Grado en Matemática Computacional

Trabajo Final de Grado

Uso de características desacopladas en redes convolucionales para transferencias de estilo

Autor:
Paula Miralles Simó

Supervisor:
Marina Martínez Garcia
Tutor académico:
Marina Martínez Garcia

Fecha de lectura: 10 de julio de 2022
Curso académico 2021/2022

Resumen

Este documento corresponde al Trabajo Final de Grado del Grado en Matemática Computacional (asignatura MT1054), en él abordamos un problema práctico en el que se aplican los diferentes conocimientos alcanzados durante los estudios del grado. Incluye tanto la fundamentación teórica de las técnicas utilizadas como los resultados obtenidos.

El problema planteado se enmarca dentro de la aplicación de las redes neuronales, en particular, redes neuronales convolucionales en el análisis y procesado de imágenes digitales. El objetivo planteado es el uso de características desacopladas mediante gradientes ortogonales para aumentar el poder descriptivo del estilo de una imagen.

Una red neuronal convolucional es una variación de un perceptrón multicapa, consiste en múltiples capas de filtros convolucionales generalmente aplicados en matrices bidimensionales, es por ello que han demostrado ser muy eficientes en la clasificación, segmentación y análisis de imágenes digitales.

El método de desacoplamiento de características propuesto es el denominado desacoplamiento por normalización y está basado en definir transformaciones dentro de variedades, siguiendo trayectorias a lo largo de los gradientes de las características. Estas transformaciones permiten definir una normalización que posibilita desacoplar características diferenciables.

Se ha trabajado en concreto a partir una red CNN VGG-19, una red neuronal convolucional que consta de 19 capas (16 capas de convolución, 3 capas *fully-connected*, 5 capas *max-pooling* y una capa *soft max*) y los resultados obtenidos han sido satisfactorios.

Palabras clave

Desacoplamiento, característica, style transfer, redes neuronales, aprendizaje profundo.

Keywords

Decoupling features, style transfer, neural networks, deep learning.

Índice general

1. Introducción **7**

1.1. Motivación . . . 7

1.2. Objetivos . . . 9

2. Contexto teórico **11**

2.1. Características y desacoplamiento . . . 11

2.2. El método de desacoplamiento . . . 12

2.3. Redes neuronales y aprendizaje profundo . . . 16

2.3.1. Redes neuronales . . . 16

2.3.2. Redes neuronales convolucionales . . . 18

2.3.3. VGG-19 . . . 20

2.4. Modelar texturas . . . 21

2.5. Redes neuronales aplicadas a la transferencia . . . 22

2.6. Transferencias de estilo mediante ruidificación . . . 25

3. Contexto computacional **27**

3.1. Aspectos computacionales previos 27

3.2. Paso 0: medir las medias de los canales en una imagen de ruido. 30

3.3. Paso 1: medir las características en la imagen origen. 31

3.4. Paso 2: ruidificar la imagen destino. 33

3.5. Paso 3: transferimos a la imagen destino el estilo de la imagen origen. 34

4. Resultados **37**

4.1. Explicaciones previas 37

4.2. Resultado de la transferencia 38

4.2.1. Transferencia usual con 2 capas 38

4.2.2. Auto transferencia con 2 capas 39

4.3. Gráficas de las funciones de pérdida. 39

4.4. Cómo se distribuyen las medias de los canales en una imagen de ruido 41

5. Conclusiones **45**

Capítulo 1

Introducción

1.1. Motivación

En este trabajo nos basamos en imágenes RGB (Red-Green-Blue) naturales. Una imagen RGB con una resolución de m por n píxeles es guardada en el ordenador como una matriz de datos m por n por 3 que define los componentes de color rojo, verde y azul para cada píxel particular, es decir, la intensidad en cada uno de estos tres canales. Teniendo en cuenta esta representación espacial de las imágenes en un espacio de tres dimensiones RGB, el total de las imágenes posibles es gigante pero no todas las imágenes tienen sentido en un entorno natural, en las imágenes naturales siempre encontramos unas relaciones de sus píxeles con los píxeles de su entorno.

Una imagen natural está formada por lo que se conoce como texturas visuales, regiones de la imagen que siguen algún patrón regular respecto a la disposición espacial del color o las intensidades (ver la Figura 1.1).

Figura 1.1: Ejemplos de texturas visuales naturales [1].

Siempre ha resultado difícil caracterizar las imágenes naturales utilizando modelos matemáticos deterministas y/o estadísticos. El término “estilo”de la imagen es difícil de definir, la iluminación, la escena, la cámara pueden tener un impacto esencial en la forma en que el espectador aprecia una imagen esto sin mencionar los parámetros propios del observador que van desde sus experiencias previas hasta sus atributos fisiológicos. Hay que tener en cuenta también que el cerebro es capaz de normalizar estas imágenes: analizamos lo que ocurre en ese espacio respecto a su entorno. Por ejemplo, cuando una zona de una imagen está muy oscura, si cambiamos sus vecinos y estos se oscurecen, esta pasa a ser una zona más clara.

Sin embargo, un textura visual, que como hemos dicho, corresponde a una pequeña parte de esta representación de la imagen natural, es en general más sencilla de caracterizar matemáticamente y, en particular, bastante susceptible a una representación estadística.

En este sentido, Julesz [6] fue capaz de caracterizar estadísticamente las texturas de una imagen y Portilla y Simoncelli [9] afianzaron una descripción estadística de la textura visual. Esta última representación dio pie a cientos de estudios sobre síntesis y transferencias de texturas [2, 3, 1] que llevaron al mismo Ero Simoncelli a demostrar el poder descriptivo de las medias de los canales de una red neuronal VGG-19 pre-entrenada para clasificación, para reproducir texturas visuales en color con una calidad aceptable [1].

Por otra parte, Javier Portilla, Eduardo Martínez-Enríquez y Mar González [7] establecieron un nuevo modelo de textura paramétrica: presentaron un formalismo para desacoplar características deterministas y mostraron su aplicabilidad para mejorar la transferencia de estilos.

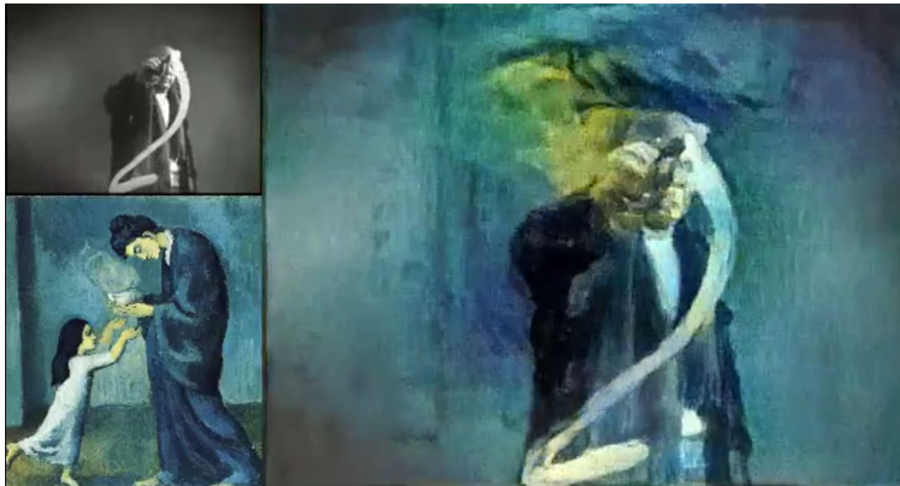


Figura 1.2: Ejemplo de transferencia de estilo.

La transferencia de estas características de reproducción del color, textura, trazo, ... de una imagen natural a otra (ver la Figura 1.2) es una parte importante en el trabajo actual de los artistas gráficos, el mundo de los videojuegos y la industria audiovisual en general. En el campo del procesamiento de imágenes, esto se denomina transferencia de estilo, un término general bajo el cual se han propuesto cientos de métodos para su cálculo automático.

1.2. Objetivos

El objetivo del trabajo es comprobar si el uso de características desacopladas mediante gradientes ortogonales aumenta el poder descriptivo del estilo de una imagen a partir de los valores promedio de los canales de una red CNN pre-entrenada. Aplicaremos una estrategia de ortogonalización jerárquica por capas, consistente con la teoría de Portilla, González y Martínez [7].

Los objetivos secundarios son:

- Hacer un repaso de los principales conceptos teóricos matemáticos que se necesitan para entender el funcionamiento de las técnicas propuestas.
- Revisar la literatura acerca de las redes neuronales y, en especial, el uso de las redes neuronales convolucionales para la detección de objetos como herramientas útiles para la transferencia de estilos.
- Implementar un método de normalización de una imagen a través de la ruidificación.
- Desarrollar un código capaz de transferir el estilo de una imagen origen a otra destino.

Capítulo 2

Contexto teórico

Este capítulo está dedicado al desarrollo teórico de las técnicas propuestas. Empezamos introduciendo todas las definiciones y conceptos necesarios, usando los términos y la notación habitual en el contexto del análisis de imágenes.

En todo este capítulo $x = (x_1, ..., x_N)$ denotará el vector N-dimensional obtenido al disponer de forma secuencial las filas (o columnas) de una matriz de tamaño $N = n \times m$, representando un canal de una imagen RGB, lo que denominaremos como *vectorización*.

2.1. Características y desacoplamiento

Definición 1 *Se define una característica f de un vector x como una función real diferenciable:*

$$f : \Omega \subset \mathbb{R}^N \to \mathbb{R}, \tag{2.1}$$

para un dominio Ω.

Se define una característica global como una característica que depende de todos los coeficientes del vector [7].

Las características usadas más habitualmente para caracterizar imágenes son funciones promedio que juegan el mismo papel que los estadísticos muestrales, por ejemplo la media ($f(x) = \bar{x} = \frac{\sum_{i=1}^{N} x_i}{N}$) o la varianza ($f(x) = \frac{\sum_{i=1}^{N}(x_i - \bar{x})^2}{N}$).

Definición 2 *Decimos que dos características f_i, f_j están desacopladas en un dominio Ω si sus vectores gradiente son ortogonales, es decir:*

$$\nabla f_i(x) \cdot \nabla f_j(x) = 0, \quad \forall x \in D.$$

De igual manera, un conjunto de características están desacopladas en un dominio Ω si todos los posibles pares de características $\{(f_i, f_j) : i, j \in \{1...M\}, i \neq j\}$ están desacoplados.

Decimos que una característica está desacoplada a un conjunto de características si está desacoplada a cada una de las características en ese conjunto.

Intuitivamente, dos características están acopladas cuando comparten una parte de la información. Los ejemplos presentados anteriormente, la media ($f(x) = \bar{x} = \frac{\sum_{i=1}^{N} x_i}{N}$) y la varianza ($f(x) = \frac{\sum_{i=1}^{N}(x_i - \bar{x})^2}{N}$) están acopladas. Si dos características están desacopladas, podemos variar una sin modificar la otra.

Como hemos explicado en el apartado de objetivos, nuestro estudio requiere realizar un desacoplamiento de características. Dado un conjunto de características acopladas, queremos encontrar un nuevo conjunto de características que tengan la propiedad de ser *parecidas* a las originales pero con la propiedad de tener gradientes mutuamente ortogonales. Es decir, dado un conjunto de características acopladas $S = \{f_i : \Omega \to \mathbb{R}, i = 1...M\}$, queremos encontrar un conjunto $\tilde{S} = \{\tilde{f}_i : \Omega \to \mathbb{R}, i = 1...M\}$ de características que sean idénticas a las de S en R_1 y desacopladas en R_D, con $R_1 \subseteq R_D \subseteq \Omega \subset \mathbb{R}^{\mathbb{N}}$. A este conjunto $\tilde{S}$ lo llamaremos conjunto desacoplado en R_D.

La motivación para usar técnicas de desacoplamiento de características no es solo aumentar la eficiencia del proceso de transferencia de características, sino aumentar el poder descriptivo de las características que transferimos [11].

2.2. El método de desacoplamiento

Como hemos explicado en 2.1, debemos definir un método para encontrar este conjunto $\tilde{S}$ con las características descritas en esta sección. Para ello, recurrimos al estudio hecho por Portilla, González y Martínez [7] que presentan un método de desacoplamiento a través de la normalización.

Para empezar, definiremos el concepto de sistema gradiente:

Definición 3 *Fijando una característica f, definida en un conjunto abierto conexo $\overline{\Omega}$. Estudiamos las trayectorias x(t) del problema de valor inicial:*

$$\begin{cases} \frac{dx}{dt} = -\nabla f(x), \\ x(0) = x_0. \end{cases} \quad (2.2)$$

Y esta ecuación diferencial ordinaria se conoce como sistema gradiente. Denotamos también como $I(x_0, f)$ a la trayectoria que pasa por x_0.

Veamos en primer lugar como funciona este método para el caso de dos características.

Dada una característica f, la idea básica de este método consiste en encontrar una transformación:

$$\breve{x}(x) : \mathbb{R}^N \to \mathbb{R}^N, \quad (2.3)$$

de manera que para otra característica, $g(x)$, se cumpla que la característica $\breve{g}(x) = g(\breve{x}(x))$ está desacoplada con f, es decir, $\nabla f(x) \cdot \nabla \breve{g}(x) = 0$

Para ello nos basamos en las siguientes ideas:

- Fijamos un valor de referencia v_{ref} para la característica f, es decir $f(x) = v_{ref}$ (por ejemplo, si la característica es la media, podemos fijar que la media sea cero: $v_{ref} = 0$). A $R_{(v_{ref})} \subset \mathbb{R}^N$ que define $f(x) = v_{ref}$, se denomina variedad de referencia.
- Consideramos las trayectorias definidas localmente por los gradientes, es decir, las trayectorias $I(x)$ del sistema gradiente:

  $$\frac{dx}{dt} = -\nabla f(x),$$

  que denominamos variedades invariantes.
- Buscamos que la transformación $\breve{x}(x)$ sea invariante ante una perturbación a lo largo del gradiente local, es decir si x_1 y x_2 pertenecen a la misma variedad invariante $I(x)$ las imágenes de estos dos puntos sea la misma: $\breve{x}(x_1) = \breve{x}(x_2)$.

Por lo tanto, es natural definir la normalización, $\breve{x}(x; v_{ref})$, como la función que envía un punto x al punto $\overrightarrow{x_{ref}}$ donde su variedad invariante $I(x)$ cruza la variedad de referencia $R_{(v_{ref})}$, que es única.

En un sentido más sencillo, construimos unas curvas que nacen de "movernos por el gradiente", es decir, nos movemos en trayectorias siguiendo el gradiente local de la característica hasta

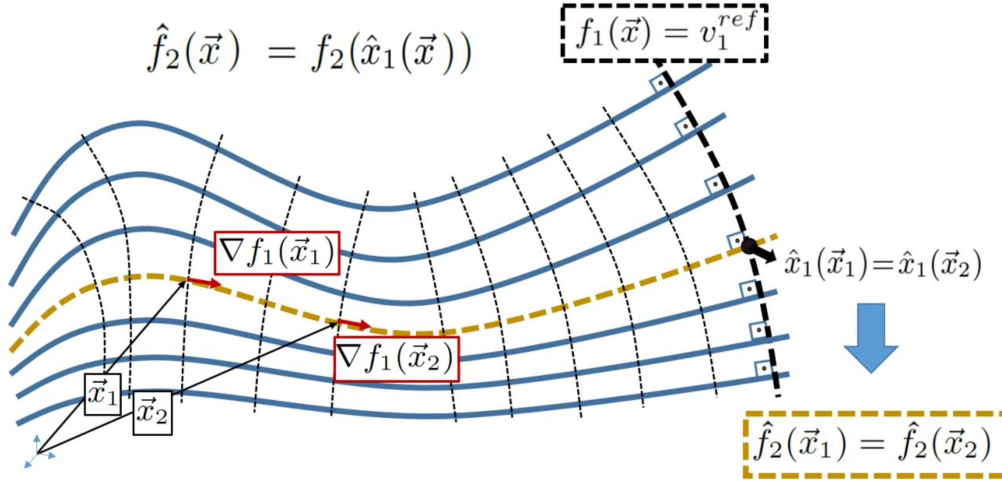


Figura 2.1: Desacoplamiento de dos características mediante la normalización: la normalización propuesta consiste en encontrar la intersección de las subvariedades de invariancia que pasan por x (líneas curvas azules) con la variedad de referencia $R_1 = \{x : f_1(x) = v_1^{ref}\}$ (línea discontinua negra) [7].

encontrar el valor de referencia, v_{ref}, para esa característica. La figura 2.1 muestra este desarrollo en este caso más simple de desacoplamiento mediante normalización de dos características.

Dada otra característica g, definimos $\breve{g}$ como:

$$\breve{g}(x) = g(\breve{x}(x)). \tag{2.4}$$

Es fácil de ver que entonces, aplicando la regla de la cadena para obtener el gradiente, $g(\breve{x}_f(x))$ estará desacoplada con f, es decir, sus gradientes son ortogonales.

Si ahora queremos extender esta definición a una ortogonalización de más de dos características simplemente extendemos el número de dimensiones. El método sigue el esquema de normalización explicado en el caso de las dos características, siguiendo trayectorias dada por los gradientes de todos los f_i. A diferencia del caso sencillo de los sistemas de gradiente, para construir variedades a partir de múltiples gradientes de características deben cumplirse otra serie de condiciones adicionales que se encuentran descritas en [7].

La idea es construir subvariedades multidimensionales de hiperplanos tangentes definidos por los gradientes de las características. Por tanto, ahora para cada punto en el espacio $\overrightarrow{x}_0 \in \mathbb{R}^N$ existe una variedad definida a partir del plano tangente $I(\overrightarrow{x}_0; S)$ que llamaremos variedad de invarianza ($S = \{f_i : \overline{\Omega} \to \mathbb{R}, i = 1...M\}$).

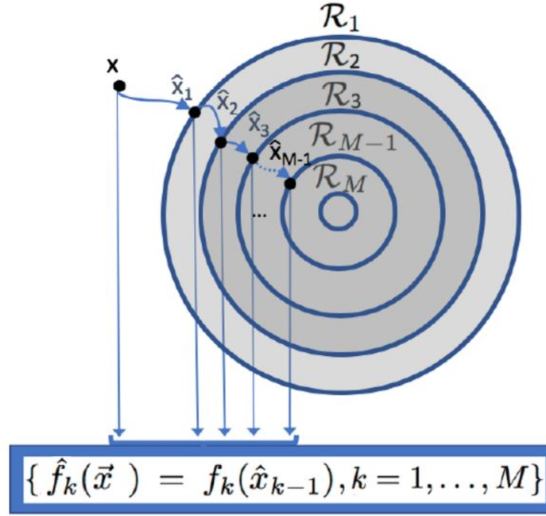


Figura 2.2: Ilustración del algoritmo para ortogonalizar más de una característica: desacoplar nuevas características de las previamente desacopladas, hasta llegar a un conjunto de características desacopladas. [7]

En este caso, y siguiendo las demostraciones hechas por Portilla, González y Martínez, se define la variedad de referencia como:

$$R(v_S^{ref}) = f_S^{-1}(v_S^{ref})$$

Y, por tanto, la normalización que buscamos del conjunto de características en este caso se obtiene a partir de la intersección entre la variedad de invarianza y la variedad de referencia:

$$\check{x}_S(x; v_S^{ref}) = I(x; S) \cap R(v_S^{ref})$$

Los autores probaron que, si existe una trayectoria $I(x, S)$, entonces, cualquier función diferenciable $g(x)$ cumple que $g(\check{x}_S(x))$ está desacoplada de S. La Figura 2.2 ilustra la estructura de un algoritmo para obtener de manera iterativa y sencilla características desacopladas. Los círculos concéntricos representan las variedades de referencia R_k, $k = 1, ..., M$, correspondientes al conjunto de imágenes que tienen los valores de referencia deseados para las características originales de 1 a k. La estructura anidada implica $R_{k+1} \subset R_k$. Esto significa que la imagen se normaliza progresivamente, manteniendo siempre los valores de referencia fijados anteriormente a la vez que buscamos los de la iteración (en un futuro, capa) actual. Las nuevas características se definen como las características originales medidas en las muestras normalizadas. Por lo tanto, obtenemos una nueva característica desacoplada cada vez que se normaliza otra característica original, manteniendo las anteriores también normalizadas.

Retomando el ejemplo habitual, al desacoplar los primeros tres momentos, primero normalizamos la media, lo que implica restarla, es decir, darle valor 0, lo que nos permite definir la

varianza de la muestra, como el segundo orden de la muestra menos su media. Luego normalizamos el valor cuadrático medio, MSE, manteniendo la media cero, lo que nos permite obtener la asimetría, como el momento de tercer orden de la muestra estandarizada, desacoplando así el momento de tercer orden tanto de la media como de la varianza. Eso es, en cada paso medimos el momento muestral de orden k en la muestra normalizada $k \pm 1$ [11].

2.3. Redes neuronales y aprendizaje profundo

2.3.1. Redes neuronales

Las redes neuronales son un modelo algorítmico que se entrena para establecer relaciones entre una entrada y una salida en un conjunto de datos, a través de un proceso que imita la forma en que opera el cerebro humano. Surgieron en los años 50, tuvieron un gran auge a finales de la década de 1980 y resurgieron en 2010 con el nuevo nombre de aprendizaje profundo, con nuevas arquitecturas.

Una red neuronal toma un vector de entrada x con N variables y construye una función $K-$dimensional $F(x)$ que predice la respuesta o la clase de x (dependiendo de si se trata de un problema de predicción o de un problema de clasificación). Esta función no lineal F se construye a partir de elementos llamados nodos o neuronas que a la vez se distribuyen en capas, formando lo que se conoce como la arquitectura de la red neuronal. Se puede definir como una composición de funciones:

$$F(x) = \Theta_n \circ L_n \ldots \Theta_1 \circ L_1(x) \tag{2.5}$$

Donde L_i son funciones lineales y Θ_i no lineales.

La arquitectura de las redes neuronales está compuesta de capas de neuronas, que contienen una capa de entrada, una o más capas ocultas y una capa de salida. Su funcionamiento, explicado en la Figura 2.3 es el siguiente: la primera capa o capa de entrada tendrá tantas neuronas como componentes de x, N; La capa de salida tantas neurona como componentes tengan las salidas deseadas, K. El número de unidades en las capas intermedias o capas ocultas, así como el número, M, de capas intermedias, dependerán de la dificultad de la función a implementar.

- Las variables de entrada , $(x_1, ..., x_N)$ se introducen por las neuronas iniciales. En esta, la respuesta es la identidad y estas neuronas sirven para distribuir las variables en la segunda capa.
- Cada neurona j-ésima de cada capa oculta, $i = 1, ..., M$, recibe un vector de entrada de dimensión k_i, donde k_i es el numero de neuronas de la capa anterior y genera una variable

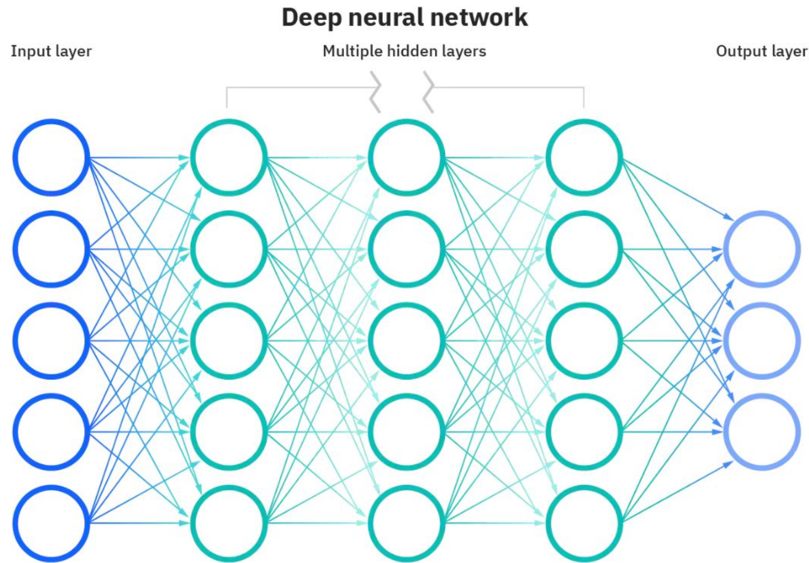


Figura 2.3: Arquitectura tradicional de una red neuronal básica.

escalar de salida:

$$z_j^{(i+1)} = \sigma_i(w_{j0}^{(i)} + w_{j1}^{(i)} z_1^{(i)} + ... + w_{jk_i}^{(i)} z_{k_i}^{(i)}).$$

- Finalmente, cada neurona de la capa de salida recibe el vector $z = (z_1, ..., z_{k_M})$ y genera las K salidas $F_k(x)$ con $k = 1, ..., K$.

σ_i es una función no lineal denominada función de activación. En los inicios se elegía habitualmente como función de activación un función sigmoide $\sigma(z) = \frac{e^z}{1+e^z}$. La opción preferida en las redes neuronales modernas es la ReLU (rectified linear unit), que se define como $\sigma(z) = z_+$. Una activación ReLU puede ser calculada y almacenada de forma más eficiente que una activación sigmoidea [5].

Para llevar a la práctica este método de predicción o clasificación, es decir para entrenar la red neuronal, se necesita estimar todos los parámetros (o pesos) $w\{_{jk}^{(i)}\}$, la estimación de estos parámetros se obtiene minimizando una función de pérdida, que depende de la aplicación en cuestión. Más adelante comentaremos la utilizada en nuestra aplicación.

Para minimizar la función de pérdida se utilizan algoritmos iterativos, los más habituales son los basados en el gradiente, obtenemos eficientemente el gradiente de la función de pérdida con respecto a los parámetros y luego se puede utilizar cualquier algoritmo de optimización basado en gradiente. En este contexto reciben el nombre de propagación hacia atrás, porque realizan un ajuste de los pesos comenzando por la capa de salida, según el error cometido y procede propagando el error has la primera capa.

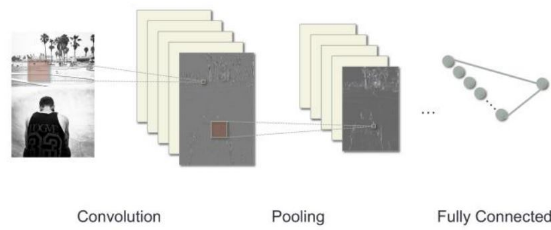


Figura 2.4: Representación esquemática de una red neuronal convolucional con max-pooling.

Las redes neuronales con varias capas intermedias se conocen como redes profundas y se utilizan para algoritmos de aprendizaje profundo.

2.3.2. Redes neuronales convolucionales

En las últimas décadas la aplicación de un tipo especial de redes neuronales profundas, las redes neuronales convolucionales (en inglés, convolutional neural netowork, habitualmente CNN) en las técnicas de aprendizaje automático han jugado un papel protagonista en el reconocimiento y segmentación de imágenes. Las redes neuronales convolucionales están formadas por las capas habituales de una red neuronal: una capa de entrada, varias capas ocultas y una capa de salida. Las capas ocultas son principalmente de dos tipos:

Capa convolucional: Una capa de convolución se compone de una gran cantidad de filtros de convolución, cada uno de los cuales es una plantilla capaz de determinar si una característica local en particular está presente en una imagen.

Un filtro de convolución 2-D se basa en una operación muy simple, llamada convolución, que consiste en multiplicar los elementos del filtro por los de la matriz de entrada y posteriormente sumar los resultados, repitiendo esta operación desplazando el filtro de convolución respecto la matriz de entrada.

Un pequeño ejemplo del funcionamiento del filtro de convolución sería el siguiente: suponemos que tenemos la siguiente representación matricial de una imagen:

$$\text{Imagen original} = \mathrm{I} = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \\ j & k & l \end{pmatrix}$$

Y consideramos también un filtro de convolución 2 x 2 que se representara como:

$$\text{Filtro} = \begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix}$$

Entonces, el resultado de la convolución no será otro que:

$$\text{Imagen convolucionada} = \text{M} = \begin{pmatrix} a\alpha + b\beta + d\gamma + e\delta & b\alpha + c\beta + e\gamma + f\delta \\ d\alpha + e\beta + g\gamma + h\delta & e\alpha + f\beta + h\gamma + i\delta \\ g\alpha + h\beta + j\gamma + k\delta & h\alpha + i\beta + k\gamma + l\delta \end{pmatrix}$$

Más concretamente, el elemento M_{11} proviene de multiplicar punto a punto cada elemento en el filtro 2 $\times$ 2 por la submatriz 2 $\times$ 2 de I, $\left(\begin{smallmatrix} a & b \\ d & e \end{smallmatrix}\right)$ y sumar los resultados.

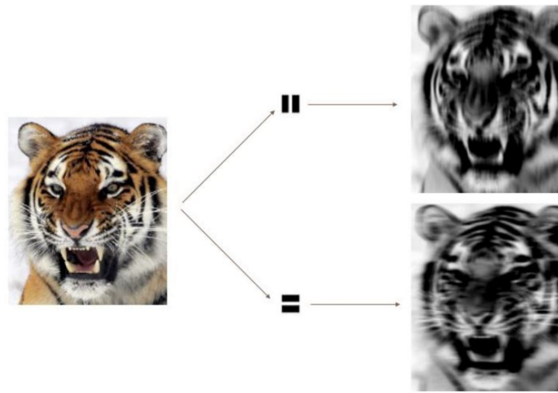


Figura 2.5: Los filtros de convolución aplicados a esta imagen del tigre nos devuelven aquellas características de la imagen que se asemejan a las del filtro: la imagen superior es el resultado de asignar valores grandes a las partes de la imagen que contienen franjas verticales, mientras que la imagen inferior ocurre igual con las franjas horizontales.

La Figura 2.5 nos muestra la aplicación de dos filtros de convolución a una imagen de 192 $\times$ 179 de un tigre. Cada filtro de convolución es una matriz de 15 $\times$ 15 que contiene ceros (negro) con una franja estrecha de unos (blanco) orientado vertical u horizontalmente respectivamente. Cuando cada filtro convoluciona con la imagen del tigre, las áreas del tigre que tienen algún parecido al filtro (es decir, que tienen franjas o bordes horizontales o verticales) reciben valores grandes, y las áreas del tigre que no se asemejan a la característica se asignan pequeños valores. Vemos que el filtro de franjas horizontales selecciona franjas y bordes horizontales en la imagen original, mientras que el filtro de franjas verticales selecciona franjas y bordes verticales en la imagen original [5].

Dado que la imagen de entrada es en color, tiene tres canales representados por un mapa de características tridimensional. Además, un filtro de convolución también tendrá tres canales, uno por cada color. Los resultados de las tres circunvoluciones se suman para formar la salida en forma de mapa de características bidimensional.

Capa de *pooling*: reduce la dimensionalidad de la matriz de entrada. El alcance de esta capa no es solo reducir la carga computacional, sino también realizar la selección de características.

Si bien hay varias formas posibles de realizar el *pooling*, la operación *max-pooling* resume cada bloque de píxeles de 2 × 2 que no se superponen en una imagen utilizando el valor máximo del bloque. Esto reduce el tamaño de la matriz por un factor de dos en cada dirección, y también proporciona una invariancia local, es decir, siempre que haya un valor grande en uno de los cuatro píxeles del bloque, todo el bloque se registra como un valor grande en la imagen reducida.

Un ejemplo sencillo de *max-pooling* es el siguiente:

$$\begin{pmatrix} 1 & 2 & 5 & 6 \\ 3 & 4 & 7 & 8 \\ 9 & 10 & 13 & 14 \\ 11 & 12 & 17 & 18 \end{pmatrix} \rightarrow \textit{max-pooling} \rightarrow \begin{pmatrix} 4 & 8 \\ 12 & 18 \end{pmatrix}$$

La capa de salida tiene la forma de una capa *full-connected*, con tantas neuronas como clases en la tarea de clasificación. Las salidas se normalizan con una función de activación *softmax*, que es un tipo de función de activación diferente a las explicadas anteriormente, puesto que se utiliza en la capa de salida en redes utilizadas para clasificar en K categorías. Esta función se asegura de que las salidas de la CNN sumen 1, es decir, que se comporten como probabilidades. Esta función es una generalización de la función sigmoide vista anteriormente.

2.3.3. VGG-19

En un intento de mejorar los resultados de estos reconocimientos, se encontró que la capacidad de las redes neuronales convoluciones multicapa, entrenadas con descenso de gradiente las convierte en candidatas obvias para tareas de reconocimiento de imágenes [8].

En este trabajo, la CNN utilizada será la VGG-19, una red neuronal creada por Simonyan y Zisserman, de la Universidad de Oxford [10]. Se trata de una red convolucional formada como muestra la Figura 2.6 de 19 capas, con 16 correspondientes a capas covolucionales y 3 *fully-connected* con capas de *max-pooling*. El modelo VGG-19 cuenta con un total de 138 millones de parámetros y está entrenado en más de un millón de imágenes y puede clasificar imágenes en 1000 categorías de objetos [13].

Como la mayoría de las CNN, VGG descarta información en cada etapa de transformación. La arquitectura VGG es modificada para lograr un aspecto importante: la transformación debe ser inyectiva, es decir, distintas entradas deberían corresponder a distintas salidas. Esto es necesario para garantizar que la medida de calidad final sea una métrica adecuada (en el

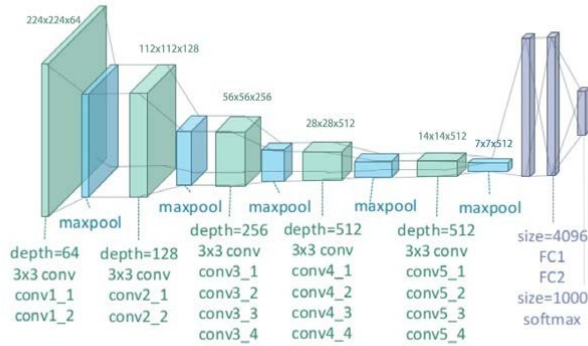


Figura 2.6: Ilustración de la arquitectura de la red VGG-19

sentido matemático), si la representación de una imagen no es única, la igualdad de las representaciones de salida no implicará la igualdad de las imágenes de entrada. Para garantizar un mapeo inyectivo, simplemente incluimos la imagen de entrada como un mapa de características adicional (la capa "cero" de la red). Entonces, la representación consta de la imagen de entrada x, concatenada con las respuestas de convolución de cinco capas VGG (etiquetadas conv1_2, conv2_2, conv3_3, conv4_3 y conv5_3) [1]:

$$f(x) = \{\tilde{x}_j^{(i)}; i = 0, ..., m; j = 1, ..., n_i\}, \tag{2.6}$$

donde $m = 5$ denota el número de capas de convolución elegidas para construir f, n_i es el número de mapas de características en la capa de convolución i-ésima, y $\tilde{x}(0) = x$.

Respecto a los algoritmos de optimización del gradiente, hay muchos que pueden ser utilizados en este tipo de redes. Sin embargo, como se explicará más tarde, se ha demostrado a base de pruebas que se prefiere un procedimiento de descenso de gradiente estocástico y, además, evita quedar atrapado en mínimos locales.

2.4. Modelar texturas

En este trabajo se va a utilizar un modelo de texturas similar al propuesto por Portilla y Simoncelli [1]. Estos autores propusieron que para generar una textura a partir de una imagen de origen dada realizamos los siguientes pasos. Primero, extraemos características de diferentes tamaños de la imagen de manera homogénea. A continuación, calculamos una estadística de resumen espacial sobre las características obtenidas en el primer paso para obtener una descripción de la imagen de origen. En este caso, usamos el conjunto de características proporcionado por una red neuronal profunda de alto rendimiento y como estadística de resumen espacial, las correlaciones entre características en cada capa de la red.

Para caracterizar una textura vectorizada dada x en nuestro modelo, primero pasamos x a través de la red neuronal convolucional y calculamos las activaciones en respuesta a x para cada capa l-ésima de la red. Estas forman un conjunto de imágenes respuesta llamadas mapas de características.

Una capa con N_l filtros distintos tiene N_l mapas de características cada uno de tamaño M_l cuando se vectoriza. Estos mapas de características se pueden almacenar en una matriz $F_l \in \mathbb{R}^{N_l \times M_l}$, donde F_{jk}^l es la activación del filtro j^{th} en la posición k-ésima de la capa l-ésima.

Estas correlaciones de características vienen dadas por la matriz de Gram $G^l \in \mathbb{R}^{N_l \times N_l}$, donde G_{ij}^l es el producto escalar entre los mapa de características i y j en la capa l-ésima [2]:

$$G_{ij}^l = \sum_k F_k^l F_{jk}^l \tag{2.7}$$

Así pues, el conjunto de matrices de Gram $G_1, G_2, ..., G_l$ de las capas $1, ..., L$ especifica completamente una textura en nuestro modelo.

2.5. Redes neuronales aplicadas a la transferencia

La transferencia de estilos tiene por objetivo sintetizar una textura a partir de una imagen de origen mientras, a la vez, se restringe la síntesis para preservar el contexto semántico de una imagen de destino. En un escenario perfecto, un algoritmo de transferencias de texturas debe ser capaz de extraer el contenido de la imagen, como pueden ser, los objetos que incluye un bodegón o los edificios que aparecen en un perfil urbano, y luego informar un procedimiento de transferencia de texturas para renderizar el contenido semántico de la imagen de destino en el estilo de la imagen de origen [2].

Un ejemplo visual de este procedimiento es el siguiente: escogemos transferir la textura de "La noche estrellada" de Van Gogh en otro retrato muy conocido: "La Monna Lisa" de Leonardo da Vinci y el resultado lo encotramos en la Figura 2.7.

Separar el contenido del estilo en las imágenes naturales sigue siendo un problema extremadamente difícil. Sin embargo, el avance de las CNNs profundas ha producido poderosos sistemas de visión por computadora que aprenden a extraer información semántica de alto nivel de imágenes naturales. Se demostró que las redes neuronales convolucionales entrenadas con suficientes datos etiquetados en tareas específicas, como el reconocimiento de objetos, aprenden a extraer contenido de imágenes de alto nivel en representaciones de características genéricas e incluso a otras tareas de procesamiento de información visual, incluido el reconocimiento de texturas, que en este caso nos ocupa.

Figura 2.7: Resultado de una transferencia de estilo con la técnica descrita por Gatys, Ecker y Bethge. En este caso, el estilo corresponde a ”La noche estrellada”de Vicent Van Gogh, y la finalidad de esta muestra es “van goghizar” la Monna Lisa.

Como hemos explicado previamente, en este caso vamos a representar las imágenes utilizando un espacio de características dado por una versión normalizada de las 16 capas de convolución y las 5 de *max-pooling* de la VGG-19. Dado que el modelo de textura también se basa en representaciones de imágenes profundas, el método de transferencia de estilo se reduce a un problema de optimización dentro de una red neuronal. El método de síntesis de texturas en una red neuronal VGG-19 se ilustra en la Figura 2.8 y se describe de la siguiente manera: en el procedimiento de análisis de texturas (izquierda), la textura original se pasa a través de la CNN y se calculan las matrices de Gram G_l, sobre las características respuesta de varias capas. En la síntesis de textura (derecha) se pasa una imagen de ruido $\hat{x}$ a través de la CNN y se calcula una función de pérdida en cada capa incluida en el modelo de textura, $E_l = \sum(\hat{G}^L - G^L)$, donde $\hat{G}^L$ es la matriz de Gram de la imagen que inicialmente era ruido blanco. La función de pérdida total L es una suma ponderada de las contribuciones E_l de cada capa. Utilizando el gradiente descendente sobre la pérdida total con respecto a la imagen original, se encuentra una nueva imagen que produce las mismas matrices de Gram, $\hat{G}^L$, que la textura original. Todo esto nos lleva al punto esencial de este procedimiento: cuando utilizamos el gradiente descendiente, estamos derivando respecto de la imagen original y no cambiando los pesos de la red neuronal, como es el procedimiento habitual en el aprendizaje profundo.

La primera capa convolucional tiene el mismo tamaño que la imagen y para las siguientes capas la relación entre los tamaños del mapa decaracterísticas permanece fija. Generalmente, cada capa de la red define un banco de filtros no lineales, cuya complejidad aumenta con la posición de la capa en la red [2].

Finalmente apuntar que los pesos son reescalados en la red de manera que la activación

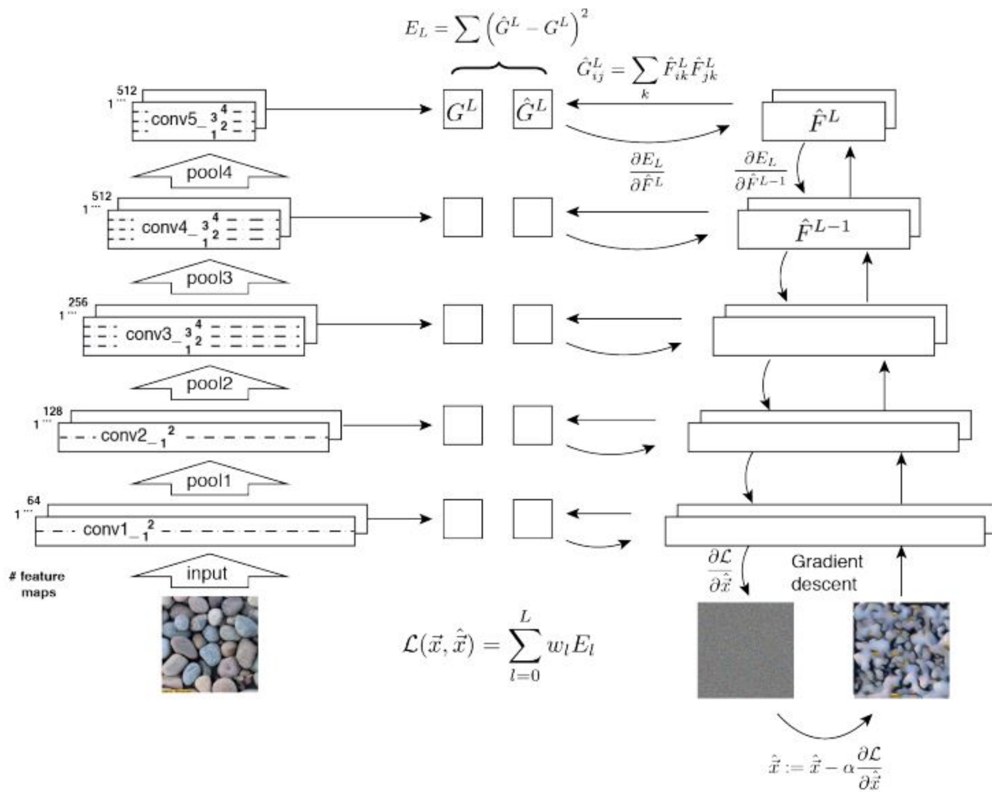


Figura 2.8: Método de síntesis de texturas en una red neuronal VGG-19. Análisis de texturas en la parte izquierda, síntesis de texturas en la parte derecha[2].

media de cada filtro sobre imágenes y posiciones sea igual a uno. Esto, consecuentemente, hace que no “toquemos” los pesos sino que estemos “entrenando”la imagen.

2.6. Transferencias de estilo mediante ruidificación

Como hemos visto en el capítulo anterior, *desacoplamiento vía normalización* se basa en encontrar una transformación invariante a lo largo del gradiente local, moviéndonos en trayectorias que siguen el gradiente de las características hasta encontrar los valores de referencia, previamente elegidos.

En nuestro caso, tenemos que tener en cuenta otro aspecto importante: para extraer las características de una imagen origen, y con ella normalizar la imagen destino, vamos a “ruidificar”, es decir, vamos a extraer las características de una imagen origen a través de imponer como valores de referencia los extraídos de una imagen de ruido uniforme $U(0, 1)$. De igual manera, para extraer el estilo de la imagen destino (y luego, poder imponer el de la imagen origen), seguiremos el mismo procedimiento: extraeremos las características mediante ruidificación.

En concreto, el proceso completo de transferencia de estilo mediante desacoplamiento de características está mostrado en la Figura 2.9 y formado por los siguientes pasos:

- Paso 0: medir las medias de los canales en una imagen de ruido, para utilizarlas como valores de referencia en las ruidificaciones.
- Paso 1: medir las características en la imagen origen para extraer su estilo, a través de forzarlo a tener las características del ruido, extraídas en el paso anterior.
- Paso 2: ruidificar la imagen destino, para despojarlo de sus propio estilo.
- Paso 3: transferir de las características de la imagen origen a la imagen destino.

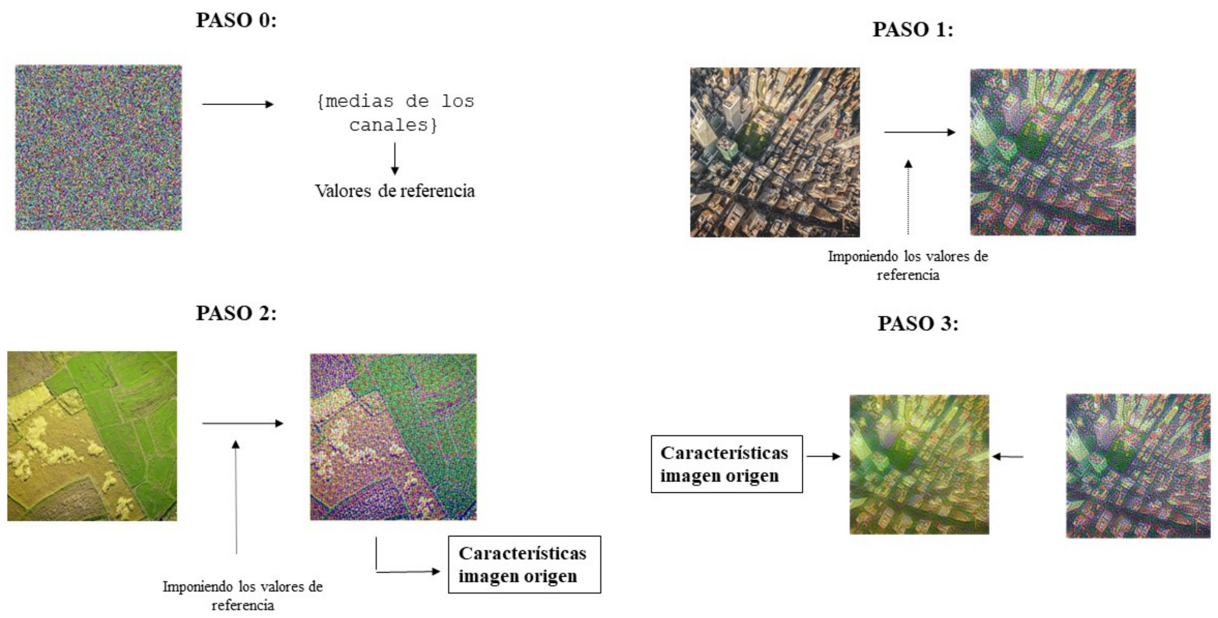


Figura 2.9: Ilustración del algoritmo de transferencia de estilo mediante desacoplamiento de característica

Capítulo 3

Contexto computacional

En este capítulo, explicaremos el algoritmo que hemos seguido para llevar a cabo la tarea que nos ocupa: dar con una metodologı́a robusta y eficiente para integrar los gradientes sin salirse de las correspondientes subvariedades y conseguir transferir el estilo de una imagen origen a una imagen destino o de una imagen a ella misma, es decir, extraer el estilo de una imagen y volvérselo a imponer, consiguiendo así un algoritmo de transferencia de estilo reversible.

En este trabajo, hemos utilizado el lenguaje de programación *Python* y, especialmente, su librería Tensorflow, una biblioteca de código abierto para aprendizaje automático desarrollado por Google. Con ella, podemos construir y entrenar redes neuronales para detectar y descifrar patrones y correlaciones, de una forma análoga al aprendizaje y razonamiento del cerebro humano [4].

3.1. Aspectos computacionales previos

Durante todos los pasos del algoritmo, hay una serie de funciones que vamos a utilizar y que trataremos de explicar en esta sección.

Librerías Durante la implementación del algoritmo se han utilizado, aparte de la librería Tensorflow, la librería *matplotlib*, para generar gráficos de calidad y la librería *numpy*, para trabajar con vectores y matrices grandes multidimensionales y operaciones de alto nivel. Además, se han importado otras funciones de librerías puntuales para ayudar a la comprensión del código como *torch* o *PIL.time*. En nuestro código, esto corresponde a:

```
import IPython.display as display
```

```
import matplotlib.pyplot as plt
import matplotlib as mpl
mpl.rcParams['figure.figsize'] = (12, 12)
mpl.rcParams['axes.grid'] = False

import numpy as np
import PIL.Image
import time
import functools

import tensorflow as tf
import torch
```

VGG-19 Durante todo nuestro código, vamos a utilizar la red neuronal VGG-19 implementada mediante la librería Tensorflow en el código:

```
def vgg_layers(layer_names):
    """ Creates a vgg model that returns a list of intermediate output values."""
    # Load our model. Load pretrained VGG, trained on imagenet data
    vgg = tf.keras.applications.vgg19(include_top=False, weights='imagenet')
    vgg.trainable = False

    outputs = [vgg.get_layer(name).output for name in layer_names]

    model = tf.keras.Model([vgg.input], outputs)
    return model
```

Clase StyleContentModel Definimos un nuevo objeto *class StyleContentModel* que define un modelo para extraer las medias espaciales de la salida del modelo de VGG en las capas seleccionadas. Cabe destacar que, pese a que vamos a definir en nuestro código cuales son las capas de convolución de la red neuronal que corresponden a la extracción de contenido, no vamos a tenerlo en cuenta en ningún momento durante nuestro procedimiento, pues solo nos interesa el estilo de las imágenes.

```
class StyleContentModel(tf.keras.models.Model):
    def __init__(self, style_layers, content_layers):
        super(StyleContentModel, self).__init__()
        self.vgg = vgg_layers(style_layers + content_layers)
        self.style_layers = style_layers
        self.content_layers = content_layers
        self.num_style_layers = len(style_layers)
        self.vgg.trainable = False
```

```
    def call(self, inputs):
        "Expects float input in [0,1]"
        inputs = inputs*255.0
        preprocessed_input = tf.keras.applications.vgg19.preprocess_input(inputs)
        outputs = self.vgg(preprocessed_input)

        style_outputs = outputs[:self.num_style_layers]
        print(style_outputs)
        print(outputs[:self.num_style_layers])

        style_outputs_mean = [mean_canal(capa_i) for capa_i in style_outputs]


        return style_outputs_mean
```

Cálculo de medias Definimos una función *mean_canal* usada en la clase *StyleContentModel* para calcular la media espacial de las capas (del tipo *tensor*) de la red neuronal que le pasamos como argumento.

```
def mean_canal(input_tensor):
    result = tf.math.reduce_mean(input_tensor, [1,2])
    return result
```

Optimizadores Como hemos apuntado en la sección de redes neuronales, el método de transferencia de estilo se reduce a un problema de optimización dentro de una red neuronal. Como se ha estado mencionando anteriormente, en nuestro estudio utilizamos el gradiente descendiente para derivar la imagen original y no para cambiar los pesos de la red neuronal. Este tipo de optimizadores (que, en realidad, siguen el mismo procedimiento para cualquier tipo de red neuronal, y es el usuario el que debe proporcionar la función que realmente necesita optimizar) están implementados en la librería *Tensorflow*. Durante la fase experimental se han probado todos los optimizadores que ofrece la librería, que son:

- `class Adadelta`: Optimizer que implementa el algoritmo Adadelta.
- `class Adagrad`: Optimizador que implementa el algoritmo Adagrad.
- `class Adam`: Optimizador que implementa el algoritmo Adam.
- `class Adamax`: Optimizador que implementa el algoritmo Adamax.
- `class Ftrl`: Optimizador que implementa el algoritmo FTRL.
- `class Nadam`: Optimizador que implementa el algoritmo NAdam.
- `class RMSprop`: Optimizador que implementa el algoritmo RMSprop.

- `class SGD`: Optimizador de descenso de gradiente.

Tras testar cada uno de ellos hemos elegido finalmente el optimizador del descenso del gradiente por varios motivos:

1. Los resultados obtenidos eran mejores siempre que la tasa de aprendizaje (en inglés, y más habitualmente, *learning rate*) fuera lo suficientemente pequeño.
2. Aunque se trata de un algoritmo más lento, en la literatura del *Deep Learning* se defiende su uso porque tiene mejor capacidad de generalización.

Así pues, definiremos el optimizador en el código de la siguiente manera:

```
opt = tf.optimizers.SGD(learning_rate=0.0001)
```

Main Incluye todas aquellas líneas de código necesarias para cualquier punto del algoritmo.

```
style_image = PIL.Image.open('imagen_origen.jpg')
```

(Aquí se incluirían unas funciones de reparametrización de las imágenes para ser adecuado como entrada de la red neuronal, y que estarán escritas en el código completo adjuntado en el Anexo).

```
style_layers = ['block1_conv1',
                'block2_conv1',
                'block3_conv1',
                'block4_conv1',
                'block5_conv1']

content_layers = ['block5_conv2']
```

3.2. Paso 0: medir las medias de los canales en una imagen de ruido.

Como hemos explicado en la sección 2.6, el primer paso a seguir es definir un algoritmo encargado de medir las características, en este caso, las medias espaciales, de los canales de la imagen de ruido uniforme. Estos corresponderán a los valores de referencia que más tarde querremos imponer en una imagen origen para recoger su estilo.

El código para realizar tal acción es el siguiente, donde la variable `valores_referencia` corresponde a las medias de los canales en una imagen de ruido:

```
extractor = StyleContentModel(style_layers, content_layers)
valores_referencia = extractor(ruido_uniforme)
```

Figura 3.1: Ejemplo de una imagen de ruido uniforme generada por la función `np.random.uniform(0,1, (1,200, 200, 3))`

En la Figura 3.1 podemos encontrar un ejemplo de una imagen de ruido uniforme generada por la función descrita en el código anterior.

3.3. Paso 1: medir las características en la imagen origen.

En este paso, vamos a medir las características en la imagen origen. Para ello, ruidificamos, es decir, surfeamos por el gradiente capa a capa hasta los valores de referencia, que en este caso y como hemos explicado en la sección anterior, pertenecen al ruido.

El procedimiento de este algoritmo es el siguiente:

1. Medimos las medias espaciales en la capa 1, y lo guardamos como un vector de características para usarlo posteriormente (c_1).
2. Ruidificamos (siguiendo la definición de ruidificar de la sección anterior) la capa 1 utilizando la función de pérdida como $(MSE)_i = \sum_{i_capa}^{1}(medias(x,i) - val_ref(i))^2$ y medimos

las características de la capa 2 (c_2), guardándolo para usarlo posteriormente como en el paso anterior.

3. Ruidificamos capa 2 manteniendo también la ruidificación en capa 1 moviéndonos por el gradiente de la 2ª capa proyectado sobre la 1ª, para no modificar el $(MSE)_1$ y medimos las características de la capa 3 (c_3).

4. Procedemos iterativamente hasta la 5ª capa de la red neuronal.

5. Finalmente obtenemos un vector de característcias $c = (c_1, ..., c_5)$, que en este caso es una estructura, pues $c_1, ..., c_n$ no se trata de un número, sino de un vector. Este c corresponden a las características que queremos imponer.

Todo esto, queda implementado en Python de la siguiente manera:

```
def style_content_loss(outputs, layer):
    loss = [tf.reduce_mean((outputs[lay]-valores_referencia[lay])**2)
        for lay in style_layers_ori[:layer+1]]
    return loss

images = [style_image.copy()]
gradients = []
layer_1 = style_layers[0]
extractor =  StyleContentModel(style_layers, content_layers)

Nits = 50000
num_layers = 5

outputs = extractor(style_image)
features=[]


features.append(outputs[layer_1])
image = tf.Variable(images[-1])


for layer in np.arange(num_layers):
  print(layer)
  loss = style_content_loss(outputs, layer)
  @tf.function()
  def train_step(image):
    with tf.GradientTape(persistent=True) as tape:
```

```
        outputs = extractor(image)['style']
        loss = style_content_loss(outputs, layer)

    grad = tape.gradient(loss, image)
    opt.apply_gradients([(grad, image)])

  for n in range(0,Nits):
        train_step(image)
        if np.mod(n, 200)==0:
            outputs = extractor(image)['style']
            loss = style_content_loss(outputs, layer)
            print(layer, n, np.array(loss))
            if(np.array(loss)[layer] <1):
              break
  if(layer<num_layers-1):
    features.append(outputs[style_layers[layer+1]])
```

3.4. Paso 2: ruidificar la imagen destino.

Para ruidificar la imagen destino, se sigue el mismo procedimiento que el paso anterior pero sin la necesidad guardar las características que extraemos, pues en este caso solo buscamos "desnudar" la imagen para poder "vestirla" con las características que hemos guardado en `features`. Además, en este caso podemos reutilizar la función de pérdida.

```
    style_image = PIL.Image.open('imagen_destino.png')
```

(Mismas funciones de parametrización para redes neuronales que en Main)

```
    images = [style_image.copy()]
    gradients = []

    outputs = extractor(style_image)
    features.append(tf.reduce_mean(outputs[layer_1]))
    image = tf.Variable(images[-1])

    for layer in np.arange(num_layers):
      loss = style_content_loss(outputs, layer)
      @tf.function()
```

```
def train_step(image):
    with tf.GradientTape() as tape:
        tape.watch(image)
        outputs = extractor(image)['style']
        loss = style_content_loss(outputs, layer)

    grad = tape.gradient(loss, image)
    opt.apply_gradients([(grad, image)])

for n in range(0,Nits):
      train_step(image)
      if np.mod(n, 5)==0:
          outputs = extractor(image)['style']
          loss = style_content_loss(outputs, layer)
          print(layer, n, np.array(loss):
          if(np.array(loss)[layer]<0.05):
            break
```

El bucle anterior, que se repetirá en todos los pasos de nuestro algoritmo completo, es la base del estudio teórico de este trabajo. Consideramos que es importante destacar que en el paso del entrenamiento, `train_step`, se aplica el gradiente a la imagen. En él, se modifica la imagen que quiero texturizar, no los pesos de la red neuronal.

3.5. Paso 3: transferimos a la imagen destino el estilo de la imagen origen.

Finalmente, vamos a juntar los dos pasos anteriores y vamos a transferir las características extraidas en `features` de la imagen origen a la imagen normalizada/ruidificada destino. En este caso, procederemos a realizar los mismos pasos que en los apartados anteriores pero en orden inverso, pues vamos a empezar ajustando las últimas capas, para acabar ajustando finalmente la primera. En este caso pues, deberemos redefinir la defición de pérdida para ajustar sobre las características y no sobre los valores de referencia, y seguir los siguientes pasos:

1. Ruidificamos (siguiendo la definición de ruidificar de la sección anterior) la capa n-ésima capa utilizando la función de pérdida como $(MSE)_i = \sum_{i_capa}^{1}(medias(x,i) - features(i))^2$ pero, este caso, para la capa actual y todas las anteriores hasta reducir esta pérdida a un valor cercano a 0.
2. Ruidificamos capa (n-1)-ésima manteniendo también la normalización de la capa n-ésima,

moviéndonos por el gradiente de la capa (n-1)-éssima proyectado sobre la (n-2)-ésima, para no modificar el $(MSE)_{n-2}$ y medimos las características de la capa 3 (c_3).

3. Procedemos iterativamente (de manera descendiente) hasta la 1ª capa de la red neuronal.

```
def loss_reverse(style_outputs, layer, features):
  style_loss = [tf.reduce_mean((style_outputs[lay]-
    features[style_layers_ori.index(lay)])**2)
    for lay in reversed(style_layers_ori[:layer+1])]
  return style_loss

outputs = extractor(image)['style']

for layer in range(num_layers-1, -1, -1):
  print(style_layers_ori[layer])
  loss = loss_reverse(outputs, layer, features)

  @tf.function()
  def train_step_reverse(image, features):
      with tf.GradientTape() as tape:
          tape.watch(image)
          outputs = extractor(image)['style']
          loss = loss_reverse(outputs, layer, features)

      grad = tape.gradient(loss, image)
      opt.apply_gradients([(grad, image)])

  for n in range(0,Nits):
        train_step_reverse(image, features)
        if np.mod(n, 5)==0:
            outputs = extractor(image)['style']
            loss = loss_reverse(outputs, layer, features)
            print(layer, n, np.array(loss))

            if(np.array(loss)[num_layers-1-layer]<0.5):
              break
```

Capítulo 4

Resultados

En este capítulo, vamos a exponer las imágenes resultado del código implementado y a mostrar y explicar diferentes gráficas que nos ayuden a entender el proceso de desacoplamiento que han seguido las características de la imagen. Además, se explicarán los problemas que han ido surgiendo durante las pruebas del código y las alternativas que se han probado para ponerles solución.

4.1. Explicaciones previas

En un escenario ideal, nuestro algoritmo sería capaz de normalizar las cinco capas de la red neuronal que tiene como entrada la imagen, ya sea la de origen para extraer sus características, como la de destino para imponérselas. Sin embargo, la implementación real de este algoritmo ha requerido de una gran cantidad de tiempo y uso de GPU y, en algunos casos, un entrenamiento perfecto (aquel que tiene una función de pérdida cercana a 0) era temporalmente inasumible.

Por ello, los resultados que vamos a mostrar a continuación se han generado usando el algoritmo presentado anteriormente (y expuesto al completo en el Anexo) para 2 capas de la red neuronal. Afortunadamente, estas capas trabajan las características que definen bordes, color y textura al trabajar solo con estas capas podemos obtener resultado de transferencia mucho más natural. Hemos distinguido dos tipos de prueba durante nuestros test:

- Una transferencia usual, tomando la textura de una imagen y transfiriéndola a otra con un contenido diferente.
- Una auto transferencia, es decir, extraer las características de una imagen (y con ella, su

estilo) para volvérsela a imponer mediante nuestro algoritmo.

Este último punto ha sido especialmente importante para ser capaces de distinguir visualmente los problemas que tiene el algoritmo, pues en un escenario perfecto, el resultado de la transferencia debería ser exactamente la imagen origen.

4.2. Resultado de la transferencia

4.2.1. Transferencia usual con 2 capas

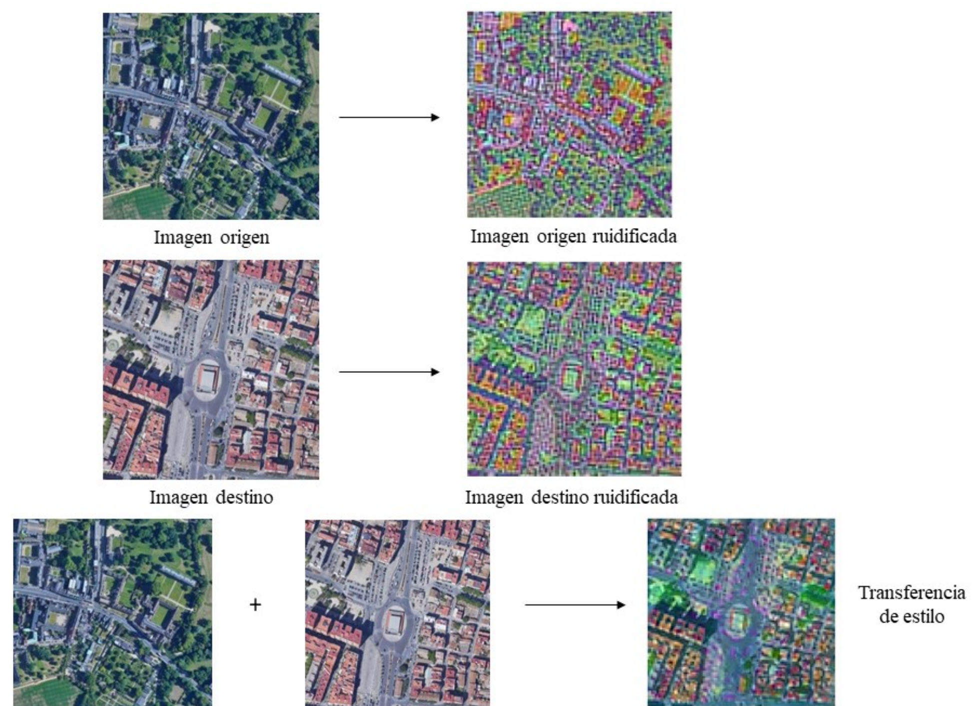


Figura 4.1: Transferencia de estilo mediante desacoplamiento de características mediante una red neuronal VGG-19 con 2 capas convolucionales. En este caso, extraemos el estilo de la ciudad de Oxford para transferírselo al Cabañal de Valencia.

4.2.2. Auto transferencia con 2 capas

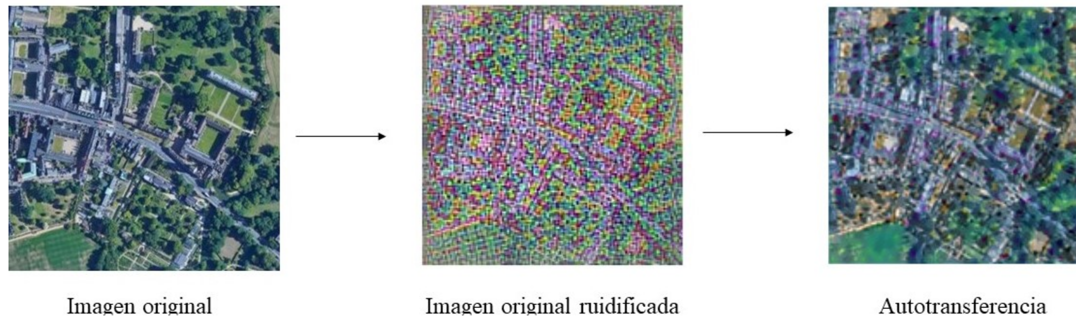


Figura 4.2: Transferencia de estilo mediante desacoplamiento de características mediante una red neuronal VGG-19 con 2 capas convolucionales. En este caso, 4.2.1 es la imagen origen y también la destino, por lo que 4.2.2 corresponde tanto al paso 1 como al 2 del algoritmo, y 4.2.3 es el resultado de la auto transferencia de estilo.

4.3. Gráficas de las funciones de pérdida.

En esta sección, vamos a presentar las gráficas de las funciones de pérdida de los diferentes pasos del algoritmo. Estas nacen de medir la función de pérdida que genera valores diferentes en cada época gracias al uso del optimizador del gradiente.

Gráficas de las funciones de pérdida de una transferencia usual con 2 capas

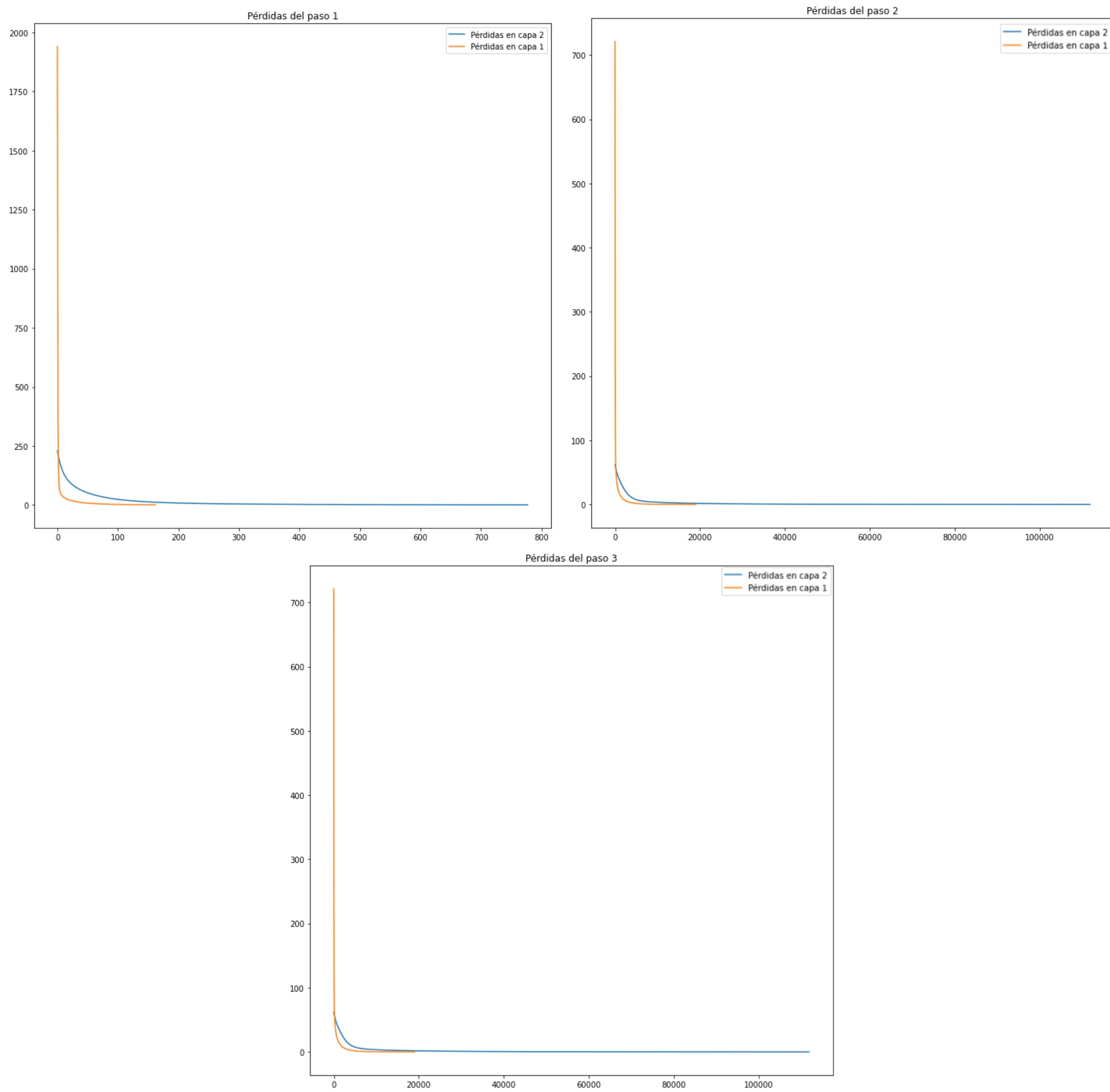


Figura 4.3: Gráficas de pérdidas de una transferencia de estilo mediante desacoplamiento de características mediante una red neuronal VGG-19 con 2 capas convolucionales.

Gráficas de las funciones de pérdida de una transferencia revertida con 2 capas

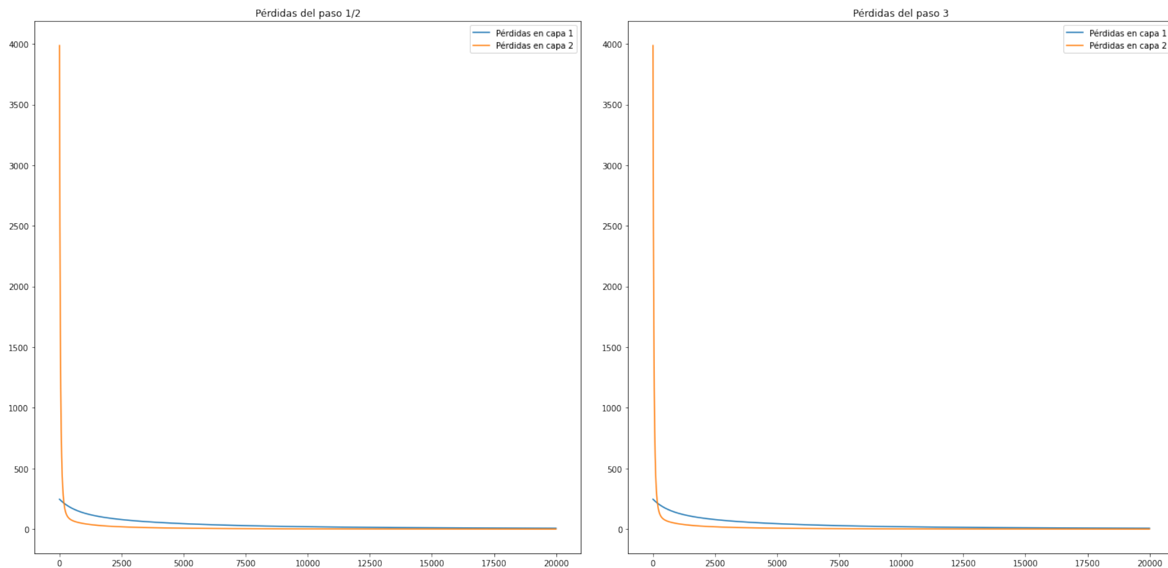


Figura 4.4: Gráficas de pérdidas de una auto transferencia de estilo mediante desacoplamiento de características mediante una red neuronal VGG-19 con 2 capas convolucionales. En este caso, al tratarse de una auto transferencia, se omite el paso 2 del algoritmo.

Tanto las gráficas de la transferencia de estilo habitual como las de la auto transferencia se tratan de gráficas del tipo “codo”, es decir, se puede observar que el uso el gradiente descendente sobre la función de pérdida tiene el comportamiento esperado, pues en todos los casos se están ajustando las imágenes correctamente hasta encontrar valores de pérdidas cercanos a 0.

4.4. Cómo se distribuyen las medias de los canales en una imagen de ruido

Durante el tiempo que ha durado el estudio de este algoritmo, se han ido observando minuciosamente los resultados que se iban extrayendo y se pudo observar que, al obtener las medias de los canales del ruido, hay canales cuya respuesta (y, por tanto, su media) tiende a cero (pudimos apreciar estas “apariciones” de ceros al representar uno a uno las medias de los canales en la Figura 4.5).

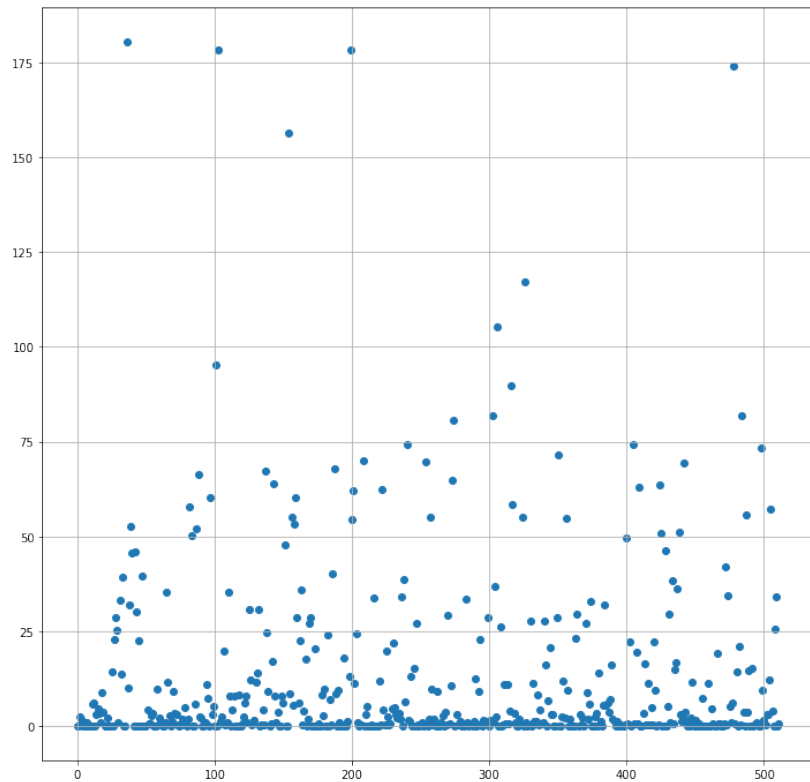


Figura 4.5: Representación de las medias de los canales, en este caso en la capa 5, donde el eje de las x corresponde simplemente a la posición del canal y el eje de las y el valor de la media de ese canal.

Esta singularidad "mata" el gradiente, y no podemos tomar su valor ni como punto de llegada ni como punto de partida, pues la variedad de referencia tiene que estar incluida en Ω, que no incluye puntos críticos, y x_0 también debe pertenecer a Ω para que el algoritmo funcione.

Analizamos el comportamiento de una imagen de ruido usando la aplicación desarrollada en [12]. Se introduce en la aplicación la imagen de ruido en una CNN preentrenada para identificar mariquitas, pizzas, pimientas, autobuses escolares, koalas, cafés expresos, pandas rojos, naranjas y coches deportivos, y se obtienen los mapas de características (Figura 4.6).

Como se puede observar, no hay ninguna capa de convolución que de resultados comprensibles. Eso es, la CNN busca una serie de características (que en este caso, la identifique con uno de esos objetos) que en una imagen de ruido es incapaz de encontrar. Eso genera que en una de esas características como es la media, de valores 0. Por tanto, estos ceros no deben ser tenidos en cuenta, pues en realidad, entran dentro de lo esperado.

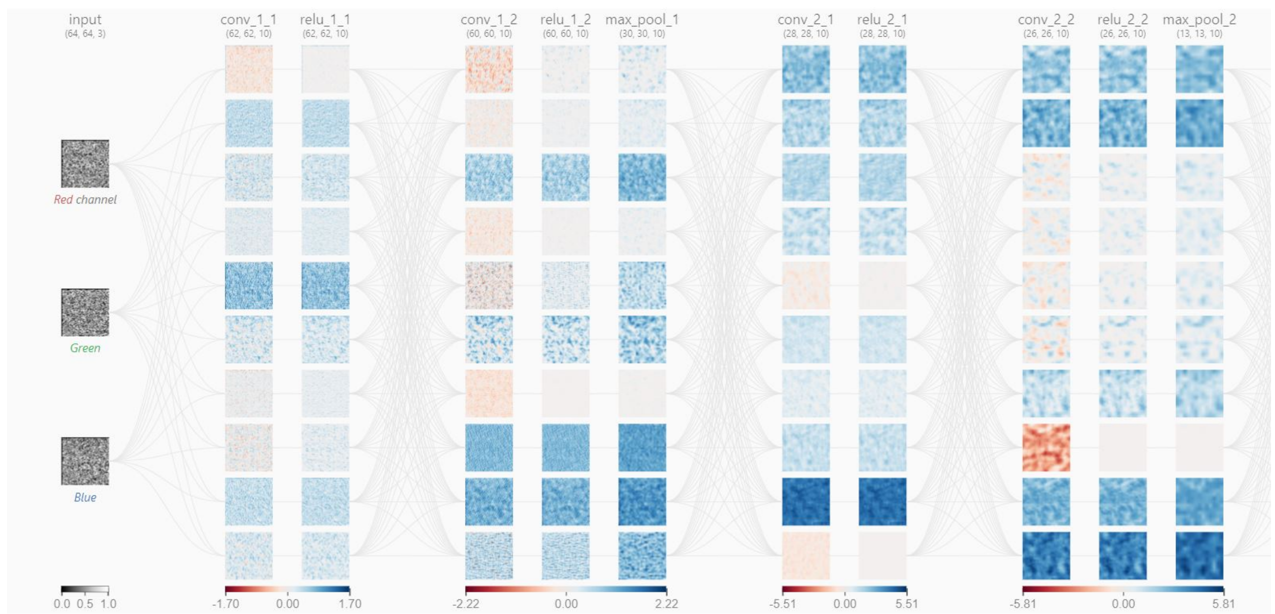


Figura 4.6: Activación de los mapas de características para cada capa de convolución de una CNN al introducir una imagen de ruido como la usada en nuestra red.

Capítulo 5

Conclusiones

Este trabajo contiene una recopilación de los conceptos fundamentales del desacoplamiento de características y su aplicación a la transferencia de estilos. Empezando por la fundamentación teórica, definimos los conceptos de característica y de desacoplamiento y presentamos un formalismo para desacoplar características deterministas. Además, hacemos un estudio básico sobre redes neuronales y exponemos por qué la red neuronal convolucional VGG-19 es idónea para afrontar la tarea de trasferir estilos. Finalmente, presentamos un código que implementa todo lo fundamentado teóricamente y mostramos los resultados obtenidos.

En general, los resultados obtenidos son satisfactorios: tanto visualmente como mediante los resultados numéricos podemos detectar que las transferencias son de calidad.

Como trabajo futuro, sería interesante implementar el algoritmo Gram-Schmidt para conseguir una ortogonalización más precisa de las características. Además, también sería interesante utilizar el método de Runge-Kutta para integrar los gradientes sin salirse (ni ligeramente) de las correspondientes subvariedades.

Además, el objetivo final siempre será poder ejecutar el algoritmo utilizando los 5 niveles de profundidad, es decir, las 5 capas. Por ello, se podría encontrar implementación más eficiente que permitiera esta circustancia.

Bibliografía

[1] Keyan Ding, Kede Ma, Shiqi Wang, and Eero P Simoncelli. Image quality assessment: Unifying structure and texture similarity. *IEEE transactions on pattern analysis and machine intelligence*, 2020.

[2] Leon Gatys, Alexander S Ecker, and Matthias Bethge. Texture synthesis using convolutional neural networks. *Advances in neural information processing systems*, 28, 2015.

[3] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. Image style transfer using convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2414–2423, 2016.

[4] Google. Tensorflow: Open source machine learning, 2015.

[5] Trevor Hastie, Robert Tibshirani, Jerome H Friedman, and Jerome H Friedman. *The elements of statistical learning: data mining, inference, and prediction*, volume 2. Springer, 2009.

[6] Bela Julesz. Visual pattern discrimination. *IRE transactions on Information Theory*, 8(2):84–92, 1962.

[7] Eduardo Martinez-Enriquez, Maria del Mar Gonzalez, and Javier Portilla. Deterministic decoupling of global features and its application to data analysis. 2022.

[8] Jonathan Masci, Ueli Meier, Dan Ciresan, Jürgen Schmidhuber, and Gabriel Fricout. Steel defect classification with max-pooling convolutional neural networks. In *The 2012 international joint conference on neural networks (IJCNN)*, pages 1–6. IEEE, 2012.

[9] Javier Portilla and Eero P Simoncelli. A parametric texture model based on joint statistics of complex wavelet coefficients. *International journal of computer vision*, 40(1):49–70, 2000.

[10] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[11] Marcelo Bertalmío Trevor D. Canham, Adrián Martín and Javier Portilla. Using decoupled features for photo-realistic style transfer. *Proceedings of the IEEE*, 2022.

[12] Zijie J Wang, Robert Turko, Omar Shaikh, Haekyu Park, Nilaksh Das, Fred Hohman, Minsuk Kahng, and Duen Horng Polo Chau. Cnn explainer: learning convolutional neural networks with interactive visualization. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):1396–1406, 2020.

[13] Yufeng Zheng, Clifford Yang, and Alex Merkulov. Breast cancer screening using convolutional neural network and follow-up digital mammography. In *Computational Imaging III*, volume 10669, page 1066905. SPIE, 2018.

Anexo A: implementación del algoritmo mediante Python

July 10, 2022

LIBRERÍAS

[47]:
```
import IPython.display as display

import matplotlib.pyplot as plt
import matplotlib as mpl
mpl.rcParams['figure.figsize'] = (12, 12)
mpl.rcParams['axes.grid'] = False

import numpy as np
import PIL.Image
import time
import functools
import torch
import tensorflow as tf
```

MAIN

[48]:
```
style_image = PIL.Image.open('imagen_origen.png')
Imsize = 150
```

Funciones de reparametrización de las imágenes para ser adecuado como entrada de la red neuronal

[49]:
```
style_image = np.asarray(style_image.resize((Imsize, Imsize)))
style_image = style_image[:,:,0:3].astype(np.float32)/255
style_image = np.expand_dims(style_image,0)
```

[50]:
```
plt.figure()
plt.imshow(style_image[0, :,:,:],vmin=0,vmax=1), plt.axis('off')
plt.title('Imagen Origen')
plt.show()
```

Imagen Origen

[51]:
```
style_layers_ori = ['block1_conv1',
                'block2_conv1',
                'block3_conv1',
                'block4_conv1',
                'block5_conv1']

content_layers = ['block5_conv2'] #No lo estamos usando, pero hay que definirlas
style_layers = style_layers_ori[0:5]
```

[52]:
```
#opt = tf.optimizers.Adam(learning_rate=0.001, beta_1=0.99, epsilon=1e-1)
opt = tf.optimizers.SGD(learning_rate=0.0001)
#opt = tf.keras.optimizers.Adadelta(learning_rate=0.0001, rho=0.95, epsilon=1e-07)
```

[53]:
```
def vgg_layers(layer_names):
    """ Creates a vgg model that returns a list of intermediate output values."""
    # Load our model. Load pretrained VGG, trained on imagenet data
    vgg = tf.keras.applications.VGG19(include_top=False, weights='imagenet')
    vgg.trainable = False

    outputs = [vgg.get_layer(name).output for name in layer_names]

    model = tf.keras.Model([vgg.input], outputs)
  # imprimeix el model que anem a gastar , nom de les capes.....
    #print(model.summary())
    return model
```

[54]:
```
def mean_canal(input_tensor):
    result = tf.math.reduce_mean(input_tensor, [1,2])
    return result
```

[55]:
```
class StyleContentModel(tf.keras.models.Model):
  # el _init_ siempre tiene que estar
    def __init__(self, style_layers, content_layers):
        super(StyleContentModel, self).__init__()
        self.vgg = vgg_layers(style_layers + content_layers)
        self.style_layers = style_layers
        self.content_layers = content_layers
        self.num_style_layers = len(style_layers)
        self.vgg.trainable = False

    def call(self, inputs):
        "Expects float input in [0,1]"
        inputs = inputs*255.0
        preprocessed_input = tf.keras.applications.vgg19.preprocess_input(inputs)
        outputs = self.vgg(preprocessed_input)
        # salida del modelo de vgg, en las capas seleccionadas
        style_outputs, content_outputs = (outputs,
                                          outputs[self.num_style_layers:])


        # Resumimos la información espacial: elimina la dependencia espacial
        style_outputs = [mean_canal(capa_i) for capa_i in style_outputs]
        content_dict = {content_name: value
                        for content_name, value
                        in zip(self.content_layers, content_outputs)}
        # dim style_output: (64, 128, 256, 512, 512)
        style_dict = {style_name: value
                      for style_name, value
                      in zip(self.style_layers, style_outputs)}
        return {'content': content_dict, 'style': style_dict, }#'sd': style_outputs_sd}
```

PASO 0: Extracción de los valores de referencia
Medir las medias de los canales en una imagen de ruido

[56]:
```
ruido_uniforme=np.random.uniform(0,1, (1,200, 200, 3))
plt.imshow(ruido_uniforme[0, :,:,:],vmin=0,vmax=1), plt.axis('off')
```

[56]:
```
(<matplotlib.image.AxesImage at 0x7f4fa0022a10>, (-0.5, 199.5, 199.5, -0.5))
```

[57]:
```
extractor = StyleContentModel(style_layers, content_layers)
valores_referencia = extractor(ruido_uniforme)
```

PASO 1: Medir las características en la imagen origen.

[64]:
```
def style_content_loss(outputs, layer):
    loss = [tf.reduce_mean((outputs[lay]-valores_referencia[lay])**2) for lay in␣
 ↪style_layers_ori[:layer+1]]
    return loss
```

[61]:
```
images = [style_image.copy()]
gradients = []
layer_1 = style_layers_ori[0]
extractor =  StyleContentModel(style_layers, content_layers)
```

[66]:
```
Nits = 50000
num_layers = 5
```

[67]:
```
outputs = extractor(style_image)['style']
features=[]
features.append(outputs[layer_1])
image = tf.Variable(images[-1])

#INSTRUCCIONES PARA GENERAR GRÁFICAS
L_plot_paso_1=[]

for layer in np.arange(num_layers):
  print(layer)
  ###############################
  # DERIVACION y el descenso por gradiente
  ###############################
  loss = style_content_loss(outputs, layer)
  @tf.function()  # Precompile
  def train_step(image):
    with tf.GradientTape(persistent=True) as tape:
        outputs = extractor(image)['style']
        loss = style_content_loss(outputs, layer)

    grad = tape.gradient(loss, image)
    opt.apply_gradients([(grad, image)])
  L_plot_paso_1_capa_i=[]

  for n in range(0,Nits):
        train_step(image)
        if np.mod(n, 200)==0:
            outputs = extractor(image)['style']
            loss = style_content_loss(outputs, layer)
            print(layer, n, np.array(loss))
            L_plot_paso_1_capa_i.append(np.array(loss[layer]))
            if(np.array(loss)[layer] <1):
              break
  if(layer<num_layers-1):
    features.append(outputs[style_layers_ori[layer+1]])
  loss_anterior = loss
```

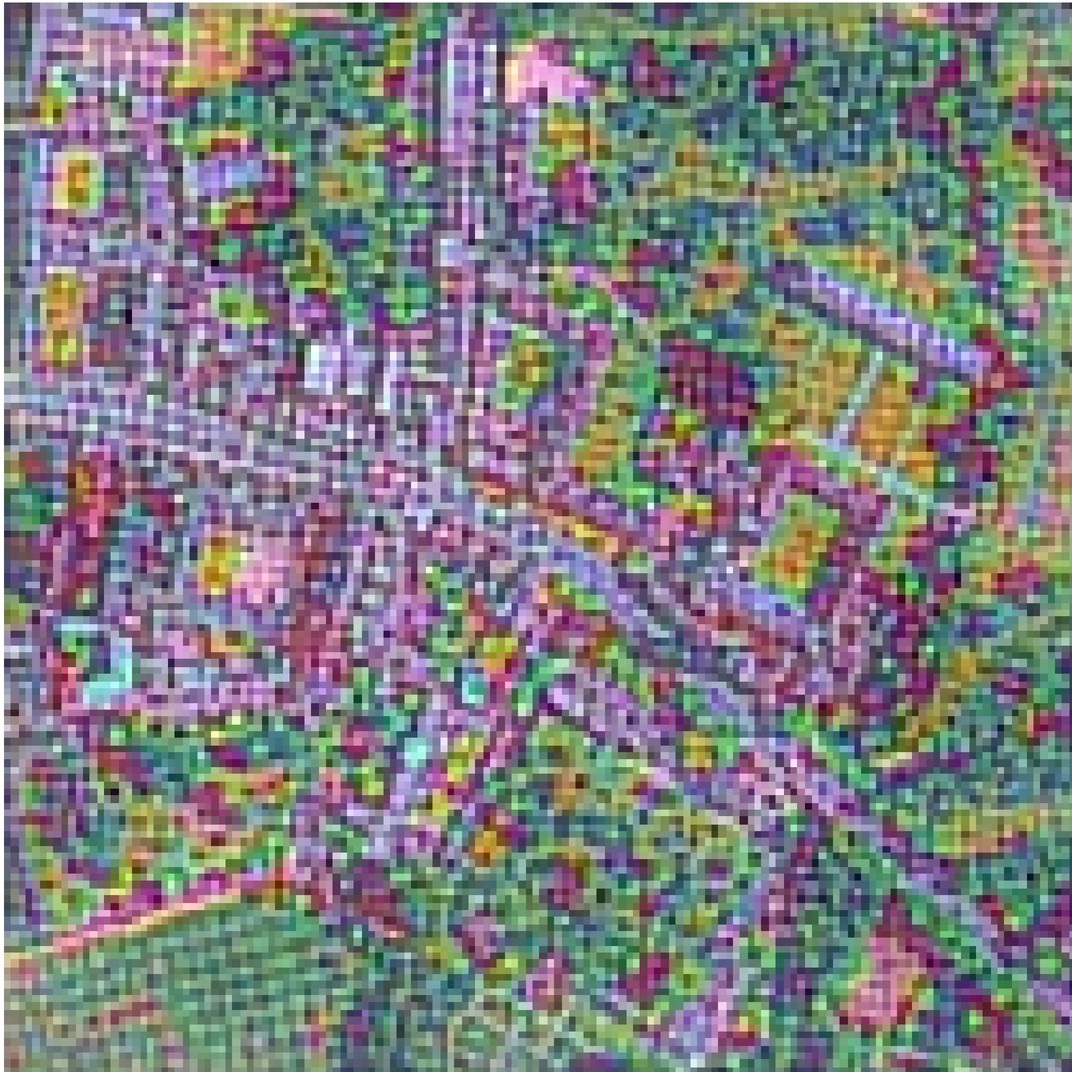
```
  L_plot_paso_1.append(L_plot_paso_1_capa_i)



plt.figure()
plt.imshow(image[0, :,:,:],vmin=0,vmax=1), plt.axis('off')
plt.show()
```

```
    Se han truncado las últimas 100 líneas del flujo de salida.
1 29800 [1.7149916 5.8408947]
1 30000 [1.7040523 5.772869 ]
1 30200 [1.693248  5.7059965]
1 30400 [1.6825751 5.6401205]
1 30600 [1.6720324 5.5752892]
1 30800 [1.6616219 5.5114937]
1 31000 [1.6513726 5.4486012]
1 31200 [1.6412967 5.386547 ]
1 31400 [1.6313691 5.3255024]
1 31600 [1.6215782 5.2653656]
1 31800 [1.6119046 5.206102 ]
1 32000 [1.6023564 5.1477776]
1 32200 [1.5929301 5.0903144]
1 32400 [1.5836246 5.03372  ]
1 32600 [1.5744424 4.9779367]
1 32800 [1.5653747 4.923025 ]
1 33000 [1.5564488 4.868909 ]
1 33200 [1.5476389 4.81553  ]
1 33400 [1.5389314 4.763002 ]
1 33600 [1.5303309 4.7111816]
1 33800 [1.5218449 4.6600566]
1 34000 [1.5134488 4.609659 ]
1 34200 [1.5051621 4.5599217]
1 34400 [1.4969777 4.5108967]
1 34600 [1.4889078 4.4625278]
1 34800 [1.4809296 4.4148703]
1 35000 [1.4730394 4.367954 ]
1 35200 [1.4652497 4.32168  ]
1 35400 [1.4575696 4.2760773]
1 35600 [1.45      4.231123]
1 35800 [1.4425464 4.1867523]
1 36000 [1.4351897 4.143019 ]
1 36200 [1.4279255 4.0998974]
1 36400 [1.4207339 4.057317 ]
1 36600 [1.413611 4.015306]
1 36800 [1.4065697 3.973785 ]
1 37000 [1.39959   3.9328027]
1 37200 [1.3926812 3.892335 ]
1 37400 [1.3858448 3.8524117]
1 37600 [1.3790948 3.8129911]
1 37800 [1.3724463 3.7740617]
1 38000 [1.3658857 3.7356086]
1 38200 [1.359394  3.6976497]
1 38400 [1.3529844 3.6602724]
1 38600 [1.3466375 3.6234279]
```

```
1 38800 [1.3403468 3.5870914]
1 39000 [1.334122  3.5512664]
1 39200 [1.3279736 3.5158844]
1 39400 [1.3218963 3.4809048]
1 39600 [1.3158895 3.446397 ]
1 39800 [1.3099592 3.4123235]
1 40000 [1.304102  3.3786516]
1 40200 [1.2983048 3.3453283]
1 40400 [1.2925668 3.3124566]
1 40600 [1.2869014 3.2800498]
1 40800 [1.2813122 3.24798  ]
1 41000 [1.2757845 3.216299 ]
1 41200 [1.270306 3.185014]
1 41400 [1.2648772 3.1541247]
1 41600 [1.2595134 3.123665 ]
1 41800 [1.2542126 3.093604 ]
1 42000 [1.2489663 3.063924 ]
1 42200 [1.2437731 3.0346246]
1 42400 [1.2386268 3.0056496]
1 42600 [1.233539  2.9770412]
1 42800 [1.2284997 2.94879  ]
1 43000 [1.2235043 2.9208963]
1 43200 [1.2185719 2.8933535]
1 43400 [1.2136881 2.8661704]
1 43600 [1.2088652 2.8393192]
1 43800 [1.2040863 2.8128424]
1 44000 [1.1993532 2.7866824]
1 44200 [1.1946542 2.7608192]
1 44400 [1.1899933 2.735321 ]
1 44600 [1.1853857 2.7101305]
1 44800 [1.1808252 2.685256 ]
1 45000 [1.1762974 2.6606834]
1 45200 [1.1718035 2.636403 ]
1 45400 [1.1673486 2.6124213]
1 45600 [1.1629436 2.5887136]
1 45800 [1.1585767 2.5653002]
1 46000 [1.1542482 2.5421524]
1 46200 [1.1499499 2.5192535]
1 46400 [1.1457    2.4966333]
1 46600 [1.1415039 2.4742908]
1 46800 [1.1373506 2.4522204]
1 47000 [1.1332355 2.4303591]
1 47200 [1.1291552 2.4087262]
1 47400 [1.1251035 2.3873296]
1 47600 [1.1210984 2.366188 ]
1 47800 [1.1171322 2.345299 ]
1 48000 [1.1131964 2.3246694]
1 48200 [1.1092966 2.3042598]
1 48400 [1.105442 2.284083]
1 48600 [1.1016104 2.2641468]
1 48800 [1.097805  2.2444272]
1 49000 [1.0940287 2.2249367]
1 49200 [1.0902815 2.205644 ]
1 49400 [1.0865645 2.1865416]
```

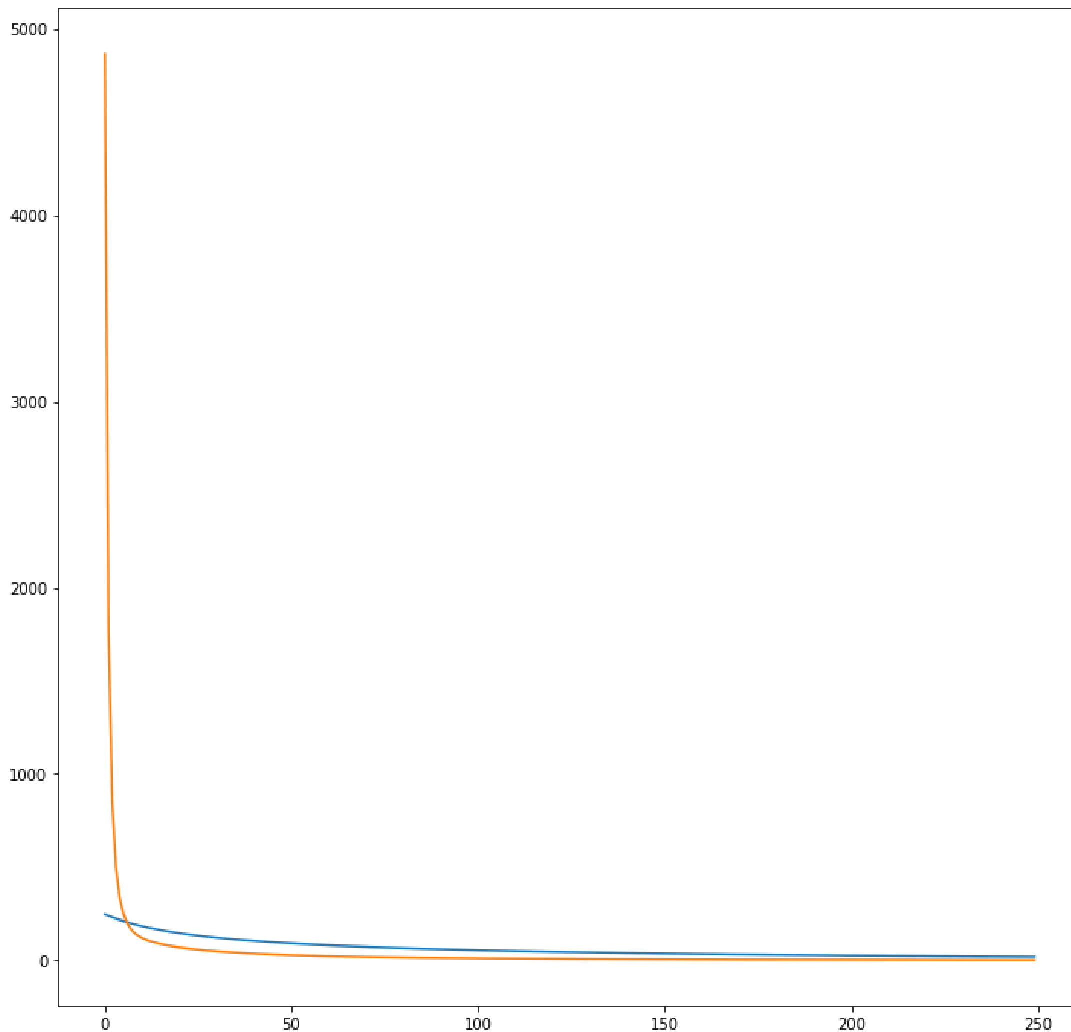
```
1 49600 [1.0828719 2.167626 ]
1 49800 [1.0792098 2.1489007]
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```



[68]:

```
plt.figure()
for i in range(num_layers):
  plt.plot(L_plot_paso_1[i])
plt.show()
```



*PASO 2: Medir las caracter ısticas en la imagen origen.**

[69]:
```
style_image = PIL.Image.open('imagen_destino.png')
style_image = np.asarray(style_image.resize((Imsize, Imsize)))
style_image = style_image[:,:,0:3].astype(np.float32)/255
style_image = np.expand_dims(style_image,0)

images = [style_image.copy()]
gradients = []
```

[70]:
```
outputs = extractor(style_image)['style']
features.append(tf.reduce_mean(outputs[layer_1]))
image = tf.Variable(images[-1])
L_plot_paso_2=[]
```

```
for layer in np.arange(num_layers):
  ###############################
  # DERIVACION y el descenso por gradiente
  ###############################
  loss = style_content_loss(outputs, layer)#esto es lo que ha cambiado, la definición␣
↪del loss
  @tf.function()  # Precompile
  def train_step(image):
      with tf.GradientTape() as tape:
          tape.watch(image)
          outputs = extractor(image)['style']
          loss = style_content_loss(outputs, layer)

      grad = tape.gradient(loss, image)
      #gram-smicht
      #grad_gs = gram_schmidt(grad)
      opt.apply_gradients([(grad, image)])

  L_plot_paso_2_capa_i=[]
  for n in range(0,Nits):
        train_step(image)
        if np.mod(n, 5)==0:
            outputs = extractor(image)['style']
            loss = style_content_loss(outputs, layer)
            print(layer, n, np.array(loss))
            L_plot_paso_2_capa_i.append(np.array(loss[layer]))
            if(np.array(loss)[layer]<0.05):
              break
  L_plot_paso_2.append(L_plot_paso_2_capa_i)

plt.figure()
plt.imshow(image[0, :,:,:],vmin=0,vmax=1), plt.axis('off')
plt.show()

# Curvas
plt.figure()
for i in range(num_layers):
  plt.plot(L_plot_paso_2[i])
plt.show()
```

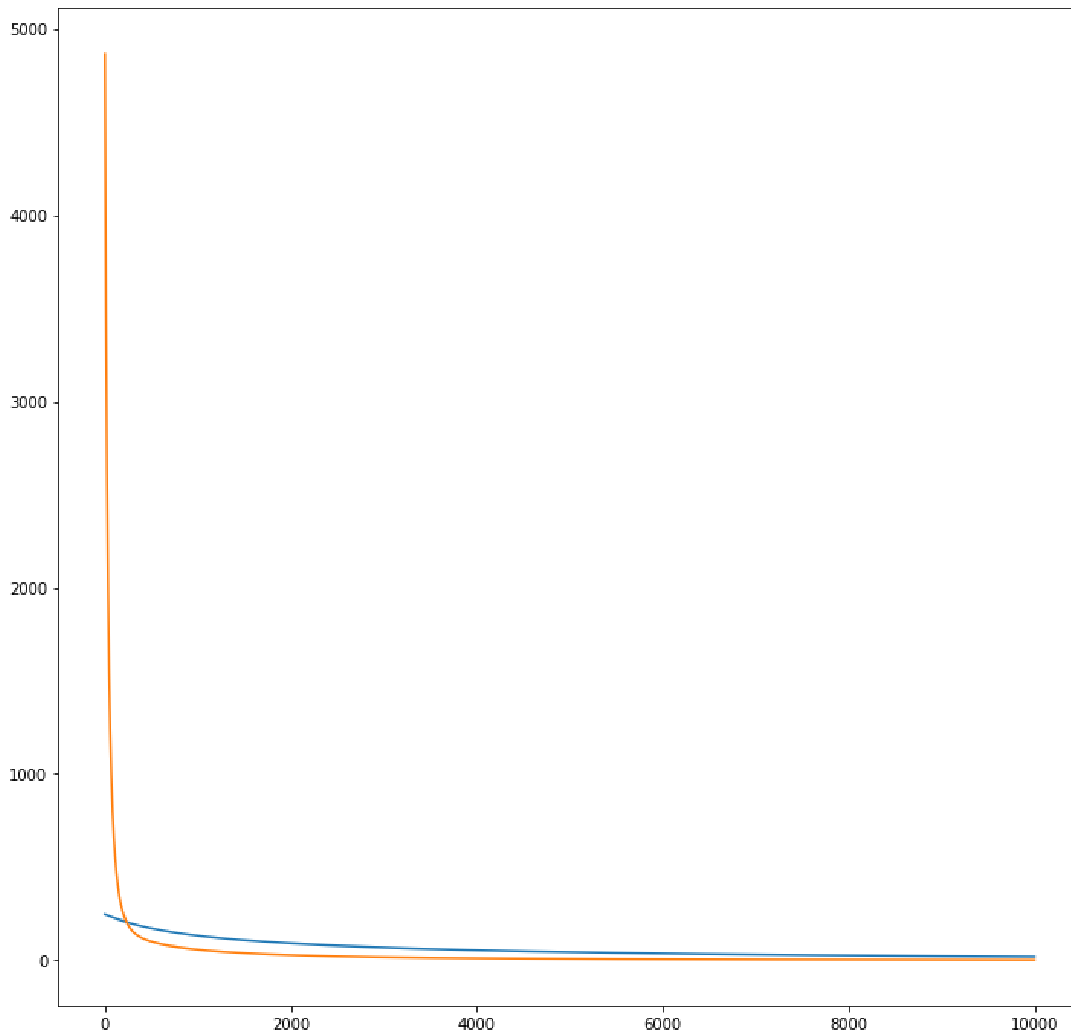
```
Se han truncado las últimas 100 líneas del flujo de salida.
1 49480 [1.0850847 2.1789532]
1 49485 [1.084993  2.1784787]
1 49490 [1.0848999 2.1780097]
1 49495 [1.0848081 2.1775374]
1 49500 [1.0847154 2.177062 ]
1 49505 [1.0846227 2.1765924]
1 49510 [1.0845308 2.1761174]
1 49515 [1.0844381 2.1756473]
1 49520 [1.0843468 2.175174 ]
1 49525 [1.0842534 2.1747065]
1 49530 [1.0841612 2.174233 ]
1 49535 [1.0840693 2.1737614]
1 49540 [1.0839765 2.1732893]
```

```
1 49545 [1.083885  2.1728153]
1 49550 [1.0837932 2.1723433]
1 49555 [1.0837005 2.1718721]
1 49560 [1.083608  2.1713984]
1 49565 [1.0835164 2.1709318]
1 49570 [1.0834239 2.1704612]
1 49575 [1.0833316 2.1699908]
1 49580 [1.0832403 2.1695151]
1 49585 [1.0831476 2.169046 ]
1 49590 [1.0830556 2.1685736]
1 49595 [1.0829635 2.1681004]
1 49600 [1.0828719 2.167626 ]
1 49605 [1.08278  2.167163]
1 49610 [1.0826886 2.16669  ]
1 49615 [1.0825963 2.1662195]
1 49620 [1.0825036 2.1657445]
1 49625 [1.082412  2.1652792]
1 49630 [1.0823197 2.1648057]
1 49635 [1.0822287 2.1643374]
1 49640 [1.0821359 2.1638706]
1 49645 [1.0820435 2.1633987]
1 49650 [1.0819526 2.1629293]
1 49655 [1.0818601 2.1624608]
1 49660 [1.081769  2.1619904]
1 49665 [1.0816772 2.1615176]
1 49670 [1.0815852 2.161049 ]
1 49675 [1.0814939 2.1605825]
1 49680 [1.0814022 2.160112 ]
1 49685 [1.081311  2.1596427]
1 49690 [1.0812193 2.159174 ]
1 49695 [1.0811284 2.1587057]
1 49700 [1.081036  2.1582372]
1 49705 [1.0809443 2.1577702]
1 49710 [1.0808531 2.1573024]
1 49715 [1.0807619 2.1568327]
1 49720 [1.0806699 2.1563666]
1 49725 [1.0805796 2.155898 ]
1 49730 [1.0804875 2.1554291]
1 49735 [1.0803953 2.154961 ]
1 49740 [1.0803041 2.154495 ]
1 49745 [1.0802131 2.1540294]
1 49750 [1.0801215 2.153561 ]
1 49755 [1.0800307 2.1530907]
1 49760 [1.0799395 2.1526291]
1 49765 [1.0798479 2.152161 ]
1 49770 [1.0797565 2.1516948]
1 49775 [1.0796658 2.1512303]
1 49780 [1.0795739 2.1507652]
1 49785 [1.0794827 2.150298 ]
1 49790 [1.0793926 2.1498294]
1 49795 [1.0793016 2.149362 ]
1 49800 [1.0792098 2.1489007]
1 49805 [1.079118  2.1484346]
1 49810 [1.0790274 2.14797  ]
```

```
1 49815 [1.0789363 2.147504 ]
1 49820 [1.0788463 2.1470387]
1 49825 [1.0787549 2.1465774]
1 49830 [1.0786643 2.1461072]
1 49835 [1.0785723 2.1456437]
1 49840 [1.078482  2.1451821]
1 49845 [1.0783908 2.1447186]
1 49850 [1.0782999 2.1442537]
1 49855 [1.0782082 2.1437883]
1 49860 [1.0781181 2.1433287]
1 49865 [1.0780272 2.142864 ]
1 49870 [1.0779371 2.1424012]
1 49875 [1.0778458 2.1419334]
1 49880 [1.0777545 2.14147  ]
1 49885 [1.0776644 2.141008 ]
1 49890 [1.0775737 2.1405427]
1 49895 [1.0774829 2.1400812]
1 49900 [1.0773921 2.1396208]
1 49905 [1.0773021 2.1391559]
1 49910 [1.0772107 2.1386905]
1 49915 [1.0771201 2.1382294]
1 49920 [1.0770302 2.137766 ]
1 49925 [1.0769384 2.1373038]
1 49930 [1.0768485 2.1368423]
1 49935 [1.076757  2.1363835]
1 49940 [1.0766665 2.1359186]
1 49945 [1.0765758 2.135455 ]
1 49950 [1.0764854 2.134994 ]
1 49955 [1.0763948 2.1345308]
1 49960 [1.0763043 2.134069 ]
1 49965 [1.0762131 2.1336129]
1 49970 [1.0761229 2.133153 ]
1 49975 [1.0760324 2.1326883]
1 49980 [1.0759417 2.1322272]
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```

```
1 49985 [1.0758514 2.1317701]
1 49990 [1.0757608 2.1313071]
1 49995 [1.0756705 2.130844 ]
```



PASO 3: Transferimos a la imagen destino el estilo de la imagen origen.

[71]:
```
def loss_reverse(style_outputs, layer, features):
  style_loss = [tf.reduce_mean((style_outputs[lay]-features[style_layers_ori.
 ↪index(lay)])**2) for lay in reversed(style_layers_ori[:layer+1])]
  return style_loss
```

[74]:
```
outputs = extractor(image)['style']
L_plot_paso_3=[]

for layer in range(num_layers-1, -1, -1):
  print(style_layers_ori[layer])
  loss = loss_reverse(outputs, layer, features)

  @tf.function()   # Precompile
```

```
def train_step_reverse(image, features):
    with tf.GradientTape() as tape:
        tape.watch(image)
        outputs = extractor(image)['style']
        loss = loss_reverse(outputs, layer, features)

    grad = tape.gradient(loss, image)
    #gram-smicht
    #grad_gs = gram_schmidt(grad)
    opt.apply_gradients([(grad, image)])
#train_step(image)
L_plot_paso_3_capa_i=[]

for n in range(0,Nits):
      train_step_reverse(image, features)
      if np.mod(n, 5)==0:
          outputs = extractor(image)['style']
          loss = loss_reverse(outputs, layer, features)
          print(layer, n, np.array(loss))
          L_plot_paso_3_capa_i.append(np.array(loss[layer]))

          #if(np.array(loss)[num_layers-1-layer]<0.5):
          #  break
L_plot_paso_3.append(L_plot_paso_3_capa_i)
```

```
Se han truncado las últimas 5000 líneas del flujo de salida.
0 49495 [1.3210123]
0 49500 [1.3207626]
0 49505 [1.3205155]
0 49510 [1.3202662]
0 49515 [1.3200195]
0 49520 [1.3197707]
0 49525 [1.3195229]
0 49530 [1.3192761]
0 49535 [1.3190273]
0 49540 [1.3187797]
0 49545 [1.3185322]
0 49550 [1.3182836]
0 49555 [1.3180364]
0 49560 [1.3177884]
0 49565 [1.3175426]
0 49570 [1.3172951]
0 49575 [1.3170472]
0 49580 [1.3168008]
0 49585 [1.3165516]
0 49590 [1.3163064]
0 49595 [1.3160596]
0 49600 [1.3158115]
0 49605 [1.3155651]
0 49610 [1.3153186]
0 49615 [1.3150715]
0 49620 [1.314826]
0 49625 [1.3145779]
0 49630 [1.3143317]
```

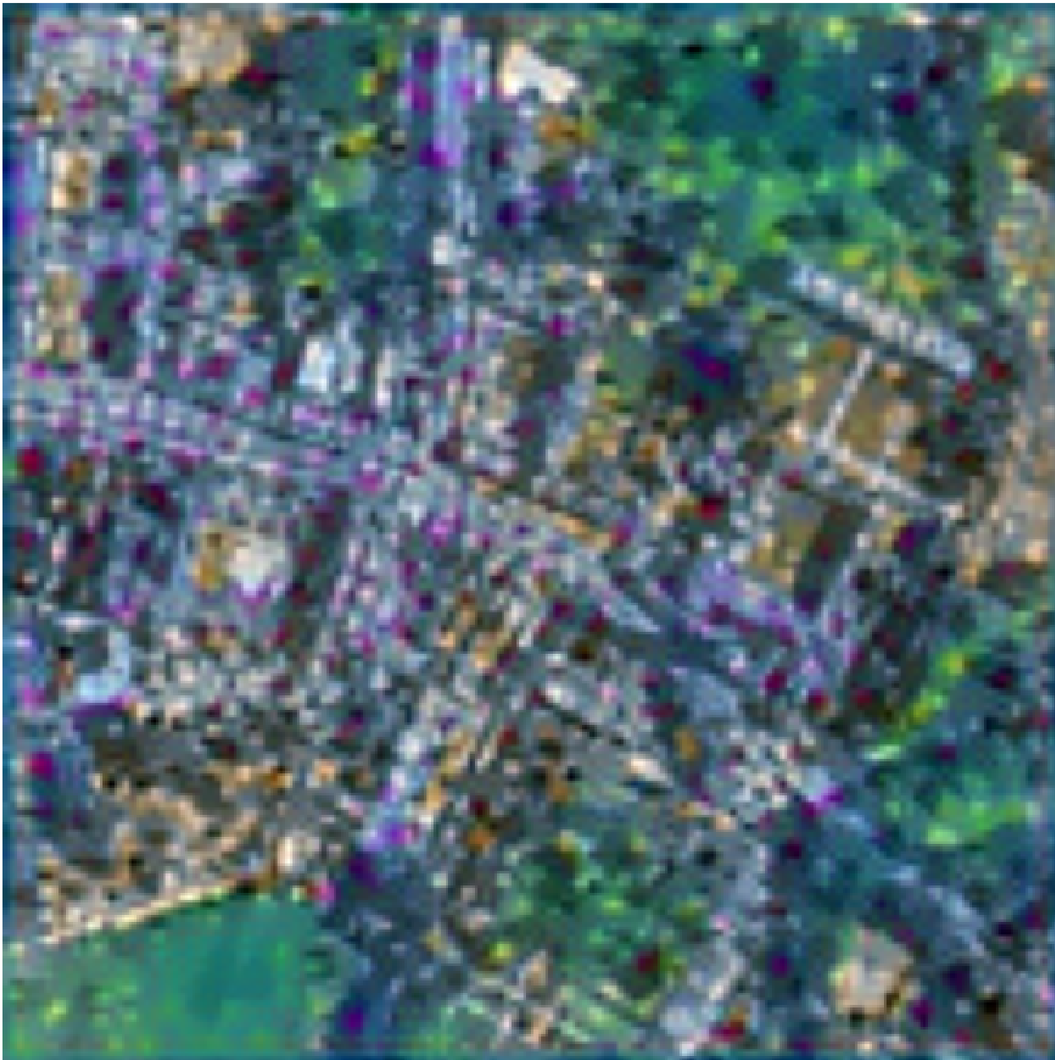
```
0 49635 [1.3140848]
0 49640 [1.3138396]
0 49645 [1.3135935]
0 49650 [1.3133457]
0 49655 [1.3131002]
0 49660 [1.3128533]
0 49665 [1.312608]
0 49670 [1.3123615]
0 49675 [1.312115]
0 49680 [1.3118696]
0 49685 [1.3116236]
0 49690 [1.3113786]
0 49695 [1.311132]
0 49700 [1.3108866]
0 49705 [1.3106405]
0 49710 [1.3103954]
0 49715 [1.3101501]
0 49720 [1.3099048]
0 49725 [1.309658]
0 49730 [1.309412]
0 49735 [1.3091674]
0 49740 [1.308923]
0 49745 [1.3086772]
0 49750 [1.3084329]
0 49755 [1.3081884]
0 49760 [1.3079417]
0 49765 [1.3076963]
0 49770 [1.3074526]
0 49775 [1.307208]
0 49780 [1.3069632]
0 49785 [1.3067188]
0 49790 [1.3064746]
0 49795 [1.3062298]
0 49800 [1.3059857]
0 49805 [1.3057408]
0 49810 [1.305496]
0 49815 [1.3052521]
0 49820 [1.3050086]
0 49825 [1.3047619]
0 49830 [1.3045187]
0 49835 [1.304276]
0 49840 [1.3040311]
0 49845 [1.3037877]
0 49850 [1.303544]
0 49855 [1.3033009]
0 49860 [1.3030558]
0 49865 [1.3028135]
0 49870 [1.3025695]
0 49875 [1.3023252]
0 49880 [1.302082]
0 49885 [1.3018384]
0 49890 [1.3015958]
0 49895 [1.3013525]
0 49900 [1.3011098]
```

```
0 49905 [1.3008673]
0 49910 [1.3006226]
0 49915 [1.3003787]
0 49920 [1.3001372]
0 49925 [1.2998934]
0 49930 [1.2996521]
0 49935 [1.2994094]
0 49940 [1.2991662]
0 49945 [1.2989234]
0 49950 [1.2986802]
0 49955 [1.2984378]
0 49960 [1.2981951]
0 49965 [1.2979536]
0 49970 [1.2977107]
0 49975 [1.2974677]
0 49980 [1.297226]
0 49985 [1.2969841]
0 49990 [1.2967434]
0 49995 [1.2964998]
```

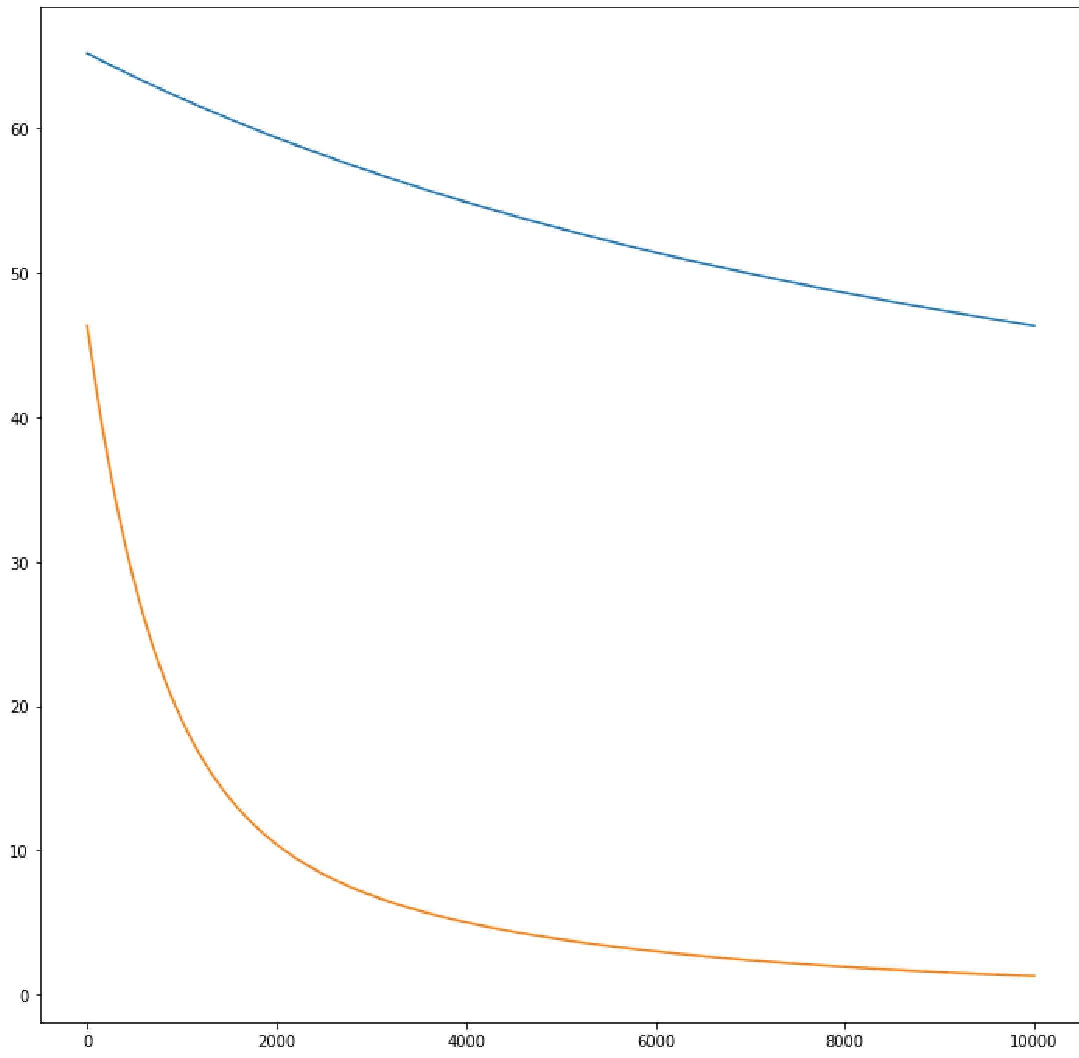
Aleshores el resultat de la trasferencia és:

[75]:
```
plt.figure(figsize=(8,8))
plt.imshow(image[0, :,:,:],vmin=0,vmax=1), plt.axis('off')
plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```



[76]:
```
# Curvas
plt.figure()
for i in range(num_layers):
  plt.plot(L_plot_paso_3[i])
plt.show()
```



60
50
40
30
20
10
0
0
2000
4000
6000
8000
10000