 64)
      downsample(128), # (b, 64, 64, 128)
      downsample(256), # (b, 32, 32, 256)
      downsample(512), # (b, 16, 16, 512)
      downsample(512), # (b, 8, 8, 512)
      downsample(512), # (b, 4, 4, 512)
      downsample(512), # (b, 2, 2, 512)
      downsample(512)  # (b, 1, 1, 512)
  ]

  up_stack = [
      upsample(512), # (b, 2, 2, 1024)
      upsample(512), # (b, 4, 4, 1024)
      upsample(512), # (b, 8, 8, 1024)
      upsample(512, dropout=False), # (b, 16, 16, 1024)
      upsample(256, dropout=False), # (b, 32, 32, 512)
      upsample(128, dropout=False), # (b, 64, 64, 256)
      upsample(64, dropout=False), # (b, 128, 128, 128)

  ]

  last = Conv2DTranspose(filters=3, kernel_size=4, strides=2, padding=
   "same", kernel_initializer=initializer,
                         activation='tanh')

  x = inputs
  s = []
  concat = Concatenate()
  for enc in down_stack:
```

```
36      x = enc(x)
37      s.append(x)
38    s = reversed(s[:-1])
39
40    for dec, sk in zip(up_stack, s):
41      x = dec(x)
42      x = concat([x, sk])
43
44    output = last(x)
45
46    return Model(inputs=inputs, outputs=output)
```

Como se puede ver, el generador cumple con las especificaciones mencionadas en el paper, es decir, una estructura de **Autoencoder** que tiene 8 capas convolucionales y 8 capas deconvolucionales generadas con las funciones **downsample** y **upsample**, y a su vez la ultima capa tiene una activación *tanh*. Finalmente en los ciclos *for* se concatenan las capas generadas y se realiza la estructura tipo **UNet** mediante la concatenación de x con sk (siendo sk las skip connections) en la linea 44, esto se puede ver en la Figura 2.3.

2.2.2. Discriminador

Para el caso del discriminador la estructura es mas simple, debido a que no se trata de un *Autoencoder*, si bien para el discriminador se podría utilizar una red convolucional que comprima la imagen hasta un valor que diga si es la imagen es falsa o no, este paper para el Pix2Pix utiliza una red denominada **PatchGAN** [3], que en lugar de llegar a un valor final que decida si la imagen parece o no real, simplemente genera una especie de mapa de segmentación donde separa las areas de la imagen que parecen reales de las que no. Esta implementación se puede ver a continuación.

```
def Discriminator():
  real_input = Input(shape=[None, None, 3], name="real_image")
  fake_input = Input(shape=[None, None, 3], name="fake_image")

  con = concatenate([real_input, fake_input])

  initializer = tf.random_normal_initializer(0, 0.02)

  dec1 = downsample(64, batch_norm=False)(con)
  dec2 = downsample(128)(dec1)
  dec3 = downsample(128)(dec2)
  dec4 = downsample(128)(dec3)

```

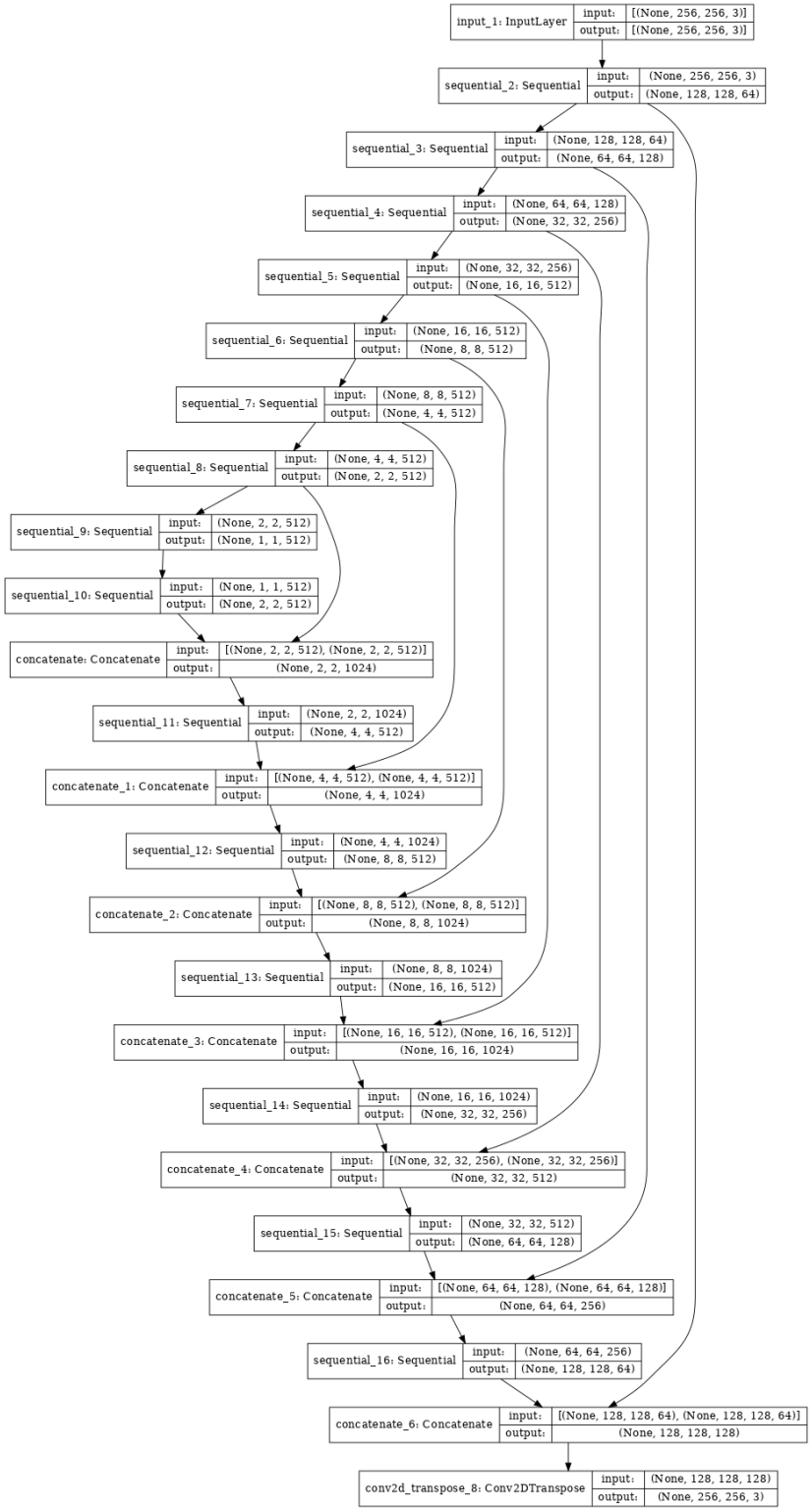


FIGURA 2.3: Estructura del generador

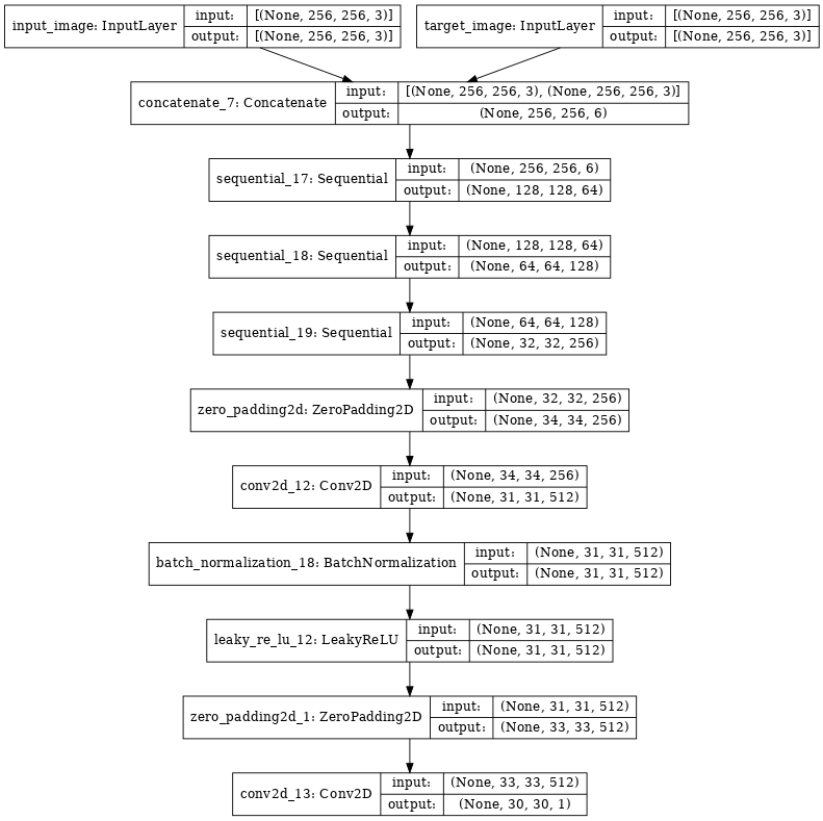


FIGURA 2.4: Estructura del Discriminador

```
14   output = Conv2D(filters=1, kernel_size=4, strides=1,
      kernel_initializer=initializer, padding='same')(dec4)
15   return Model(inputs=[real_input, fake_input], outputs=output)
```

Este sistema de discriminador y su estructura se puede ver en la Figura 2.4.
De esta manera, ya tenemos definidos nuestros modelos y están listos para ser utilizados y entrenados.

2.3. Losses

Para poder entrenar al modelo, simplemente falta definir las funciones de costo a utilizar y a optimizar para que nuestro modelo aprenda, para esto se definen funciones de costo diferentes para el generador y para el discriminador. Para ambas funciones, se utiliza la funcion **BinaryCrossEntropy** (Entropía cruzada binaria) de la librería *Tensorflow*.
Para el discriminador, se define la función de costo como se ve a continuación:

```
1  def discrimator_loss(disc_real_output, disc_generated_output):
2
3    #Diferencia entre la imagen real y la detectada por el discriminador
4    real_loss = loss_object(tf.ones_like(disc_real_output),
      disc_real_output)
```

```
  #Diferencia entre la imagen generada y la detectada por el
   discriminador
  fake_loss = loss_object(tf.zeros_like(disc_generated_output),
   disc_generated_output)

  total_disc_loss = real_loss + fake_loss

  return total_disc_loss
```

Y para el generador, de la misma manera, la funcion de costo queda definida como:

```
LAMBDA = 100

def generator_loss(disc_generated_output, gen_output, target):
  gan_loss = loss_object(tf.ones_like(disc_generated_output),
   disc_generated_output)

  #Error absoluto
  l1_loss = tf.reduce_mean(tf.abs(target - gen_output))

  total_gen_loss =  gan_loss +(LAMBDA * l1_loss)

  return total_gen_loss
```

2.4. Entrenamiento

Para el entrenamiento se definen las funciones necesarias para que nuestra red aprenda. Para esto, primero se define el **train_step** que sera la función que dada una imagen de entrada y una imagen objetivo, realiza el camino por ambas redes y calcule las *losses*, de tal manera que el sistema en cada step vaya aprendiendo. Estas secuencias de entrenamiento para el generador y discriminador se pueden ver en la Figura 2.5 y el código que la implementa se ve a continuación.

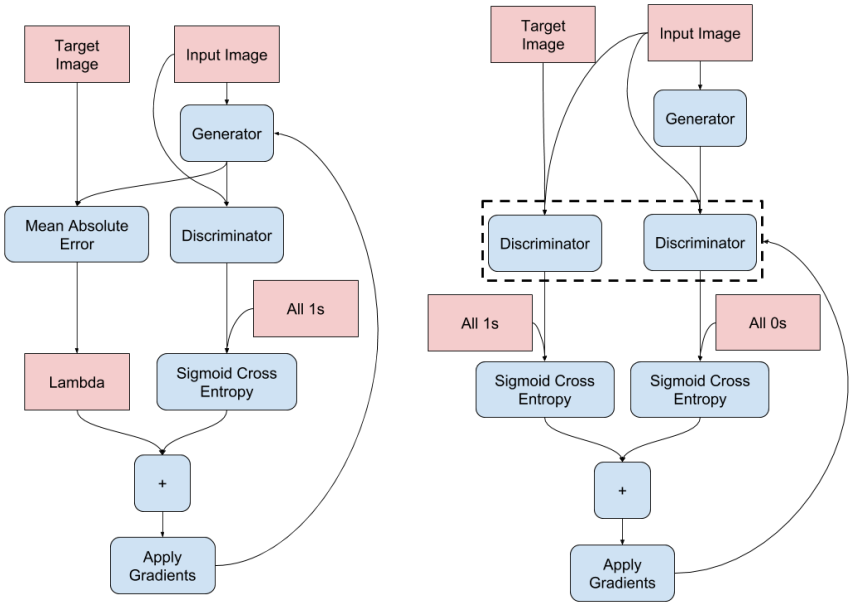
```
def train_step(input_image, target):

  with tf.GradientTape() as gen_tape, tf.GradientTape() as discr_tape:

    output_image = generator(input_image, training=True) # Genero la
   imagen con el generador

    output_gen_discr = discriminator([output_image, input_image],
   training=True) # Ingreso con ambas imagenes (generador y original)
    al discriminador

```



(A) Entrenamiento del Generador (B) Entrenamiento del Discriminador

FIGURA 2.5: Secuencias de entrenamiento para las diferentes redes

```
    output_target_discr = discriminator([target, input_image],
    training=True) # Ingreso con ambas imagenes (objetivo y original)
    al discriminador

    discr_loss = discrimator_loss(output_target_discr,
    output_gen_discr)  #Calculo las losses
    gen_loss = generator_loss(output_gen_discr,output_image, target)

    generator_grads = gen_tape.gradient(gen_loss, generator.
    trainable_variables)  #Calculo los gradientes
    discriminator_grads = discr_tape.gradient(discr_loss,
    discriminator.trainable_variables)

    generator_optimizer.apply_gradients(zip(generator_grads, generator
    .trainable_variables))  #Optimizo los modelos
    discriminator_optimizer.apply_gradients(zip(discriminator_grads,
    discriminator.trainable_variables))
```

Finalmente, se realiza un for por cada trainstep para cada imagen del dataset y para cada epoca del sistema.

```
def train(dataset, epochs):
  for epoch in range(epochs):
    imgi = 0
    for input_image, target in dataset:
      imgi += 1
```

```
6       print ('epoch ' + str(epoch) + ' - train: ' + str(imgi) + '/' +
    str(len(train_urls)))
7       train_step (input_image, target)
8       clear_output(wait=True)
9
10
11    imgi = 0
12    for inp, tar in test_dataset.take(1):
13      generate_images(generator, inp, tar, str(imgi) + '_' + str(epoch
    ), display_imgs=True)
14      imgi +=1
15
16    ##Saving
17    if (epoch + 1) % 1 == 0:
18      checkpoint.save(file_prefix=checkpoint_prefix)
```

3. Dataset

Para resolver el problema se utilizo el dataset Art Images[2] conteniendo 2032 imágenes en Blanco y Negro, y su respectiva imagen a color, y para poder entrenar el sistema se implemento un sistema de Data augmentation. En las Figuras 3.1 y 3.2 se pueden ver algunos ejemplos de las imágenes del dataset.

FIGURA 3.1: Imagen de pareja en el dataset

FIGURA 3.2: Imagen de Persona en el dataset

3.1. Data Augmentation

Para el *Data Augmentation* se realiza una serie de algoritmos a las imágenes de entrada de tal manera de generar imágenes lo suficientemente diferentes a las originales como para que el modelo no haga un overfitting, pero sin perder el 'sentido' de las imágenes, es decir que cada imagen sea entendible[6].
Para esto primero a las imágenes de entrada se les realiza un resize interpolando para que las imágenes queden mas grandes, esto se realiza con la porción de código que se ve a continuación:

```
1 def resize(input_img, tar_img, img_size):
```

```
    input_img = tf.image.resize(input_img, [img_size, img_size])
    tar_img = tf.image.resize(tar_img, [img_size, img_size])

    return input_img, tar_img
```

Una vez realizado el resize de la funcion, se procede a normalizar los valores de colores de las imágenes entre -1 y 1.

```
def normalize(input_img, tar_img):
    input_img = (input_img/255.) - 1
    tar_img = (tar_img/255.) - 1
    return input_img, tar_img
```

Finalmente, se recorta la imagen con un desplazamiento random obteniendo el tamaño de imagen deseado, y se la espeja (o no) de acuerdo al resultado de una variable aleatoria obtenida de tensorflow.

```
def random_jitter(input_img, tar_img):
    input_img, tar_img = resize(input_img, tar_img, 572)

    #Apila las imagenes
    stacked_image = tf.stack([input_img, tar_img], axis=0)
    #Recorta
    cropped_image = tf.image.random_crop(stacked_image, size=[2,
   img_size, img_size, 3])

    input_img, tar_img = cropped_image[0], cropped_image[1]
    if tf.random.uniform(()) > 0.5:
        input_img = tf.image.flip_left_right(input_img)
        tar_img = tf.image.flip_left_right(tar_img)
    return input_img, tar_img
```

4. Resultados

Para poder observar los resultados obtenidos, realizamos la prueba del sistema en diferentes épocas del entrenamiento y observando los resultados sobre una imagen con la que el sistema no fue entrenado, y los resultados son los que se pueden observar en la Figura 4.1.
Como se puede observar en la Figura, los mejores resultados para esa imagen se comienzan a obtener alrededor de la época 10, por lo tanto a partir de ese momento es posible que el sistema haya entrado en overfitting, por lo que no se ven mejoras sustanciales a partir de ese momento. Sin embargo, dado que se le dio una imagen antigua completamente en blanco y negro, el resultado fue satisfactorio debido a que las imágenes generadas tienen un nivel de color aceptable para el tamaño del dataset utilizado. En la Figura 4.2 se pueden ver los resultados de la aplicación del modelo a datos del test set en la epoch 25[1]. Finalmente, para una segunda implementación de este proyecto se podrían aplicar varias mejoras que posiblemente obtendrían mejores resultados, estas mejoras pueden ser:

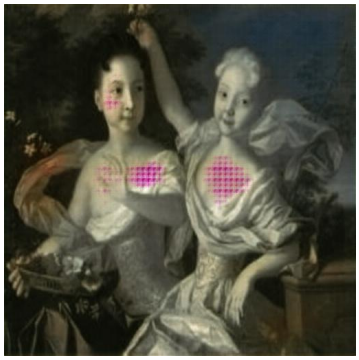
- Utilizar **Pix2PixHD**[2] [10]
- Entrenamiento con un Dataset de paisajes[3]

Sin embargo, el resultado obtenido se encuentra dentro de los parámetros aceptables, y listo para ser implementado en otras aplicaciones.

[1]Las 25 épocas tuvieron un tiempo aproximado de 3 horas con la GPU de Google Colab.
[2]Una versión mejorada de Pix2Pix implementada por NVIDIA capaz de realizar operaciones sobre imágenes en alta definición.
[3]Ya que a la hora de colorizar paisajes en pinturas antiguas el resultado generado no es aceptable.

(A) Imagen de entrada

(B) Imagen objetivo



(C) Epoch = 1

(D) Epoch = 5

(E) Epoch = 10

(F) Epoch = 15

(G) Epoch = 20

(H) Epoch = 25

FIGURA 4.1: Entrenamiento en diferentes épocas

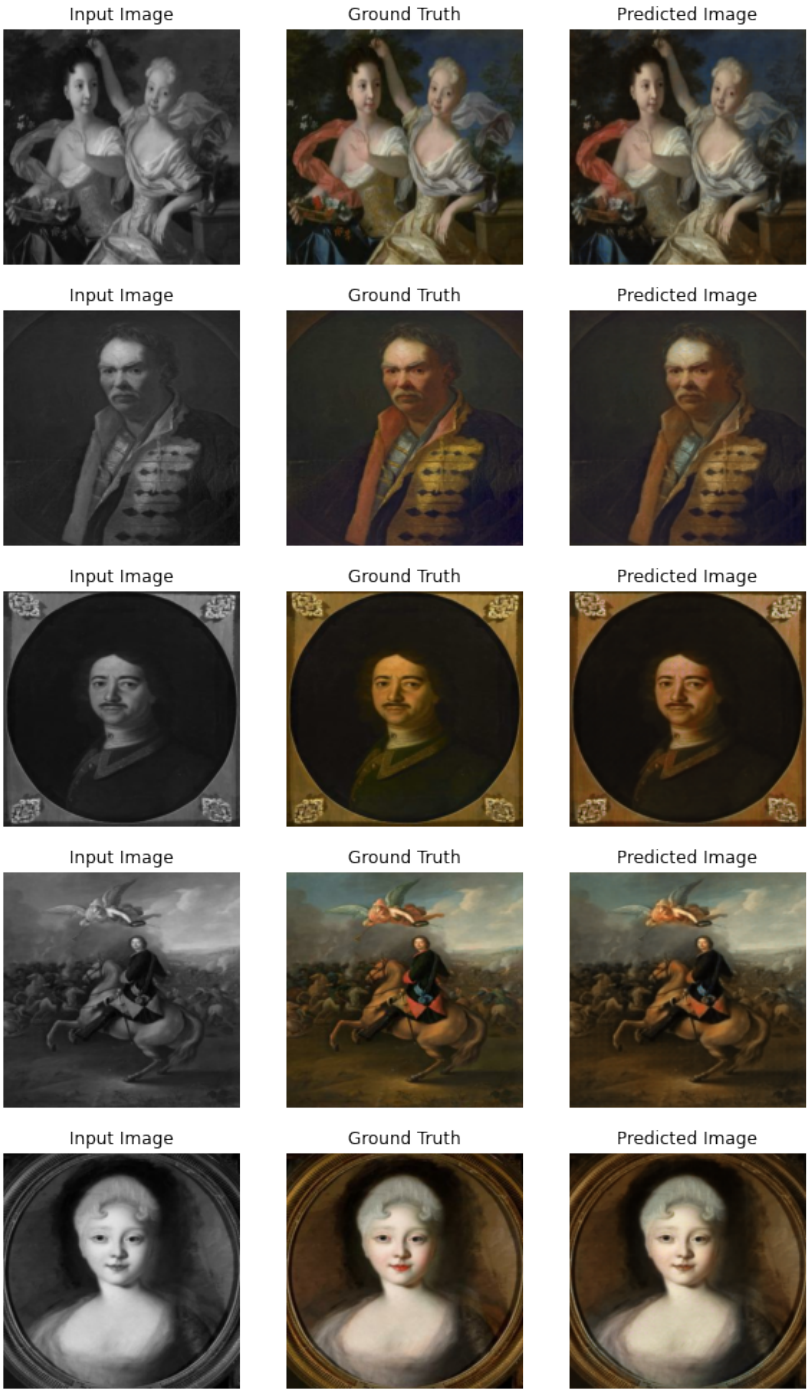


FIGURA 4.2: Resultados del test set en la epoch 25

Referencias

[1] Andrew Brock, Jeff Donahue y Karen Simonyan. "Large Scale GAN Training For High Fidelity Natural Image Synthesis". En: (feb. de 2019).

[2] Danil. *Art Images: Drawing/Painting/Sculptures/Engravings*. 2018. URL: https://www.kaggle.com/thedownhill/art-images-drawings-painting-sculpture-engraving.

[3] Ugur Demi y Gozde Unal. "Patch-Based Image Inpainting with Generative Adversarial Networks". En: (mar. de 2018).

[4] Ian Goodfellow, Yoshua Bengio y Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[5] Ian Goodfellow y col. "Generative Adversarial Nets". En: (jun. de 2014).

[6] Phillip Isola y col. *Image-to-Image Translation with Conditional Adversarial Networks*. Nov. de 2018.

[7] Michael Nielsen. *Neural Networks and Deep Learning*. neuralnetworksanddeeplearning.com. Determination Press, 2015.

[8] Carlos Santana Vega. *DotCSV DeepNude*. https://www.youtube.com/watch?v=ysEjAqnHp64&t=1169s. 2019.

[9] Carlos Santana Vega. *DotCSV Pix2Pix*. https://www.youtube.com/watch?v=YsrMGcgfETY&t=4506s. 2019.

[10] Ting-Chun Wang y col. "High-Resolution Image Synthesis and Semantic Manipulation with Conditional GANs". En: (ago. de 2018).