Redes Neuronales

Moreno Santiago José Miguel.

Transferencia de estilo.

1. Introducción.

La **Transferencia Neuronal de Estilo** (Neural Style Transfer, NST), que **consiste en el aprendizaje, por parte de una red neuronal, del estilo de una fotografía o una obra de arte determinada** (por ejemplo "Noche Estrellada", de Van Gogh) **y la transferencia del mismo a otra imagen de entrada diferente.**

1.1. Espacio de característica.

Llámese espacio de características o espacio de Fukushima **a las diferentes posibilidades de configuración de una capa convolucional**. Se trata de un hiperespacio dotado de tantas dimensiones como neuronas haya en la capa. Cada neurona individual se identifica por tres números: el mapa al que pertenece (z) y la posición (x,y) en la que se ubica dentro del mismo. Cada vector viene conformado por los valores de activación de las neuronas de esa capa.

Los espacios más cercanos a la entrada codifican rasgos de menor nivel (trazos, figuras geométricas), mientras que **las capas más profundas almacenan los conceptos de nivel más abstracto** (árboles, automóviles, etc..). El conjunto de todos los espacios de características de la red constituye un ámbito imaginario simbólico-semántico capaz de almacenar gran cantidad de patrones, texturas, ideas, arquetipos y categorías.

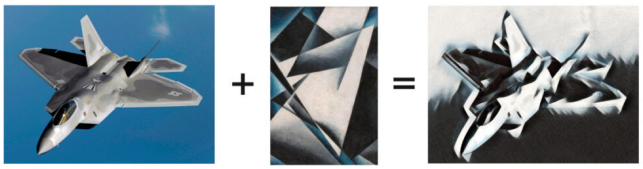
En dichos espacios pueden realizarse operaciones matemáticas que determinan la cercanía semántica entre dos estados de red determinados (vectores en el espacio de Fukushima). De particular interés son, en el ámbito de Transferencia Neuronal de Estilo, los conceptos de distancia euclídea entre dos vectores y matriz de Gram de una capa convolucional. Ambos fueron utilizados por tres investigadores de la Universidad de Tubinga para crear un ingenioso algoritmo capaz de codificar no sólo el contenido semántico de una imagen determinada, sino también su estilo.

1.2. Codificando el estilo.

Los autores de este nuevo trabajo (Gatys et al) **emplearon una red convolucional VGG-16 (una versión modificada de AlexNet) entrenada para el reconocimiento de objetos**. En VGG-16, las capas convolucionales se agrupan en cinco bloques, en cada uno de los cuales hay el mismo número de mapas de características, y todos los cuales son de la misma resolución. Esta sección convolucional es seguida por dos capas densamente conectadas, tras las cuales se sitúa la capa de salida, cuya función de activación es softmax, como es usual en las redes clasificadoras. La red se entrena previamente para reconocer las 1000 categorías diferentes del banco de imágenes ImageNet. Y de este modo se fijan los valores de los pesos sinápticos de los kernels de las convoluciones.

En la transferencia neuronal de estilo, VGG-16, ya entrenada, es usada para sintetizar una nueva imagen a partir de una fotografía y una obra de arte, cuyo estilo se transfiere a la primera. La imagen se crea aplicando el procedimiento de descenso del gradiente de la función de pérdida, actualizando tras cada iteración los valores de activación de la capa de entrada, y no los pesos sinápticos de la red, que permanecen invariables. Lo que se retropropaga no es el gradiente de la función de pérdida respecto a los pesos sinápticos (como sucede en el entrenamiento), sino respecto a los valores de entrada de cada neurona. La síntesis defini-

tiva tiene lugar progresivamente tras centenares o miles de iteraciones.



2. Procedimiento para la Transferencia Neuronal de Estilo.

2.1. La función de pérdida.

El procedimiento de Gatys et al es, en líneas generales, el siguiente:

La función de pérdida $L(c, s, m)$ (que debe ser minimizada) **se define para cada capa convolucional** y está compuesta de dos términos. El primero de ellos se relaciona con el contenido de la imagen a sintetizar y el segundo de ellos con su estilo:

$$L(c, s, m) = \lambda_c \cdot L_{\text{contenido}}(c, m) + \lambda_s \cdot L_{\text{estilo}}(s, m) \quad (1)$$

Donde $L(c, s, m)$ es la función de pérdida total, $L_{\text{contenido}}$ la parte de la función relacionada con el contenido semántico de la imagen y L_{estilo} la relacionada con su estilo. λ_c y λ_s son hiperparámetros que señalan el peso de cada una de estas dos partes, mientras que c, s y m representan respectivamente la fotografía o contenido, la obra de arte cuyo estilo se adopta y la imagen en proceso de síntesis. Inicialmente, los píxeles de la imagen m se determinan de un modo aleatorio, a partir de ruido.

2.2. Función de pérdida del contenido.

La parte de la función de pérdida relacionada con el contenido es equivalente a la distancia euclídea al cuadrado entre dos vectores. El primer vector es el correspondiente a las activaciones de la capa convolucional de que se trate generadas por la fotografía. El segundo tiene como componentes las activaciones que la imagen en proceso de síntesis genera en esa misma capa:

$$L_{\text{contenido}} = \frac{1}{2} \sum_{(x,y,z)} (o_{x,y,z} - n_{x,y,z})^2 \quad (2)$$

donde el sumatorio se realiza a través de todas las neuronas (x, y, z) de un mismo mapa de características, siendo $o_{x,y,z}$ la activación de la célula causada por la imagen (m) que se está generando (recordemos que inicialmente estos valores son aleatorios) y $n_{x,y,z}$ la activación generada por la fotografía (c). La elección de este término asegura que, con el transcurso de las iteraciones, las activaciones de los mapas de características de ambas imágenes irán convergiendo.

2.3. Matrices de Gram.

En el algoritmo de transferencia neuronal de estilo, el contenido de una imagen viene determinado por el conjunto de activaciones de los mapas de características. El estilo, por el contrario, se determina por las pautas de coactivación existentes entre los pares de mapas de una misma capa. Por pautas de coactivación entendemos la similitud (o disimilitud) de las distribuciones de valores de los dos mapas.

Dichas pautas de coactivación son descritas por el algoritmo de la Transferencia Neuronal de Estilo a través de las denominadas matrices de Gram. A cada capa de la red le corresponde una matriz de Gram. Y cada elemento de esta matriz es igual al producto de Frobenius de dos mapas de características diferentes.

$$G_{i,j} =< F_i, F_j >_F = \sum_{n=1}^{n=x} \sum_{n=1}^{n=y} (n_{x,y,i} n_{x,y,j}) \quad (3)$$

Donde $G_{i,j}$ es el elemento (i, j) de la matriz de Gram de la capa convolucional de que se trate y $< F_i, F_j >_F$ el producto de Frobenius de los mapas de características F_i y F_j. El sumatorio tiene lugar a través de las coordenadas x e y. Lo que se multiplica (y después se suma) son los valores de activación de las neuronas.

La nueva matriz de Gram tendrá, pues, tantos elementos como posibles pares de mapas de características en dicha capa. Si la quinta capa convolucional tiene 512 mapas, como sucede en el caso de la arquitectura VGG16, entonces su matriz de Gram tendrá $512^2 = 262144$ elementos diferentes.

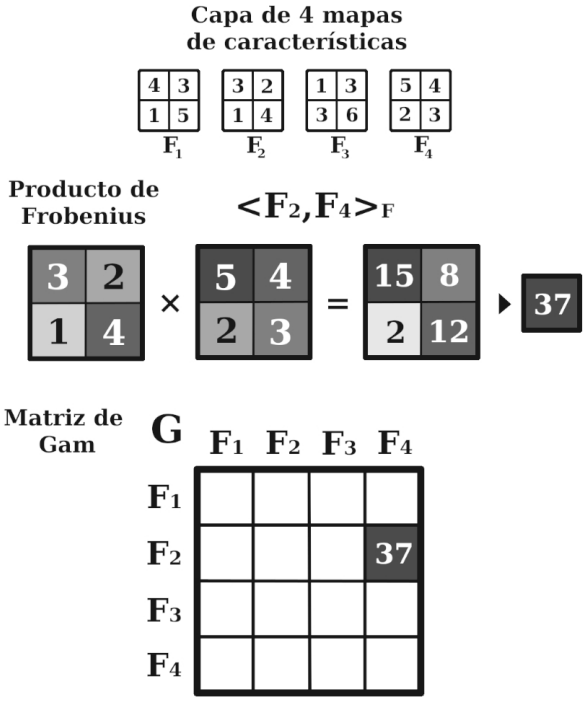


Figura 1: En este ejemplo mostramos una capa de cuatro mapas de características, de una resolución de 2×2 (o también $x = 2$ e $y = 2$). Tras esto mostramos el producto de Frobenius de los mapas F_2 y F_4 que resultará en una de las celdas de la matriz de Gram.

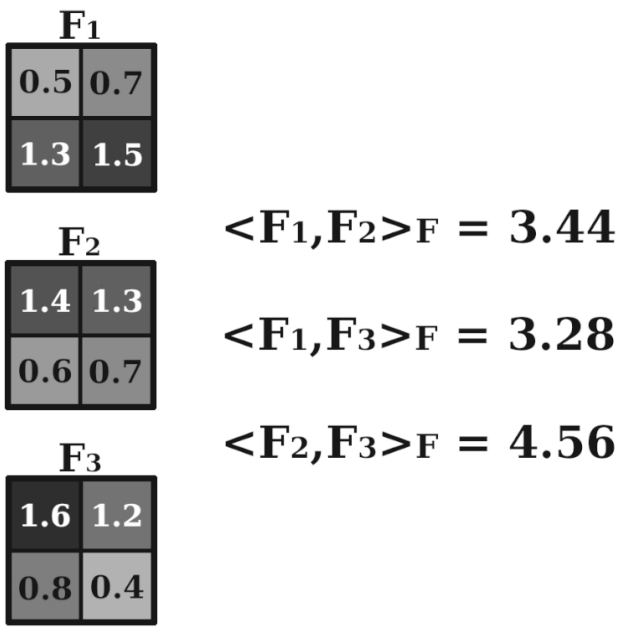


Figura 2: Mostramos aquí tres mapas de características. Aunque la media de todos ellos es la misma, vemos que los valores de los mapas F_2 y F_3 se distribuyen de forma similar. Por eso, el producto de Frobenius de ambos mapas es mayor que el de otros pares. Tanto los productos de Frobenius que muestran correlación entre mapas, como los que manifiestan su ausencia, forman en conjunto el estilo de una imagen, codificado en la matriz de Gram.

2.4. Coactivación de mapas.

Usualmente, en las redes convolucionales la operación de convolución suele venir acompañada de un procedimiento de normalización por lotes y de la aplicación subsiguiente de la función rectificadora. La normalización provoca que los valores de los mapas de características (resultado de la mera aplicación del kernel convolucional) tengan una media de 0 y una desviación típica de 1. La función de activación rectificadora posterior elimina los valores negativos. Después de estas dos operaciones se calcula el producto de Frobenius y la matriz de Gram. Tras la normalización de los mapas, la norma de Frobenius de todos ellos tiene un valor similar. Como consecuencia de ello, los diferentes valores de la matriz de Gram vendrán causados por la coincidencia (o divergencia) de los valores de activación de las neuronas de los dos mapas. Cuando dos matrices están normalizadas, su producto de Frobenius nos indicará las similitudes en la distribución de sus valores. **Cuanto mayor sea la coincidencia entre los valores de activación, mayor será el producto de Frobenius**.

2.5 Función de pérdida.

La matriz de Gram contiene el estilo de la imagen, es decir, las pautas de respuesta conjunta de los mapas de características de la capa. **La función de pérdida de estilo se calcula comparando las matrices de Gram de la imagen generada y de la imagen de la que tomamos el estilo**. El algoritmo buscará minimizar la diferencia entre ambas, y así transferirá el estilo de una imagen a otra.

El término de la función de pérdida relacionada con el estilo se calcula también para cada capa convolucional. Se expresa algebraicamente de la siguiente forma:

$$L_{\text{estilo}} = \frac{1}{4N^2} \sum (G_{i,j} - A_{i,j})^2 \qquad (4)$$

Donde N es el número de neuronas de la capa, $G_{i,j}$ es la matriz de Gram de la imagen que estamos construyendo y $A_{i,j}$ la del cuadro cuyo estilo queremos copiar. Las dos matrices se restan y tras ello se calcula la norma de la matriz resultante. El sumatorio tiene lugar, pues, a través de los

índices i, j. Conforme la función de pérdida tienda a cero, los estilos de las dos imágenes irán convergiendo.

Para reconstruir eficientemente la información relativa al estilo de una obra de arte es necesario utilizar en el cómputo los datos de las capas convolucionales más profundas, que contienen información global sobre la imagen, al contrario de lo que sucede en la reconstrucción de los rasgos de la fotografías, que puede realizarse perfectamente a partir de las capas más cercanas a la entrada.

3. Pasos del algoritmo de Transferencia Neuronal de Estilo.

Los pasos del algoritmo son los siguientes:

1. La imagen de la obra de arte se hace pasar a través de la red. Las matrices de Gram A_{ij} son entonces calculadas para las primeras capas de cada uno de los cinco bloques convolucionales.

2. La imagen de la fotografía se hace pasar a través de la red. Se calculan los valores de activación $n_{(x,y,z)}$ de las neuronas de la primera capa de convolución del cuarto bloque.

3. Se crea una imagen aleatoria a partir de ruido. Dicha imagen también se hace pasar a través de la red.

4. $L_{\text{contenido}}$ sólo se calcula para la primera capa del cuarto bloque convolucional. En el resto de las capas es cero. L_{estilo} y su gradiente se calculan para la primera capa de cada bloque convolucional.

5. Los gradientes se suman y se retropropagan hacia la entrada.

6. Se modifican los valores de la función de activación de la capa de entrada, cuyos datos -ya modificados- vuelven a penetrar de nuevo en la red.

7. Los pasos descritos en los tres apartados anteriores se iteran cientos de veces.

3.1. Parámetros.

Durante la síntesis de la imagen se produce una suerte de pugna entre los dos términos de la función de pérdida, de tal manera que el primero de ellos ($L_{\text{contenido}}$, parametrizado por λ_c), pujará porque la imagen de resultado absorba el máximo contenido semántico y geométrico posible de la fotografía original. El segundo (L_{estilo}, parametrizado por λ_s) término, por el contrario, tratará de compeler a la red y a la imagen final a asumir información relativa al estilo de la obra de arte señalada. Variando ambos hiperparámetros podemos conseguir que un tipo de información predomine sobre el otro.

Figura 3: Modulando los hiperparámetros λ_c y λ_s de la función de pérdida podemos hacer que predomine la información visual sobre la estilística o viceversa. Cuanto más a la derecha, mayor es la ratio $\frac{\lambda_c}{\lambda_s}$. La obra de arte es "Composición VII", de Vasili Kandinski.

4. Interpretación de la Transferencia Neuronal de Estilo.

La principal novedad del algoritmo que a grandes rasgos hemos analizado es que es capaz de capturar la información relativa al estilo de una fotografía o una obra de arte determinada.

Dicho estilo viene determinado por las relaciones de correlación entre las activaciones de los mapas de características de una misma capa. Ello parece señalar una suerte de comportamiento hebbiano, que afectaría, no tanto a las neuronas individuales como a los mapas de características en su conjunto. Si dos mapas de características reaccionan de una manera parecida ante la imagen que porta el estilo, el algoritmo fuerza también a que haya la misma reacción conjunta durante la síntesis de la imagen final. De esta manera se transmite el estilo de una imagen a la otra.

Nos preguntamos si en el cerebro humano también existen mecanismos de aprendizaje hebbiano que afectarían a grupos enteros de neuronas, encargados de procesar diferentes tipos de información. Tal vez sea éste uno de los mecanismos que permiten al cerebro procesar niveles superiores de significación a la simple segmentación semántica.

5. Reporte.

5.1. Modelo VGG-16.

Antes de comenzar con la transferencia de estilo de imágenes, no está de más comprender cómo funciona el modelo VGG16 en la tarea de clasificación de imágenes. Con el siguiente código, podemos cargar cualquier imagen, predecir sus clases usando el modelo VGG16.

```
# Cargar el modelo VGG16 preentrenado
model = VGG16(weights='imagenet')

# Cargar una imagen de ejemplo y
    preprocesarla
image_path = 'Nami_op.jpg'
img = load_img(image_path, target_size=(224,
     224))
img_array = img_to_array(img)
img_array = np.expand_dims(img_array, axis
    =0)
img_array = preprocess_input(img_array)

# Realizar la prediccion
predictions = model.predict(img_array)

# Decodificar y mostrar los resultados
decoded_predictions = decode_predictions(
    predictions, top=3)[0]
```

Con este código, podemos cargar cualquier imagen y predecir sus clases usando el modelo VGG16. Esta capacidad de clasificación y reconocimiento de características del modelo VGG16 es la base fundamental para técnicas más avanzadas como la transferencia de estilo de imágenes.

Nota: La diferencia principal entre VGG16 y VGG19 es el número de capas, lo que puede afectar la capacidad del modelo para aprender representaciones más complejas de las imágenes, pero también puede aumentar la complejidad computacional y el riesgo de sobreajuste en conjuntos de datos pequeños.

Apartir de ahora usaremos VGG19.

5.2. Transferencia de Estilo.

Este fragmento de código se encarga de cargar las imágenes de destino y de referencia de estilo, y también calcula las dimensiones de la imagen que se generará durante el proceso de transferencia de estilo.

```
from keras.preprocessing.image import
    load_img, img_to_array
from PIL import Image

# La imagen que adoptar  el estilo
target_image_path = "gato.jpg"

img = Image.open(target_image_path)
img_width, img_height = img.size

# La imagen que dar  el estilo
style_reference_image_path = "Estilo.jpg"

# Dimensiones de la imagen generada
width, height = load_img(target_image_path).
    size
img_height = img_height
img_width = img_width
```

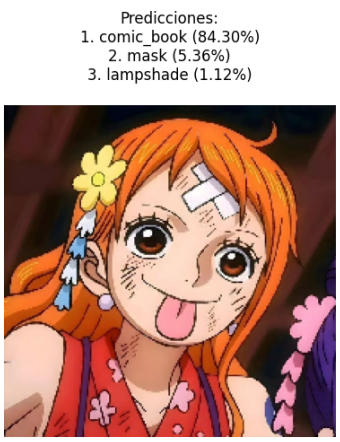


Figura 4: Ejemplo de VGG-16.

Necesitaremos algunas funciones auxiliares.

```
import numpy as np
from keras.applications import vgg19

def preprocess_image(image_path):
    img = load_img(image_path, target_size=(
        img_height, img_width))
    img = img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img = vgg19.preprocess_input(img)
    return img

def deprocess_image(x):
    x[:, :, 0] += 103.939
    x[:, :, 1] += 116.779
    x[:, :, 2] += 123.68
    x = x[:, :, ::-1]  # BGR -> RGB
    x = np.clip(x, 0, 225).astype("float64")
    return x
```

estas funciones son utilizadas para preparar las imágenes para su entrada al modelo VGG19 y para revertir las transformaciones después de la generación de imágenes. Esto es comúnmente utilizado en aplicaciones de transferencia de estilo donde se emplea el modelo VGG16.

Configuramos la red VGG19. Tomamos como entrada un lote de tres imágenes: la imagen de referencia del estilo, la imagen de destino y un marcador de posición que contendrá la imagen generada. Un marcador de posición es simplemente un tensor simbólico, cuyos valores se proporcionan externamente a través de matrices Numpy. La referencia de estilo y la imagen de destino son estáticas y, por lo tanto, se definen mediante K.constant, mientras que los valores contenidos en el marcador de posición de la imagen generada cambiarán con el tiempo.

```
from keras import backend as k

target_image = k.constant(preprocess_image(
    target_image_path))
style_reference_image = k.constant(
    preprocess_image(
    style_reference_image_path))

# Este marcador de posici n contendr
    nuestra imagen generada
combination_image = k.placeholder((1,
    img_height, img_width, 3))

# Combinamos las 3 imagenes en un solo lote
input_tensor=k.concatenate([target_image,
    style_reference_image,combination_image
    ], axis=0)
```

Este fragmento de código establece las imágenes de destino y de referencia de estilo como tensores constantes, define un marcador de posición para la imagen generada y combina todas las imágenes en un solo tensor para su entrada en el modelo de transferencia de estilo.

Construimos la red vgg19 con nuestro lote de 3 imágenes por entrada.

```
model = vgg19.VGG19(input_tensor=
    input_tensor,
                    weights= "imagenet",
                    include_top=False)
print("Modelo cargado correctamente.")
```

El modelo se cargara con pesos de ImageNet previamente entrenados.

Definimos la pérdida de contenido, destinada a garantizar que la capa superior del convnet VGG16 tenga una imagen similar a la imagen de destino y a la imagen generada:

```
def content_loss(base, combination):
  return k.sum(k.square(combination - base))
```

Ahora, aquí está la pérdida de estilo. Aprovecha una función auxiliar para calcular la matriz de Gram de una matriz de entrada, es decir, un mapa de las correlaciones encontradas en la matriz de características original.

```
def gram_matrix(x):
    features = k.batch_flatten(k.
        permute_dimensions(x, (2,0,1)))
    gram = k.dot(features, k.transpose(
        features))
    return gram

def style_loss(style, combination):
    S = gram_matrix(style)
    C = gram_matrix(combination)
    channels = 3
    size = img_height * img_width
    return k.sum(k.square(S-C)) / (4. * (
        channels ** 2) * (size ** 2))
```

A estos dos componentes de pérdida sumamos un tercero, la "pérdida por variación total". Su objetivo es fomentar la continuidad espacial en la imagen generada, evitando así resultados demasiado pixelados. Podrías interpretarlo como una pérdida de regularización.

```
def total_variation_loss(x):
    a = k.square(
        x[:, :img_height -1, :img_width -1,
            :] - x[:, 1:, :img_width -1, :])
    b = k.square(
        x[:, :img_height -1, :img_width -1,
            :] - x[:, :img_height -1, 1:,
            :])
    return k.sum(k.pow(a+b, 1.25))
```

La pérdida que minimizamos es un promedio ponderado de estas tres pérdidas. Para calcular la pérdida de contenido, solo aprovechamos una capa superior, la capa blocks_conv2, mientras que para la pérdida de estilo usamos una lista de capas que abarca tanto las capas de bajo como las de alto nivel, agregamos la pérdida de variación total al final.

Dependiendo de la imagen de referencia de estilo y la imagen de contenido que esté utilizando, es probable que desee ajustar el coeficiente content_weight, el contribución de la pérdida de contenido a la pérdida total. Un content_weight más alto significa que el contenido de destino será más reconocible en el imagen generada.

Este código combina las pérdidas de contenido, estilo y variación total para formar la pérdida total que se utilizará en el proceso de optimización para generar una imagen que conserve el contenido de una imagen de referencia y el estilo de otra imagen de referencia.

```
# Crear un diccionario que asocia nombres de
    capas con sus salidas
outputs_dict = dict([(layer.name, layer.
    output) for layer in model.layers])

# Nombre de la capa usada para la p rdida
    de contenido
content_layer = "block5_conv2"

# Nombre de las capas usadas para la
    p rdida de estilo
style_layers = ["block1_conv1",
                "block2_conv1",
                "block3_conv1",
                "block4_conv1",
                "block5_conv1"]

# Pesos para las diferentes p rdidas en el
    c lculo total
total_variation_weight = 1e-4  # Peso para
    la p rdida de variaci n total
style_weight = 1.0             # Peso para
    la p rdida de estilo
content_weight = 0.025         # Peso para
    la p rdida de contenido

# Inicializar la variable de p rdida total
loss = k.variable(0.0)

# Extraer caracter sticas de la capa de
    contenido
layer_features = outputs_dict[content_layer]
target_image_features = layer_features[0, :,
    :, :]   # Caracter sticas de la imagen
    objetivo
combination_features = layer_features[2, :,
    :, :]    # Caracter sticas de la imagen
    combinada
loss = loss + content_weight * content_loss(
    target_image_features,
    combination_features)  # A adir la
    p rdida de contenido a la p rdida
    total

# Calcular y a adir la p rdida de estilo
    para cada capa de estilo
for layer_name in style_layers:
    layer_features = outputs_dict[layer_name
        ]
    style_reference_features =
        layer_features[1, :, :, :]  #
        Caracter sticas de la imagen de
        referencia de estilo
    combination_features = layer_features[2,
        :, :, :]      # Caracter sticas de
        la imagen combinada
    sl = style_loss(style_reference_features
        , combination_features)  # Calcular
        la p rdida de estilo
    loss += (style_weight / len(style_layers
        )) * sl        # A adir la p rdida
        de estilo ponderada a la p rdida
        total

# A adir la p rdida de variaci n total a
    la p rdida total
loss += total_variation_weight *
    total_variation_loss(combination_image)
```

Finalmente, configuramos el proceso de descenso de gradiente. En el artículo original de Gatys et al., la optimización se realiza utilizando el algoritmo L-BFGS, por lo que es también lo que usaremos aquí. Esta es una diferencia clave con el ejemplo de Deep Dream de la sección anterior. L-BFGS viene empaquetado con SciPy. Sin embargo, existen dos pequeñas limitaciones con la implementación de SciPy:

- Requiere que se pase el valor de la función de pérdida y el valor de los gradientes como dos funciones separadas.

- Solo se puede aplicar a vectores, mientras que tenemos una matriz de imágenes 3D.

```
import tensorflow as tf

# Funci n para calcular la p rdida y los
    gradientes
@tf.function
def compute_loss_and_grads(image):
    with tf.GradientTape() as tape:
        tape.watch(image)
        loss = total_variation_weight *
            total_variation_loss(image)
    grads = tape.gradient(loss, image)
    return loss, grads

# Clase Evaluator para evaluar la p rdida y
    los gradientes
class Evaluator(object):
    def __init__(self):
        self.loss_value = None
        self.grad_values = None

    # M todo para calcular la p rdida
    def loss(self, x):
        assert self.loss_value is None
        x = x.reshape((1, img_height,
            img_width, 3))
        x_tensor = tf.convert_to_tensor(x)
        loss_value, grad_values =
            compute_loss_and_grads(x_tensor)
        self.loss_value = loss_value.numpy()
        self.grad_values = grad_values.numpy
            ().flatten().astype("float64")
        return self.loss_value

    # M todo para obtener los gradientes
    def grads(self, x):
        assert self.loss_value is not None
        grad_values = np.copy(self.
            grad_values)
        self.loss_value = None
        self.grad_values = None
        return grad_values

# Dimensiones de la imagen
img_height = 400
img_width = 1846

# Instancia de la clase Evaluator
evaluator = Evaluator()
```

Finalmente, podemos ejecutar el proceso de ascenso de gradiente utilizando el algoritmo L-BFGS de SciPy.

```
from scipy.optimize import fmin_l_bfgs_b
import time
import imageio

# Prefijo para los nombres de los archivos
    de resultado
result_prefix = "style_transfer_result"
# N mero de iteraciones de optimizaci n
iterations = 1

# Preprocesar la imagen de destino
x = preprocess_image(target_image_path)
# Aplanar la imagen preprocesada para la
    optimizaci n
x = x.flatten()
# Bucle para realizar las iteraciones de
    optimizaci n
for i in range(iterations):
    print("Inicio de iteraciones", i)
    start_time = time.time()  # Registrar el
        tiempo de inicio
    # Ejecutar la optimizaci n L-BFGS-B
    x, min_val, info = fmin_l_bfgs_b(
        evaluator.loss, x,fprime=evaluator.
        grads,maxfun=20)
    print("Valos de perdida actual:",
        min_val)

    # Copiar y remodelar la imagen
        optimizada
    img = x.copy().reshape((img_height,
        img_width, 3))
    # Deshacer el preprocesamiento para
        obtener la imagen final
    img = deprocess_image(img)
    # Generar el nombre del archivo de
        resultado
    fname = result_prefix + "_at_iteration_%
        d.png" % i

    # Guardar la imagen resultante
    imageio.imwrite('Imagen generada.png',
        img)


    end_time = time.time()  # Registrar el
        tiempo de finalizaci n
    print("Iteraci n %d completa en %ds" %
        (i, end_time - start_time))
```

```
Lossy conversion from float64 to uint8. Range [10.659027092196538, 225.0]. Convert image to uint8 prior to saving to suppress t
his warning.

Valos de perdida actual: 541.337896559444
Iteración 4 completa en 27s
Inicio de iteraciones 5

Lossy conversion from float64 to uint8. Range [11.669977733856541, 225.0]. Convert image to uint8 prior to saving to suppress t
his warning.

Valos de perdida actual: 382.4766530261215
Iteración 5 completa en 27s
Inicio de iteraciones 6

Lossy conversion from float64 to uint8. Range [12.474213701735934, 225.0]. Convert image to uint8 prior to saving to suppress t
his warning.

Valos de perdida actual: 287.5230979796815
Iteración 6 completa en 26s
Inicio de iteraciones 7

Lossy conversion from float64 to uint8. Range [13.099059697679039, 225.0]. Convert image to uint8 prior to saving to suppress t
his warning.

Valos de perdida actual: 232.39695765455483
Iteración 7 completa en 31s
Inicio de iteraciones 8

Lossy conversion from float64 to uint8. Range [13.970930146344443, 222.21136941410526]. Convert image to uint8 prior to saving
to suppress this warning.

Valos de perdida actual: 175.0549582712559
Iteración 8 completa en 27s
Inicio de iteraciones 9

Lossy conversion from float64 to uint8. Range [14.57725237791091, 219.84556942886658]. Convert image to uint8 prior to saving t
o suppress this warning.

Valos de perdida actual: 143.67505751877204
Iteración 9 completa en 26s
```

Figura 5: 10 iteraciones.

```
img
```

```
array([[[112.92480713,  92.09454996,  78.33110161],
        [112.90872923,  92.09828283,  78.38038591],
        [112.75812285,  92.01065367,  78.44800104],
        ...,
        [213.18483706, 169.59611749, 130.78901227],
        [213.2461824 , 169.65152979, 130.85637693],
        [213.32426996, 169.71931245, 130.95376765]],

       [[112.38580749,  91.70177556,  78.3273949 ],
        [112.38243316,  91.70748825,  78.37120918],
        [112.24356619,  91.65443089,  78.41269794],
        ...,
        [213.09559507, 169.52110364, 130.69623248],
        [213.15128094, 169.56979784, 130.74506081],
        [213.21102516, 169.62528047, 130.79075384]],

       [[111.52079946,  91.14214943,  78.22142214],
        [111.48513039,  91.13434629,  78.23096995],
        [111.33923058,  91.06001936,  78.24842775],
        ...,
        [212.95993111, 169.40364341, 130.56008171],
        [213.00328682, 169.44050818, 130.58920787],
        [213.03461887, 169.46411747, 130.59465506]],

       ...,
```

Figura 6: Resultado

Conclusión.

En conclusión, la técnica de Transferencia de Estilo ofrece una curiosa e interesante ventana a la creatividad digital al permitir la transferencia de estilos artísticos entre imágenes.

Sin embargo, al intentar mostrar la imagen generada, se presenta un error, probablemente debido a que la imagen excede la normalización de 255. A pesar de múltiples intentos, no ha sido posible ajustar correctamente los valores para visualizarla de manera adecuada. Además, cambiar el tipo de dato de las imágenes causa un desorden en los tensores, complicando aún más la visualización directa de la imagen.

No obstante, podemos observar los efectos del cambio al analizar los píxeles de la imagen generada. Esta inspección pixel a pixel permite apreciar cómo se han transformado y combinado los elementos visuales para crear una nueva obra de arte. En definitiva, aunque la visualización directa de la imagen generada presenta desafíos, el análisis matricial de los píxeles confirma el éxito de la transformación estilística lograda mediante NST.