Trabajo Fin de Grado

Grado en Ingeniería de las Tecnologías de Telecomunicación

Deep Learning para realce de huellas dactilares aplicado a la detección de patrones en telas de cuadros del siglo XVII

Autor: Manuel Manzano Hernández-Lissen

Tutor: Juan José Murillo Fuentes

Dpto. Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2022

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Deep Learning para realce de huellas dactilares aplicado a la detección de patrones en telas de cuadros del siglo XVII

Autor:
Manuel Manzano Hernández-Lissen

Tutor:
Juan José Murillo Fuentes
Profesor Catedrático de Universidad

Dpto. de Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2022

Trabajo Fin de Grado: Deep Learning para realce de huellas dactilares aplicado a la detección de patrones en telas de cuadros del siglo XVII

Autor: Manuel Manzano Hernández-Lissen

Tutor: Juan José Murillo Fuentes

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2022

El Secretario del Tribunal

Agradecimientos

Son muchas las personas que me han acompañado en estos últimos años, cinco años llenos de recompensas a la par que de sacrificios y durante los cuales se ha forjado una gran parte de la persona que soy a día de hoy. Es por ello que sería impensable para mí finalizar esta etapa sin hacer una mención especial a todos aquellos que han estado ahí y a los cuales tanto les debo.

En primer lugar, agradecer a mi familia por confiar, comprenderme, escucharme, animarme siempre a dar un poquito más de mí y hacerme entender que en esta vida nadie te regala nada y que la única manera de cosechar éxitos es trabajando y dando todo de nosotros.

A mis abuelos, a quienes se les llena la boca cuando hablan de mí y cuyo orgullo es uno de los mayores regalos que me ha dado mi etapa universitaria.

A todas las personas con las que el destino ha querido cruzarme en la ETSI. A mi Bancada, a mis fastidiades, a los telep**, Cristina, Fran, Sema, Ivan, Rubén… Personas con las que he compartido tantas noches en vela y tantos sinsudores pero con las que más pronto que tarde acabaré celebrando y disfrutando del fruto de nuestro trabajo.

A toda mi gente de Poznań, a mi gente de Málaga, Madrid, a mis gallegos, a mi Wito… gente con la que también he compartido innumerables vivencias y noches en vela y con quienes he aprendido muchas cosas iguales o más importantes que hacer balances de potencia o modelos de pequeña señal.

Por último, agradecer a mi tutor Juan José Murillo por el enorme favor que me ha hecho aceptando a llevar este trabajo conmigo y por hacerlo con la mejor de las actitudes, brindándome en general una ayuda inestimable de la que estaré siempre agradecido.

A todos os debo mucho, quizá más de lo que creéis, así que si por cualquier motivo habéis acabado leyendo esto solo queda decir que os quiero y que mi mano quedará siempre tendida por si en algún momento la necesitáis.

Manuel Manzano Hernández-Lissen

Sevilla, 2022

Resumen

La inteligencia artificial se ha consolidado como uno de los campos más importantes y con mayor proyección en el mundo de la ingeniería actual, siendo totalmente diversas y multidisciplinares sus aplicaciones.

Bajo esta premisa, se pretende explorar en el potencial que puedan tener algoritmos de este tipo, más concretamente de Deep Learning a la hora de mejorar el desempeño de algoritmos existentes, actualizando alguna funcionalidad de éstos mediante el uso de redes neuronales.

Con este objetivo, se propone el uso de la red FPD-M-Net de realce de imágenes de huellas dactilares para intentar extraer un patrón característico de imágenes de rayos X de cuadros del Museo Nacional del Prado, cuya implementación será desarrollada en Python.

Abstract

Artificial intelligence has established itself as one of the most important and promising fields in the world of engineering today, being its applications totally diverse and multidisciplinary.

Under this premise, the aim is to explore the potential that algorithms of this type, more specifically of Deep Learning, can have when improving the performance of existing algorithms, updating some of their functionality through the use of neural networks.

With this objective, we propose the use of the FPD-M-Net network for fingerprint image enhancement to try to extract a characteristic pattern from X-ray images of paintings of the Museo Nacional del Prado, whose implementation will be developed in Python.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

ÍNDICE DE CÓDIGOS

Notación

GPU	Graphical Process Unit
TPU	Tensor Processing Unit
FGPA	Field Programmable Gate Array
CPU	Central Processing Unit
MIT	Massachusetts Institute of Technology
IA	Inteligencia Artifical
ML	Machine Learning
DL	Deep Learning
AFIS	Automatic Fingerprint Identification System

1 INTRODUCCIÓN

1.1 Motivación

El término “Inteligencia Artificial”, como aquel que define la capacidad de un ordenador para emular el comportamiento y raciocinio humano a la hora de resolver todo tipo de problemas es desde hace años cada vez más sonado y conocido popularmente, hasta el punto en el que se ha consolidado como uno de los temas que conforman nuestra “cultura pop”. Sirviendo como gran fuente de inspiración para escritores, directores y guionistas podemos encontrar alusión directa a este concepto en obras que pueden remontarse desde 1950 cuando Isaac Asimov publica *Yo, robot*, hasta *Her* (2013), donde un viudo solitario interpretado por Joaquin Phoenix se enamora perdidamente de la inteligencia artificial detrás del asistente de voz de su teléfono.

Es más que probable que gran parte de la popularidad de ésta se deba a la ciencia ficción y que un alto porcentaje de aquellos que dan sus primeros pasos en el mundo de la IA y el Machine Learning lo hagan con aquella percepción romántica de lo que supone entender el proceso en el que un ordenador es capaz de imitar a simple vista la psique humana. Sin embargo, el mundo que se abre a continuación va mucho más allá, destapando lo que es a día de hoy uno de los campos de mayor estudio y con mayor proyección en la ingeniería.

Bajo esta premisa nace la motivación de este trabajo, la cual es, posterior a un estudio teórico de los términos ya mencionados y otros primordiales para comprender este campo tales como el Deep Learning y las Redes Neuronales Artificiales, buscar una aplicación real para así demostrar el potencial de esta tecnología.

1.2 Objetivos

A la hora de decidir dicha aplicación nos encontramos con el Trabajo de Fin de Máster de María del Mar Velasco: *Extracción de patrones en telas de cuadros del siglo XVII a partir de placas de rayos X*, en él se diseña un algoritmo que tiene como objetivo caracterizar el patrón que presentan las telas de algunos cuadros, formadas por el entrelazado de los hilos que las componen, para así extraer más información que contribuya al estudio de dichas obras de arte [1].

Como primera parte del algoritmo se hace uso de un software de realce de imagen de huellas dactilares, (dada las similitudes que a simple vista se puede intuir que comparten los surcos que forman los hilos que conforman una tela y los propios de una huella dactilar, aunque se hará hincapié más adelante) como parte del pre-procesamiento que se lleva a cabo en las imágenes de los cuadros.

Al igual que en muchísimos otros ámbitos, pues las aplicaciones del Deep Learning son prácticamente inabarcables, en el campo de las huellas dactilares éste tiene una gran importancia en el estudio actual de técnicas de reconocimiento, realce y eliminación de ruido, entre muchas otras. Es por ello por lo que se antoja como un reto más que interesante el estudio de dichas técnicas de DL, con el objetivo final de intentar mejorar el desempeño global del algoritmo diseñado por María del Mar.

De este modo, se tratará de evaluar el estado del arte en dicha materia para buscar una red neuronal capaz de sustituir el algoritmo de realce de huellas usado originalmente. Posteriormente se evaluará el resultado del código de la compañera para hacer una comparativa y arrojar las conclusiones pertinentes.

1.3 Estructura de la memoria

Para cumplir los objetivos previamente mencionados se llevará a cabo el trabajo bajo la estructura que sigue a continuación:

- **Capítulo 1: Introducción**. En este primer capítulo se explica cuáles han sido los intereses que han motivado al autor a llevar a cabo este trabajo, así como los objetivos y metas establecidas en éste.
- **Capítulo 2: Inteligencia Artificial, Machine Learning y Deep Learning**. En el segundo capítulo se hará una primera toma de contacto con el área de estudio de este trabajo, presentando los tres conceptos que dan nombre al mismo y la jerarquía existente entre ellos.
- **Capítulo 3: Marco teórico**. Aquí se hará una introducción teórica a la redes neuronales convolucionales, así como a sus características principales necesarias para el claro entendimiento de los capítulos posteriores.
- **Capítulo 4**: **Propuesta de trabajo**. Donde se detallará desde un enfoque más técnico el objetivo del trabajo.
- **Capítulo 5: Estado Del Arte.** Aquí se resumirá todo el estudio realizado en materia de redes neuronales para el realce y obtención del campo de orientación de imágenes de huellas dactilares, así como la búsqueda de la red neuronal empleada finalmente.
- **Capítulo 6: Implementación del proyecto.** En este capítulo se detallarán las cuatro actualizaciones al código original de la red neuronal empleada de forma cronológica, en función de los resultados obtenidos tras cada una de ellas y las mejoras que fueron necesarias realizar.
- **Capítulo 7: Conclusiones y líneas futuras.** Por último, se analizará el resultado final del trabajo y se propondrán nuevas líneas de estudio que tomen como referencia o punto de partida el presente trabajo.

2 INTELIGENCIA ARTIFICIAL, MACHINE LEARNING Y DEEP LEARNING

Aunque se trate de un concepto de extensísima popularidad, lo cierto es que no existe una definición totalmente consensuada dentro de la comunidad científica sobre lo que realmente es la inteligencia artificial, o sobre qué respuestas o resultados obtenidos por un ordenador o software informático pueden caracterizarse realmente como "inteligentes".

La realidad es que se trata de una ciencia amplia, cuyas barreras no están perfectamente delimitadas a día de hoy. Se trata de un campo multidisciplinar, que en muchas ocasiones se une con otras ciencias tales como la psicología, la economía e incluso la filosofía.

2.1 Inteligencia Artificial

De este modo, pues podríamos definir de forma generalizada la inteligencia artificial como la capacidad de un ordenador para resolver problemas del mismo modo que lo haría un ser humano, desde su comienzo investigadores han diferenciado varias formas de concebirla dependiendo de si este comportamiento simplemente trata de emular el raciocinio humano, en términos de elegir la mejor solución dado un contexto específico, o si por el contrario es necesario analizar todo el proceso cognitivo que se lleva a cabo en el cerebro humano (a nivel neuronal inclusive) con el objetivo de replicarlo en algoritmos computacionales [2].

Inteligencia Artificial	Ciencia Cognitiva	Sistemas expertos Algoritmos genéticos Agentes inteligentes Sistemas multiagente
	Robótica	Visión artificial Locomoción Navegación
	Interfaces de usuario natural	Reconocimiento del lenguaje natural Comprensión del lenguaje Realidad Virtual

Tabla 2-1. Áreas de la IA [2], [3].

2.2 Machine Learning

Dentro de la inteligencia artificial, cuyas áreas se clasifican en función de los paradigmas previamente mencionados y que se muestran junto con algunos ejemplos en la figura 2-1 podemos destacar la

subdisciplina del aprendizaje automático o *Machine Learning*. Éste tiene como principal característica que aquellos modelos o algoritmos usados para abordar cada problema particular se ajustan para dar la solución más óptima posible de forma automática, es decir, sin necesidad de seguir un algoritmo fijo que dependa de interacción humana [4].

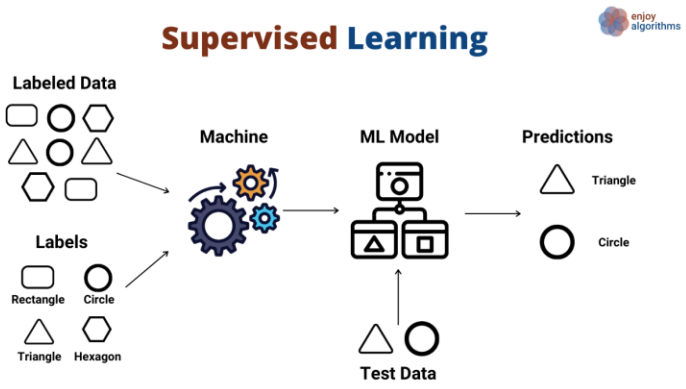
Esto se consigue mediante un proceso de entrenamiento, que parte de un conjunto de datos, llamado datos de entrenamiento, de los cuales se intentan obtener similitudes, características o patrones para así predecir futuros resultados dentro del problema concreto para el que sea destinado. Dependiendo de si existe algún tipo de conocimiento previo o *feedback* asociado a estos datos o no se definen tres clases principales de algoritmos de aprendizaje automático: supervisado, no supervisado y por refuerzo (*Reinforcement Learning*) [5]. Detallaremos las características principales de los dos primeros:

2.2.1 Aprendizaje supervisado

Llamamos etiquetas (o *label*, en inglés) a aquellas características o atributos relevantes que poseen los datos de entrada de un algoritmo de machine learning y que se espera que se puedan obtener y clasificar de forma predictiva por éste una vez pasado el proceso de entrenamiento. En el aprendizaje supervisado se conocen todas las etiquetas de los datos de entrenamiento, por lo que el objetivo del sistema será el de mapear los datos de entrada a estas etiquetas ya conocidas. Algoritmos conocidos que empleen esta metodología son la regresión lineal, árboles de decision y redes neuronales [4] [6].

2.2.2 Aprendizaje no supervisado

Por el contrario, si no se conocen explícitamente estos atributos estaremos hablando de algoritmos de aprendizaje no supervisado. Dado que no se especifica qué tipo de atributos se han de identificar dentro de los datos de entrada, estos algoritmos tratarán de mapear todas las características o *features* de los datos de entrada en sus salidas, lo que los convierten en los más apropiados para desempeñar tareas de *clustering*. En este tipo de tareas es el propio sistema el que trata de agrupar los datos de entrada en subconjuntos en función de características que se haya determinado que tienen en común [4] [5] [6].



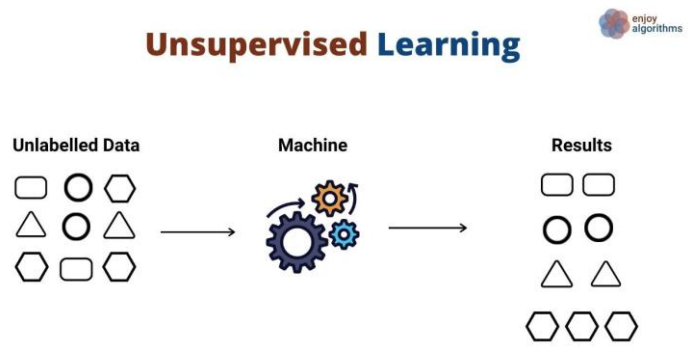


Figura 2-1. Tipos de aprendizaje automático [7].

2.3 Deep Learning

A medida que el conocimiento en Machine Learning fue aumentando, horizontes cada vez más ambiciosos fueron descubriéndose y junto con ellos se fue intentando abarcar problemas cada vez más complejos y con mayor capacidad y requerimientos computacionales. Nace así el aprendizaje profundo, más conocido por su traducción al inglés, Deep Learning.

Se definen como algoritmos de DL aquellos que utilizan múltiples capas donde se procesan datos. Conforme aumentan el número de capas y las interconexiones entre las neuronas artificiales que las componen aumenta el nivel de abstracción y por tanto la complejidad de los problemas que será capaz de resolver.

Modelos de DL de gran escala pueden contener decenas de billones de parámetros que deberán ser ajustados en el proceso de entrenamiento. Sin embargo, la evolución en la demanda actual de hardware cada vez más potente y la llegada de GPUs, TPUs y FGPAs, capaces de realizar entre 10^{14} y 10^{17} operaciones por segundo (frente a las 10^{10} de una CPU convencional [5]) unidas con el auge actual del Big Data hacen que estos algoritmos no solo puedan llevarse a cabo, sino que presenten un futuro más que prometedor en el mundo de la ciencia computacional.

Queda cubierto así el objetivo de este capítulo introductorio, el cual es dar un enfoque global de la inteligencia artificial como el conjunto de técnicas para resolver problemas humanos mediante la computación. Dentro de éste se destaca el Machine Learning como un subconjunto que resuelve estos problemas con modelos capaces de ajustarse de forma automática con el objetivo de dar una solución óptima. Por último, aquellos algoritmos de ML que siguen una estructura de redes neuronales, agrupadas en capas profundas se definen bajo la categoría de Deep Learning.

Un esquema a modo de resumen es el que se muestra en la Figura 2-2.

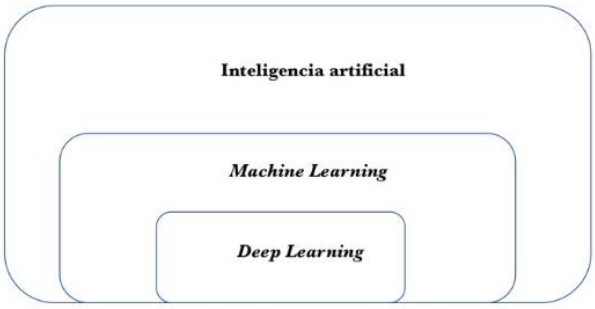


Figura 2-2. Relación entre IA, ML y DL [4].

3 MARCO TEÓRICO

Como se ha introducido en el capítulo anterior, las aplicaciones más complejas y por tanto interesantes de la inteligencia artificial vienen de la mano del aprendizaje profundo o Deep Learning. Siendo ésta la rama de estudio de este trabajo, pues se tratará de trabajar con redes neuronales convolucionales, será menester hacer una introducción teórica sobre aquellos conceptos básicos tanto de Machine Learning como de Deep Learning que faciliten la comprensión de los capítulos posteriores.

3.1 Redes Neuronales Artificiales

En los años 50, de la mano del nacimiento de la inteligencia artificial, los procesos de funcionamiento del cerebro humano suscitaron un gran interés dentro de la comunidad científica. Pronto, científicos presentaron diversos modelos matemáticos que trataban de explicarlo de forma esquemática, tomando como referencia los conocimientos en neurociencia que se tenían hasta la fecha.

De este modo los investigadores del MIT Warren McCulloch y Walter Pitts presentaron en 1943 el primer modelo de una red neuronal, en él, las diferentes neuronas podían activarse o no dependiendo de la presencia de estímulos externos. En su estado de activación emitirían señales a todas las neuronas interconectadas con ellas emulando la sinapsis en el caso real. Si bien este modelo no contemplaba la capacidad de moldear sus valores de activación, pesos sinápticos y en definitiva su capacidad de aprender, plantearon un nuevo paradigma a la hora de resolver problemas empleando unidades (llamadas neuronas ya que se intentaba replicar el comportamiento del cerebro) interconectadas entre sí [8].

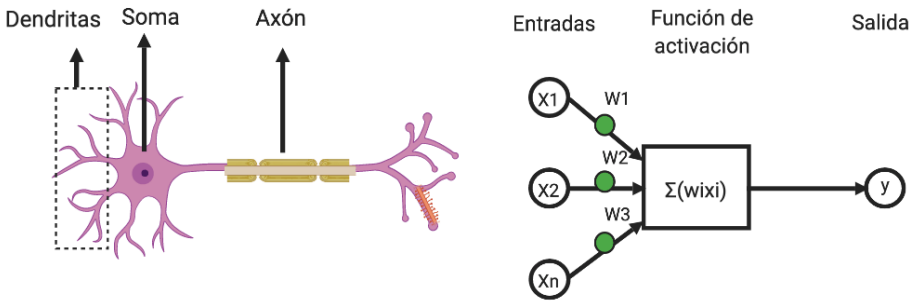


Figura 3-1. Neurona biológica y modelo de McCulloch y Pitts [9].

Faltaba pues emular esa capacidad del ser humano de aprender y desarrollarse en base a las experiencias vividas y conocimientos adquiridos durante el tiempo, lo cual ocurrió una década después, en el 1957, cuando Frank Rossenblatt inventó el primer modelo una de red neuronal artificial capaz de hacerlo: el **perceptron**, asentando así las bases del Deep Learning.

3.2 El perceptron

El perceptron es la unidad básica constitutiva de las redes neuronales artficiales, pues modela el comportamiento de una red formada por una única capa y una única neurona tal y como se muestra en la Figura 3-2:

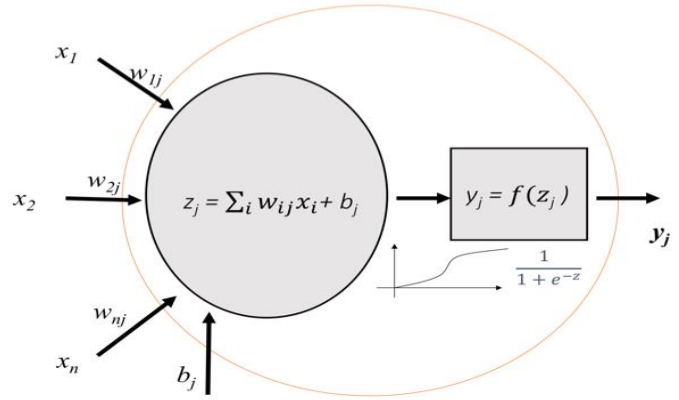


Figura 3-2. Esquema del perceptrón [4].

Esta neurona recibe un conjunto de **entradas** $x=[x_1, x_2, ..., x_n]$ y un sesgo b_j. Cada uno de ellos es multiplicado por su **peso** asociado, que de forma vectorial se representa de la siguiente forma: $w=[w_1, w_2, ..., w_n]$. La salida de la neurona es por tanto una combinación lineal de las entradas y el sesgo que sigue la fórmula 3-1:

$$z_j = w^T x = \sum_{i=1}^{n} w_{ij} x_i + b_j \tag{3-1}$$

El **sesgo** b_j, también conocido como *bias* en inglés, funciona a todos los efectos como un peso más, el cual es multiplicado por una entrada que siempre tendrá valor unitario ($x_0 = 1$). Gracias a él, la salida de la neurona será siempre distinta de cero incluso cuando todas sus salidas son cero. Puede intuirse que la existencia de este sesgo define un valor mínimo a la salida y es por ello que también se le conoce con el nombre de **umbral de activación** [5] [10].

Nombrando el sesgo como un peso más, pues así será tratado a la hora de ajustar su valor, obtenemos la formula genérica del perceptron:

$$z_j = \sum_{i=0}^{n} w_{ij} x_i \tag{3-2}$$

Aquí los vectores de entrada y de pesos tienen dimensión n+1: $x=[1, x_1, ..., x_n]$, $w=[b_j, w_1, ..., w_n]$. El subíndice j hace referencia a la j-ésima capa, por lo que al tratarse del caso concreto de una sola capa del perceptron podría obviarse.

La salida final del perceptron será el resultado de aplicar una función no lineal $f(\cdot)$, llamada **función de activación**, a la salida z_j. El hecho de que esta function deba ser no lineal se respalda en el **teorema de aproximación universal**, el cual aplicado en materia de redes neuronales establece que ante un número suficiente de capas, que empleen un número suficiente de funciones de activación no lineales, el algoritmo podrá acercarse de forma exponencial a representar cualquier función lineal, es decir, a ajustarse a un problema de cualquier tipo y clasificación de sus datos [11].

La forma que pueda tomar esta función dependerá del objetivo para el que se diseñe la red y será por

tanto tarea del programador elegirla, sin embargo se muestran a continuación las más comunes:

La más simple de todas es la función de activación lineal (***linear***), si bien es cierto que no añade la no linearidad tan importante mencionada anteriormente, para propósitos sencillos y aplicaciones concretas en las que no es necesaria basta con replicar la señal de entrada y esta función puede emplearse.

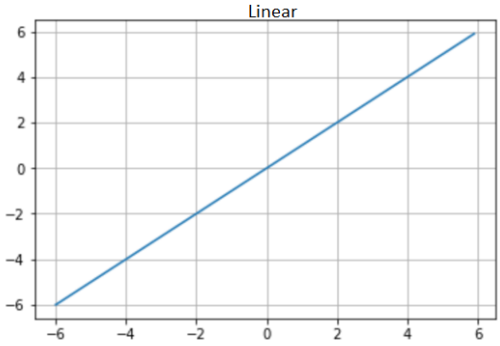


Figura 3-3. Función lineal [4].

Usada ampliamente para aplicaciones de clasificación binaria, o para obtener resultados que modelen la probabilidad de que algo ocurra o no se utiliza la función sigmoidea (***sigmoid***).

Atendiendo a su fórmula (3-3), resulta inmediato entender su funcionamiento: para valores positivos (o superiores al umbral de activación, en caso de tener en cuenta el bias) la función *sigmoid* mapea rápidamente la salida a uno. Por el contrario, para valores negativos rápidamente se acerca a cero.

$$f_{sigmoid}(x) = \frac{1}{1+e^{-x}} \tag{3-3}$$

El mayor problema relacionado con el uso de esta función reside en que para ambas asíntotas su gradiente se anula. Esto se conoce como el **problema del desvanecimiento del gradiente** (*vanishing gradient problem*) y, como veremos más adelante, ralentiza enormemente el proceso de obtención de los pesos [10].

Como alternativa para paliar este problema se puede emplear la función **tangente hiperbólica** (tanh). Como puede verse en su representación en la Figura 3-4, al tener mayor rango: (-1,1) es más robusta.

$$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} = \frac{2}{1+e^{-2x}} - 1 = 2\,f_{sigmoid}(2x) - 1 \tag{3-4}$$

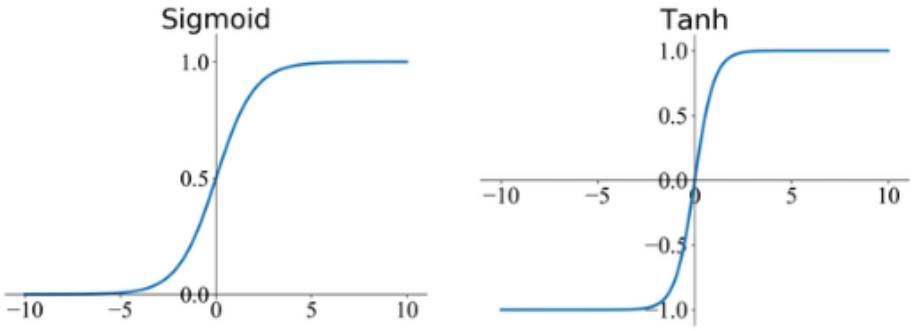


Figura 3-4. Funciones sigmoid y tanh [12].

Para aplicaciones, también muy comunes, en las que haya que clasificar múltiples clases se emplea la función ***softmax***. Suele emplearse como función de activación en una capa de salida con tantas neuronas como clases se pretenda clasificar en el problema. Toma todos los valores de salida de las neuronas de su capa y devuelve la probabilidad relativa de que el valor de su entrada pertenezca a su clase asociada [13].

Para la i-ésima neurona y un número total de clases y neuronas en su capa igual a K, la función *softmax* se puede expresar de la siguiente manera:

$$f_{soft\max}(x)_i = \frac{e^{x_i}}{\sum_{k=1}^{K} e^{x_k}} \tag{3-4}$$

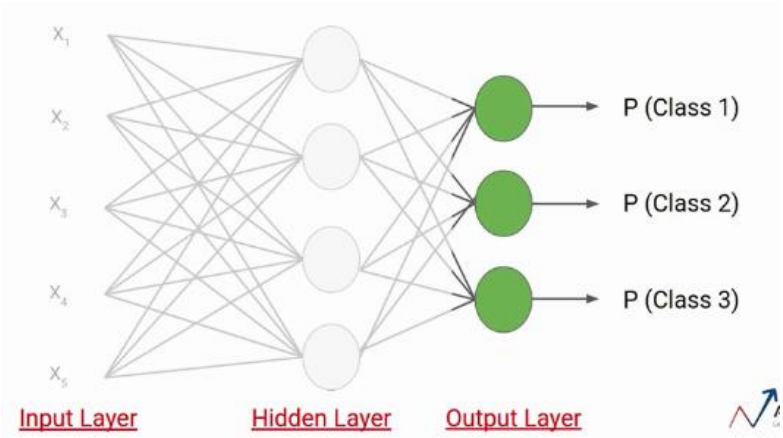


Figura 3-5. Ejemplo de clasificación de 3 clases con softmax [13].

Por último, una de las funciones más populares en los algoritmos actuales es la **ReLU** (*Rectified Linear Unit*).

El mapeo que realiza es: para entradas por encima de un umbral, de forma lineal, no activando el nodo si la entrada no supera dicho umbral. Esto convierte esta función en una muy eficiente, pues simplifica muchos valores dando una salida nula, sin dejar de ofrecer una respuesta no lineal que recordemos es vital para obtener resultados complejos [10].

$$f_{\mathrm{Re}LU}(x) = \begin{Bmatrix} 0 \text{ for } x < 0 \\ x \text{ for } x \geq 0 \end{Bmatrix} \tag{3-5}$$

Para aumentar más todavía la capacidad de aprendizaje al hacer posible el cálculo del gradiente en la zona por debajo del umbral, se añade cierta pendiente a , obteniendo así la **leaky ReLU**:

$$f_{leaky\ \mathrm{Re}LU}(x) = \begin{Bmatrix} ax \text{ for } x < 0 \\ x \text{ for } x \geq 0 \end{Bmatrix};\ (a = 0.1 \text{ p.e}) \tag{3-6}$$

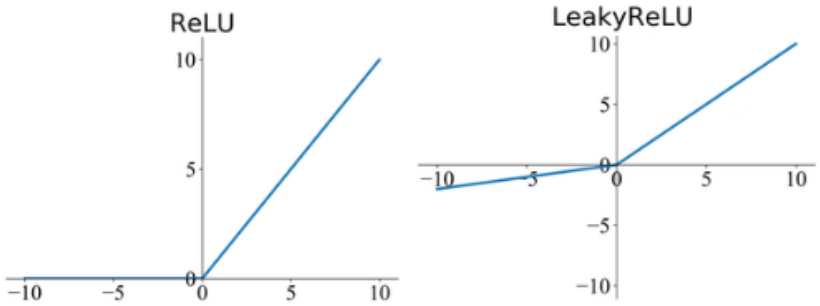


Figura 3-6. Funciones ReLU y Leaky ReLU [12].

Una vez conocida la función de activación $f(\cdot)$ podemos denotar finalmente como y_j a la salida del perceptron y expresarla mediante la siguiente formula:

$$y_j = f(z_j) \tag{3-7}$$

3.3 El perceptron multicapa

Siguiendo el razonamiento de este trabajo, el siguiente paso lógico para poder abarcar problemas de mayor envergadura mediante algoritmos de ML será el de agrupar perceptrones (a partir de ahora neuronas) en capas. Nace así el concepto de perceptron multicapa o ***Multi-Layer Perceptron*** **(MLP)**.

En su forma más genérica, una red neuronal será una sucesión de capas formadas por neuronas que realizarán un mapeo de un conjunto de entradas en otro de salidas. La conexión entre capas será sin bucles de realimentación y la salida de cada capa funcionará como entrada de la capa inmediatamente posterior, por lo que también se las conoce como ***Feed-Forward Neural Networks*** **(FFNN)**. Además, si la información de salida de cada neurona de una capa es propagada a todas las neuronas de la siguiente se clasifican bajo el nombre de redes o capas densamente o **completamente conectadas** (*fully-connected layers/networks*) [4] [8].

Estas capas se agrupan bajo tres categorías:

- **Capa de entrada**: Recibe la entrada del exterior y se limita a replicarla en la primera capa oculta.
- **Capas ocultas**: En inglés, *hidden layers*, deben su nombre a que sus valores no aparecen en los datos de entrada, por el contrario, será el propio modelo el que a medida que vaya aprendiendo vaya decidiendo qué características o relaciones entre ellas son útiles dentro de los datos que obtenga a su entrada [14]. Como se ha adelantado en capítulos anteriores, conforme aumente el

número de capas ocultas, aumentará la capacidad de abstracción de la red y por tanto la complejidad de los problemas que esta será capaz de resolver. Si la red posee más de una capa oculta, será calificada como profunda (***deep neural network***) y estará abarcando problemas bajo el área del aprendizaje profundo o *Deep Learning*.

- **Capa de salida**: Arroja el resultado de la red. Este resultado será visible y tendrá una interpretación relacionada con la entrada de la red, lo cual no era necesario durante todo el paso a lo largo de las capas ocultas.

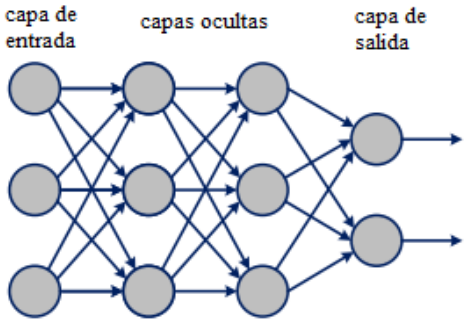


Figura 3-7. Esquema de una fully-connected FFNN, con dos capas ocultas [8].

3.4 Redes Neuronales Convolucionales

Un tipo de redes neuronales artificiales son las redes neuronales convolucionales, en inglés ***Convolutional Neural Networks,*** también conocidas como ***ConvNets*** o simplemente por sus siglas **CNN**.

La estructura de este tipo de redes es igual que las vistas previamente, con la salvedad de que los datos de entrada son tensores (matrices de dos o más dimensiones) y que en una o varias de sus capas ocultas se realizan **operaciones de convolución**, recibiendo así su nombre. Esto mismo es lo que las hace especialmente útiles a la hora de trabajar con señales de audio, video o imagen y cuyo uso resultó en un cambio de paradigma en el área de la visión por computador.

Centrándonos en el caso de las imágenes, éstas serán tensores de tres dimensiones: Altura (*height*), anchura (*width*) y profundidad (*depth*). La profundidad en la capa de entrada no hace referencia más que a su número de canales, por lo que también se conoce como *channels*. Para imágenes RGB el número de canales será 3, mientras que para imágenes en blanco y negro solo será 1, correspondiente a su nivel de gris.

3.4.1 Convolución y Kernels

A la hora de extraer información de una imagen, es lógico pensar que las características relevantes para la resolución del problema no se extraigan en un píxel en concreto, sino que haya que tener en cuenta las relaciones entre **píxeles adyacentes**. Es ésta precisamente la característica principal de las CNNs, pues toma pequeñas regiones locales dentro de la imagen de las que va extrayendo la información significativa.

De esta forma, como se muestra en la imagen 3-8, las capas iniciales de la red serán capaces de extraer información muy simple, como pueden ser bordes. Sin embargo, a medida que crezca el modelo el nivel de abstracción de la red convolucional aumentará, permitiendo poco a poco diferenciar contornos, partes

de objetos hasta llegar al objetivo deseado [5].

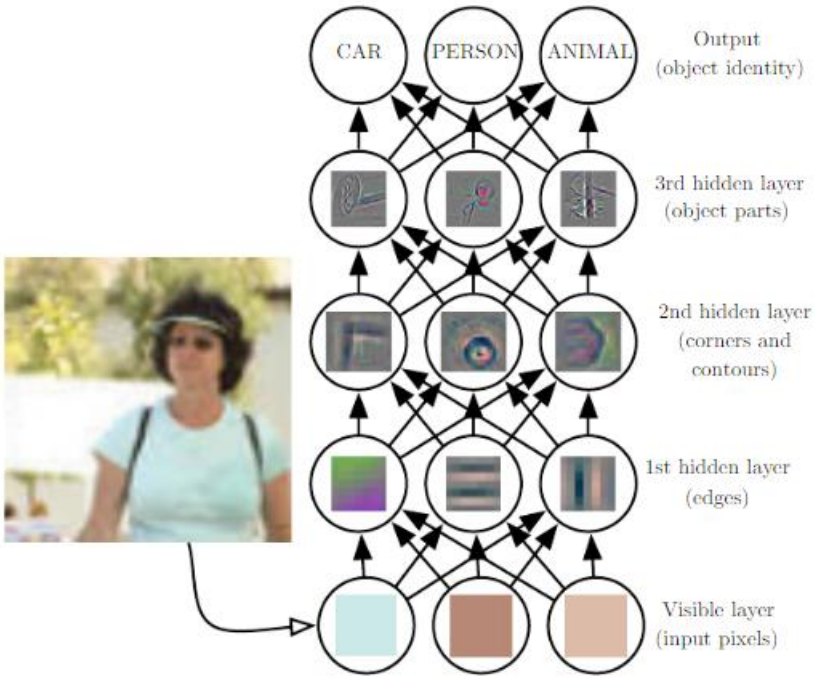


Figura 3-8. Esquema de como una CNN aumenta su nivel de abstracción conforme aumenta su número de capas [14].

Esto se consigue mediante la aplicación de filtros, llamados ***Kernels***. Cada capa convolucional vendrá de realizar un producto escalar entre un puñado de píxeles (neuronas de la capa de entrada) y el propio *kernel*, cuyos valores serán los pesos a calcular durante la fase de entrenamiento, a lo largo de toda la imagen de entrada. El tamaño de la ventana corresponderá con el del filtro y a esta operación se la conoce en terminología de DL como **convolución**.

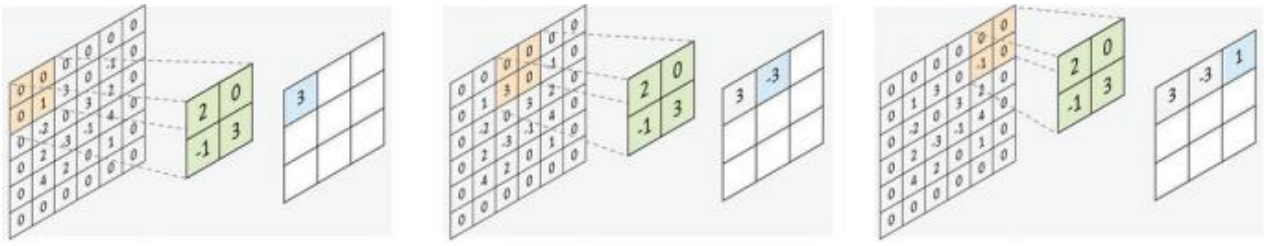


Figura 3-9. Ejemplo de se aplicaría un *kernel* (en verde) para obtener una capa convolucional (derecha) a partir de una capa de entrada (izquierda) [15].

Finalmente hay que tener en cuenta que al entrenar cada *kernel* este sirve para identificar una única característica o *feature*, por lo que en la obtención de cada capa de la red se suelen aplicar varios filtros, cambiando asimismo la profundidad de ésta. En el caso de ejemplo de la Figura 3-10 se estarían intentando detectar 32 *features*, por lo que a partir de una capa de entrada de tamaño 28x28x1 obtendríamos una capa oculta de 24x24x32.

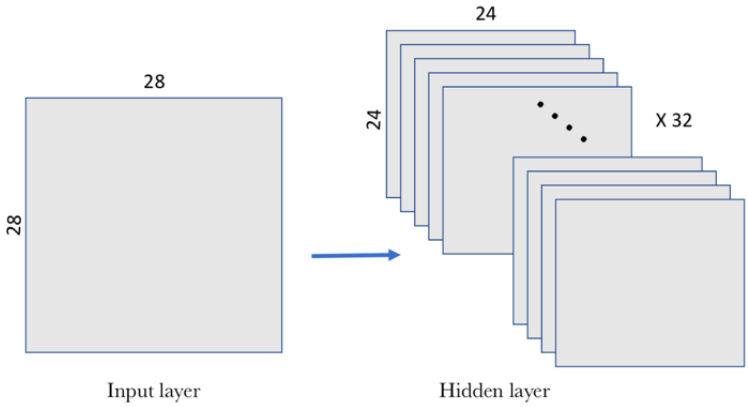


Figura 3-10. Ejemplo de cómo aumenta la profundidad de una capa oculta al computar más de un *kernel* [4].

3.4.2 Pooling y autoencoders

Igual de importantes que las capas convolucionales son las capas de ***pooling***, las cuales tienen como objetivo condensar y simplificar la información de una capa convolucional manteniendo su relación espacial.

Su obtención será similar a las de las capas convolucionales, pues una ventana recorrerá las imágenes de la capa anterior. La diferencia es que en este caso los valores del filtro ya estarán definidos y tendrán como objetivo aplicar una función lineal (en el caso del ***average-pooling***, donde el valor del píxel de salida corresponde al de la media de los pixeles de la ventana) o no lineal, si se trata por ejemplo del ***maxpooling***, donde se toma como valor de salida el mayor de los valores de la ventana.

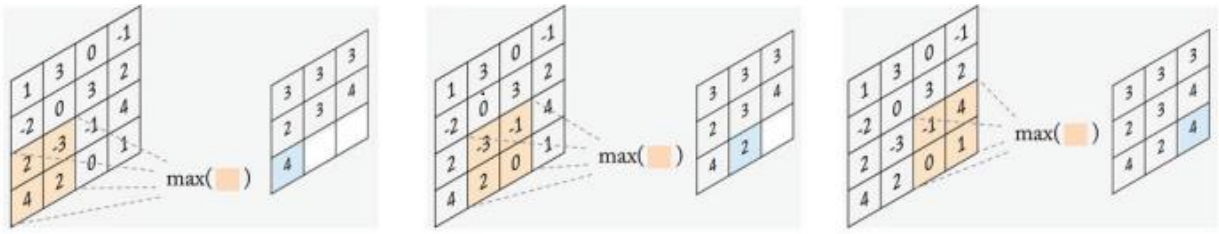


Figura 3-11. Ejemplo de *maxpooling* [15].

De la misma forma que puede reducirse el tamaño de las imágenes mediante el *pooling* también se puede hacer el proceso inverso, pues hay que tener en cuenta que existen aplicaciones en las que la salida no es necesariamente una clasificación, (para las que sí que se necesita que la parte final de la red corresponda con una *fully-connected* y que por tanto la información se vaya comprimiendo sucesivamente) sino otra imagen que preserve las dimensiones de la original.

Este proceso es conocido con el nombre de ***unpooling***. Ejemplos de él pueden ser el *max-unpooling*, *nearest neighbour* (replicar el valor del cada píxel de la ventana en el número de filas y columnas como se quiera aumentar el tamaño de la imagen) o la *transpose convolution*. Esta última consiste en realizar la operación inversa a la convolución con un *kernel* cuyos parámetros también son obtenidos en la fase de

entrenamiento.

Si la red está formada por capas tanto de *pooling* como de *unpooling*, recibe el nombre de ***autoencoder*** pues la información de entrada está siendo codificada en la primera etapa y descodificada en la segunda [16].

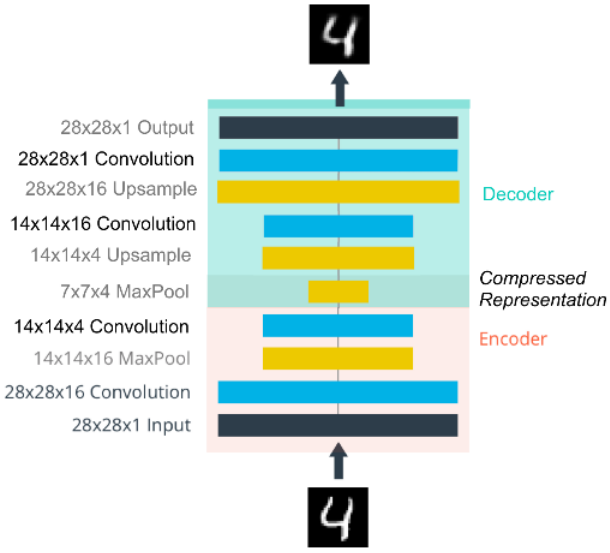


Figura 3-12. Ejemplo de *autoencoder* [16].

3.4.3 Hiperparámetros de una CNN

Para completar este repaso general de los conceptos básicos sobre las redes neuronales convolucionales solo quedan por estudiar los hiperparámetros de las capas convolucionales. Éstos son cuatro y son elegidos por el programador a la hora de diseñar la red:

- **Número de filtros**: Como hemos visto previamente, este será igual al número de características diferentes que se quieran detectar y determinará la profundidad de la capa convolucional.
- **Tamaño de la máscara de convolución**: Define el tamaño del filtro y como se ha visto anteriormente la región de la capa de entrada que computa, también llamado campo receptivo de la neurona. Los tamaños de máscara suelen ser reducidos e iguales en ambos ejes (i.e 3x3, 5x5 o 7x7)
- **Paso, salto o zancada**: En inglés, ***Stride***. Dependiendo de las necesidades del problema es posible que no sea necesario realizar la convolución tradicional, sino que puedan saltarse filas y/o columnas con el objetivo de reducir el número de operaciones a realizar o reducir el tamaño de la capa de salida. Tomando de ejemplo la figura 3-9, podemos ver que un *stride* de 2 píxeles reduciría en un factor de 4 el tamaño de la salida (pasamos de una imagen de entrada de 6x6 a una de salida de 3x3).

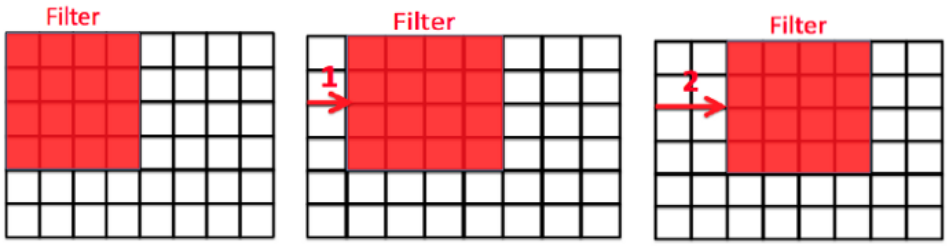


Figura 3-13. Ejemplo del barrido del filtro para *Stride*=1 (centro) y *Stride*=2 (derecha) [17].

- ***Padding***: Como se ha visto anteriormente, después de cada operación de convolución la imagen original pierde dimensionalidad con un factor que depende del *stride*, incluso cuando éste es unitario. Esto es lo que se conoce con el nombre de **efecto de borde** y puede afectar de manera considerable al rendimiento de la red, especialmente cuando se usan imágenes pequeñas o filtros grandes. El *padding*, que no es más que un relleno de ceros en las filas y columnas exteriores a la imagen de entrada ayuda tanto a preservar el tamaño original de ésta, delegando el submuestreo en las capas de pooling, diseñadas para ello, como para evitar los efectos de borde.

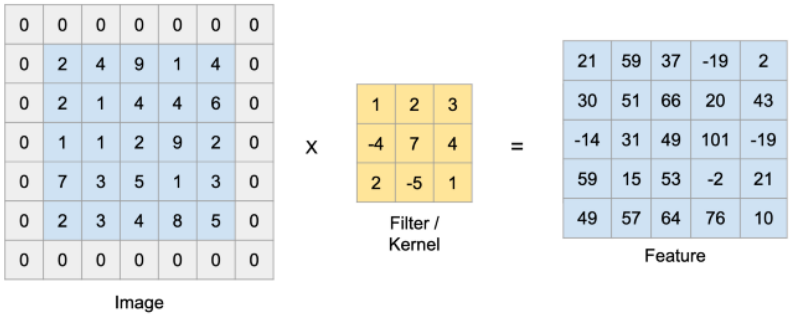


Figura 3-14. Ejemplo de *padding* para conservar las dimensiones de la capa de entrada [18].

4 PROPUESTA DE TRABAJO

Una vez hecho el contexto teórico sobre el área de estudio de nuestro trabajo: el Deep Learning y más concretamente el uso de redes neuronales convolucionales, podemos enfocar nuestro punto de mira en el objetivo de éste, el cual recordamos es estudiar la viabilidad del uso de algún algoritmo de DL para mejorar programas existentes.

4.1 Definición de objetivos

El escogido en este caso es el TFM de María del Mar Velasco, antigua alumna de la Universidad de Sevilla. En su trabajo se toman imágenes de rayos X de cuadros del Museo Nacional del Prado con el objetivo de extraer patrones característicos, en forma de romboides, que posteriormente se puedan analizar para así extraer datos que sirva como nueva fuente información relevante en estudios posteriores [1].

Su algoritmo tiene una fase inicial, que podríamos considerar de pre-procesado, en la que a partir de la imagen de rayos X original se pretende resaltar el entrelazado de los hilos del lienzo en cuestión. En la Figura 4-1 se ve con claridad el objetivo, pues se eliminan manchas que pueden ser procedentes tanto del propio deterioro de la tela como de que parte de la pintura sea captada incluso al tratarse de una imagen de rayos X.

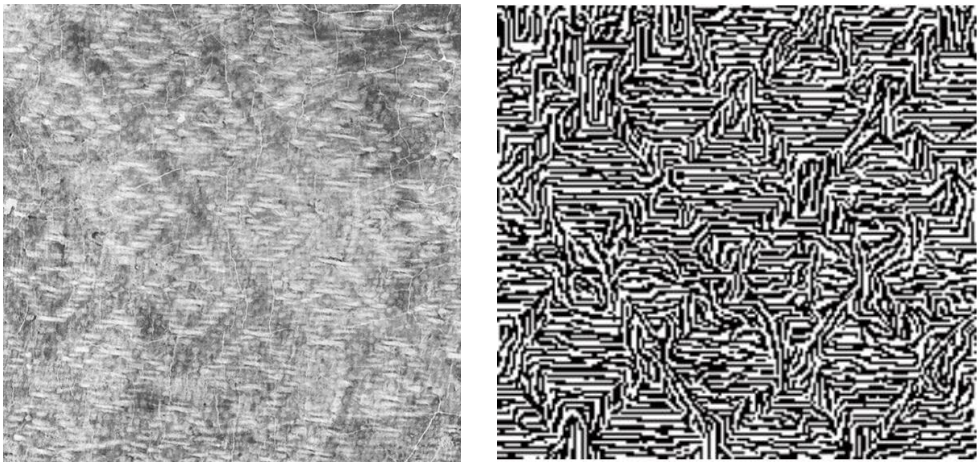


Figura 4-1. Fragmento de la imagen de rayos X de uno de los cuadros (izquierda) y el resultado de aplicar el algoritmo de huella dactilar a la misma (derecha) [1].

Para obtener dicho objetivo se empleó el software de Matlab de uso libre de Peter Kovesi [19], quien adaptó el algoritmo rápido de realce de huellas dactilares desarrollado por Lin Hong, Yifei Wan y Anil Jain [20].

Es en uno de los pasos posteriores del algoritmo de María del Mar donde estará de nuevo nuestro punto de mira, pues tras procesar la imagen binaria de huella dactilar resultante de la fase inicial (tal y como se explica en la sección 4.2 de su trabajo [1]) se extraen las direcciones principales de la textura del lienzo.

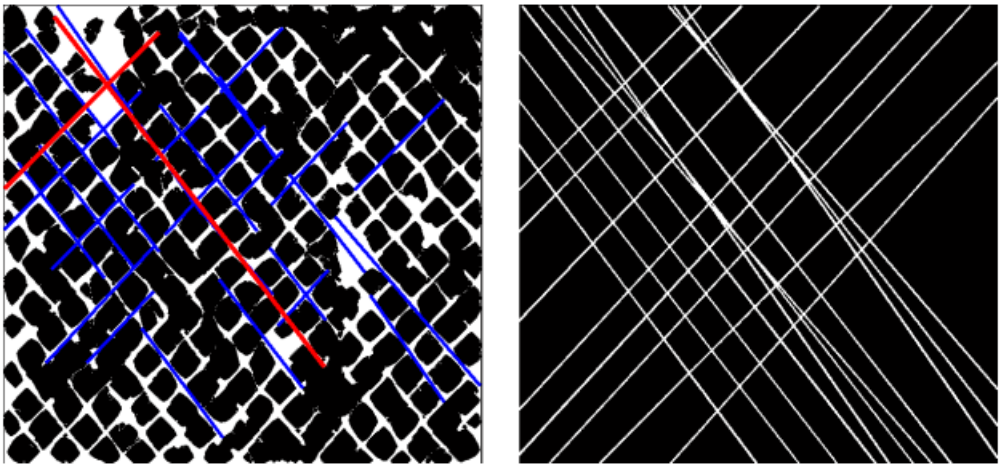


Figura 4-2. Izquierda: Líneas encontradas con información relevante. Derecha: Máscara formada a partir de éstas [1].

El resultado de este paso es el que se muestra en la Figura 4-2. Las líneas rojas de la imagen de la izquierda hacen referencia a las dos direcciones principales mientras que las azules presentan todas aquellas con ángulos muy parecidos a los principales. A partir de estos datos continúa el algoritmo de María del Mar, pero nos detendremos en este punto.

En este trabajo se analizarán dos formas en las que, mediante algoritmos de DL, se pueda contribuir a mejorar la fase del trabajo mencionada anteriormente. Estos son:

- El uso de una **red neuronal de realce de huellas dactilares** que sustituya el algoritmo de Peter Kovesi. De esta forma los resultados obtenidos serían analizados en las sucesivas fases del algoritmo de María del Mar.
- El uso de una **red neuronal capaz de calcular las direcciones principales de los surcos y valles de una huella dactilar**. Aplicando esta red al entrelazado de los hilos del cuadro también se conseguiría sustituir la segunda fase mencionada dentro del algoritmo.

Para ello se deberá hacer un estudio sobre el estado del arte en estas materias, con el objetivo de encontrar alguna red de uso libre que se pueda adaptar nuestro contexto para recibir las mismas imágenes de rayos X de los cuadros que emplea el trabajo original.

5 ESTADO DEL ARTE

Los sistemas biométricos, debido a la necesidad constante de actualizar y extender las tecnologías existentes en la seguridad y en contra de los fraudes de identidad, están en continuo desarrollo. La **biometría**, que tiene como definición aquella rama de la ciencia que estudia características tanto físicas como en la conducta de las personas que pueden medirse es, por tanto, tema de estudio por universidades y organismos fiscales a lo largo de todo el mundo. Podemos encontrar ejemplos de biometría en nuestros propios dispositivos móviles desde hace casi una década, cuando Apple presentó el *iPhone 5s*, el cual incluía el primer sensor de huellas dactilares destinado para la autenticación de usuarios en el mercado [21]. Otros ejemplos, usados más allá del ámbito comercial serían las medidas del iris del ojo, rasgos faciales, la firma y la voz.

El estudio de las huellas dactilares es uno de los más importantes en este campo, pues lleva en continuo desarrollo desde hace más de cien años, cuando se comenzaron a implementar en el ámbito forense con la creación de registros de huellas de criminales con el objetivo de identificarlas en posibles delitos reincidentes [22]. El origen de este fuerte interés suscita en que, debido a que las huellas se generan por el movimiento del feto durante su desarrollo, la distribución de los datos relevantes que las componen (llamados en la literatura como *minutiae*) y sus posiciones relativas garantizan probabilísticamente que sea imposible encontrar dos individuos con una misma huella dactilar, lo que lo convierte en una métrica perfecta para métodos de identificación personal [23].

5.1 Identificación de huellas dactilares

Al medir una huella dactilar se analiza en qué forma se curva la piel de los dedos, formando líneas con patrones únicos para cada persona. Usando la nomenclatura habitual de la literatura, los **surcos** (***ridges***, en inglés) son por tanto los relieves dentro de la huella y los **valles** (***valleys***) los huecos generados entre ellos. Los elementos de los cuales se pueden extraer algún tipo información relevante para el estudio de biometrías de una huella dactilar se clasifican en tres niveles:

- **Nivel 1**. Son características globales sobre la forma de los surcos. Llamamos ***loop*** cuando un surco se curva y ***delta*** cuando dos surcos distintos se encuentran, formando un triángulo.

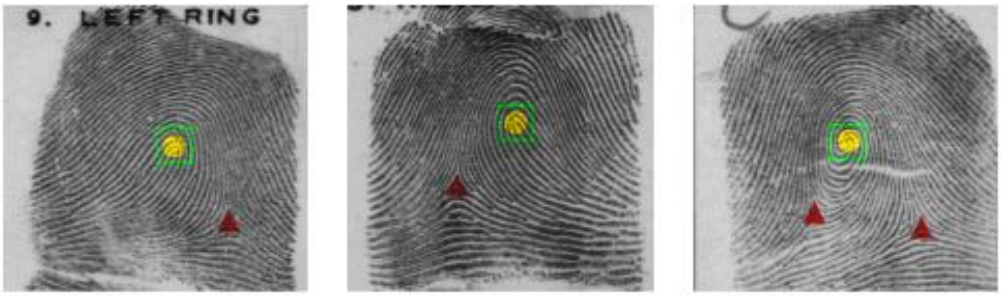


Figura 5-1. Características de nivel 1. Los círculos amarillos representan loops y los triángulos rojos deltas [24].

- **Nivel 2**. Son las conocidas ***minutiae*** (minucias en español, aunque se usa su traducción inglesa). Son consideradas *minutiae* cambios concretos en la estructura de un surco y existen varios tipos,

tratándose de los más importantes los **finales** y las **bifurcaciones** de los surcos (***ridge ending*** y ***bifurcations*** en inglés). La mayoría de los sistemas y softwares desarrollados por la comunidad científica se han centrado en identificar estas *minutiae*, pues sus localizaciones y posiciones relativas dentro de la huella son las que otorgan su autenticidad.

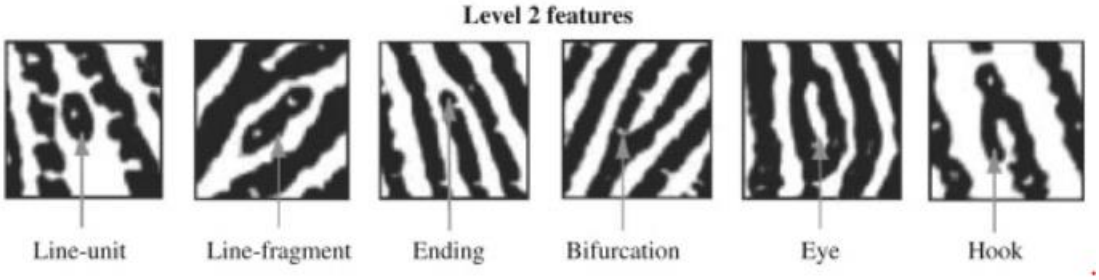


Figura 5-2. Características de nivel 2 [25].

- **Nivel 3**. Son aquellas características de más bajo nivel, como pueden ser los mismos **poros de la piel**. Aunque para incluirlos en la caracterización de las huellas hagan falta imágenes de alta resolución (de más de 1000dpi siendo 500 dpi el usado comúnmente en la actualidad) ya se están desarrollando algoritmos de DL que los incluyen. Este es el caso de la red propuesta por Zhenzhen Yang *et al* [26].

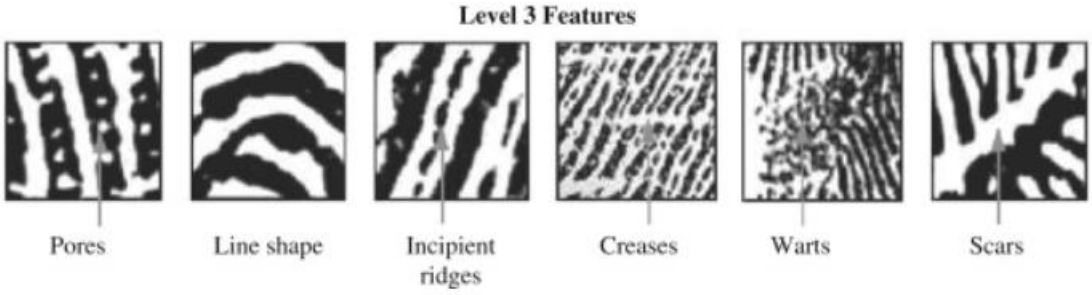


Figura 5-3. Características de nivel 3 [25].

Los dispositivos electrónicos encargados de capturar las imágenes, extraer la información relevante e identificar la persona de la que procede se conocen con el nombre de ***Automatic Fingerprint Identification Systems*** (**AFIS** en lo sucesivo). El proceso de identificación de huellas dentro de un AFIS se realiza siguiendo los pasos de la Figura 5-4, mostrada a continuación:

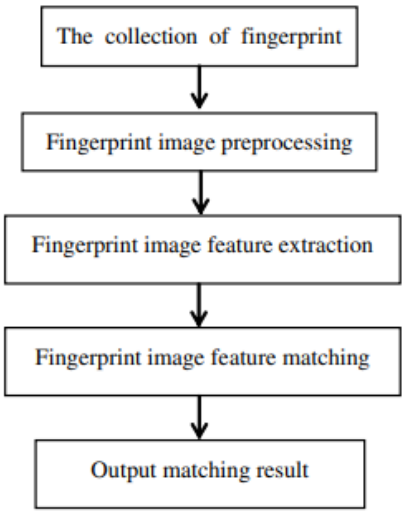


Figura 5-4. Diagrama de flujo de un AFIS [27].

Una vez recopilada la imagen de la huella dactilar, es de gran importancia tratarla en la etapa de **pre-procesamiento**. Pueden ser muchos los factores externos capaces de generar imperfecciones en la primera etapa, lo cual puede acarrear pérdidas de resolución y de minutiae, así como la generación de minutiae espurias que degraden considerablemente el desempeño global del AFIS. La etapa de pre-procesamiento se divide de forma general, por tanto, en cuatro pasos [27]:

1. Segmentación de la imagen de huella dactilar.
2. Realce de la imagen.
3. Binarización de la imagen.
4. Refinamiento de la imagen.

La literatura al respecto es muy extensa y para la gran mayoría de aplicaciones esta etapa es considerada un paso intermedio, más que una aplicación en sí misma. Al tratarse de pre-procesado, entre sus objetivos están el de acondicionar la imagen para las etapas posteriores mediante la eliminación de ruido o espurios, realce de *minutiae* o la obtención de una representación simplificada de la imagen (*ridge skeletons* o *thinned maps* [28]). Esto hace que no exista un estándar y que sea posible encontrar algoritmos que superpongan algunos de estos pasos o incluso los descarten, es por ello que, tras un estudio exhaustivo sobre la literatura existente, este trabajo se centrará únicamente en analizar el estado del arte de 2 métodos de pre-procesado: estimación del campo de orientación (***Orientation Field Estimation***) y realce (***Enhancement***) de imágenes de huellas.

5.2 Estimación del Campo de Orientación de imágenes de huellas dactilares

El campo de orientación de una imagen de huella dactilar, ***Orientation Field*** en inglés, (**OF** en lo sucesivo) es una matriz discreta cuyos valores almacenan vectores tangentes a la dirección media de los surcos de la huella en una región.

Figura 5-5. Imagen de huella dactilar (izquierda) y su correspondiente OF (derecha) [28].

La literatura en este campo es, una vez más, muy extensa. Para arrojar un poco de luz nos apoyamos en un estudio del 2019 realizado por Weixin Bian *et al*. en el que se hace un registro histórico de los trabajos más relevantes en el estudio de la estimación del OF [29], en él podemos ver que dichos trabajos se agrupan bajo tres categorías: basados en **gradiente**, **modelos matemáticos** o **aprendizaje máquina**. Los métodos basados en el gradiente fueron los primeros en aparecer y datan de finales de los años 80, sin embargo, la gran mayoría de algoritmos introducidos en la última década son del grupo de ML o DL.

A la hora de hacer una comparativa, esta última categoría se desmarca favorablemente de las demás cuando se trabaja con imágenes de baja calidad, pues métodos de DL basados en redes convolucionales son capaces de aprender variaciones del tipo de rotaciones, traslaciones y distorsiones respecto a los datos de entrenamiento [24] , [29].

Teniendo esto en cuenta, se plantea como una opción bastante razonable buscar alguna red neuronal existente que realice el cálculo del OF, pues es precisamente información sobre la direccionalidad local de los hilos (el equivalente a los surcos al extrapolar los métodos existentes a nuestro problema) la métrica buscada para sustituir el algoritmo de [1] hasta su segunda fase.

A continuación, se realizó una búsqueda exhaustiva en la literatura con el objetivo de encontrar redes neuronales de obtención de OF. Fueron varios los recursos encontrados, pues son muchas las universidades investigando en este tema actualmente. Sin embargo, elegir una no fue tan sencillo como pareciera en un principio, pues a parte de intentar seleccionar una red actualizada al estado del arte se pretendía que ésta fuera de uso libre. Encontrar código libre (en páginas como *github*, *paperswithcode* o los propios portales de las universidades investigadoras) es más complicado de lo que pudiera parecer, aunque estas instituciones o investigadores apoyen a la comunidad científica con artículos en donde se muestran las mejoras de sus redes propuestas respecto al estado del arte en el momento de su publicación, no suelen acompañarlos con el código. El motivo de esto puede deberse a que decidan reservarlos simplemente como parte de la propiedad intelectual de los propios autores o para competiciones futuras como la serie de competiciones LivDet: Liveness Detection Competition, celebrada cada dos años desde el 2009 hasta la actualidad o la más trascendental en este ámbito: Fingerprint Verification Competition [30], [31].

En base a estas restricciones, se han seleccionado dos redes: **FingerNet** y **Orienet**.

5.2.1 FingerNet

Esta red diseñada por Yao Tang *et al.* para la extracción de minutiae y presentada en la *IEEE International Joint Conference on Biometrics* de 2017 consiguió superar los resultados obtenidos por los demás modelos de DL hasta entonces [32]. A parte del OF, obtenido como paso intermedio, la red devuelve una versión realzada de la imagen.

Su código se puede encontrar en: https://github.com/592692070/FingerNet.

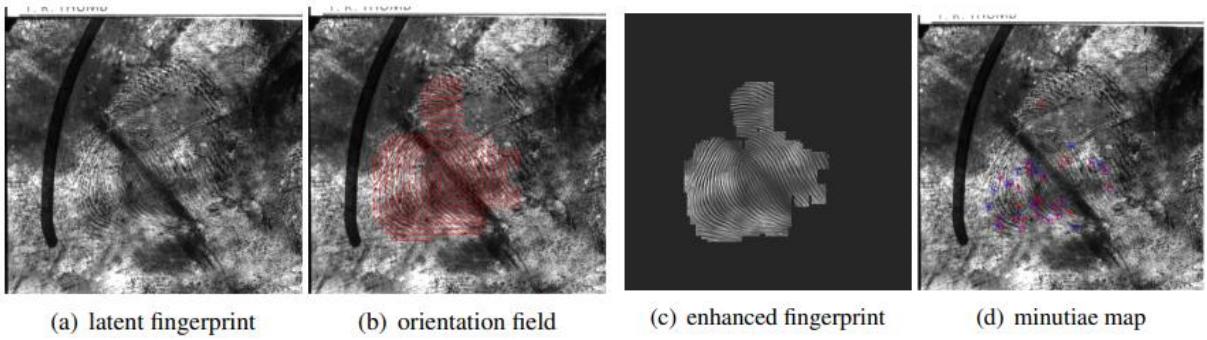


Figura 5-6. Ejemplo del resultado de FingerNet [32].

Estos resultados se consiguieron combinando el conocimiento previo del área y métodos de aprendizaje profundo: en una primera fase se adaptaron métodos tradicionales de estimación del OF, segmentación, realce mediante filtros de Gabor y extracción de minutiae a estructuras de filtros kernel convolucionales, con pesos fijos. En la segunda fase se expandirían estos filtros a redes convolucionales entrenables, que tomarían esos pesos fijos como valores iniciales en el proceso de entrenamiento para mejorar el rendimiento de la red final unificada.

En la Figura 5-7 se muestra la curva de ***Cumulative Match Characteristic*** o CMC, forma en la que usualmente se compara el rendimiento de los métodos de identificación de biometrías. Para ella se utilizó la base de datos NIST SD27 y se refleja claramente como se consiguieron resultados por encima del estado del arte del momento.

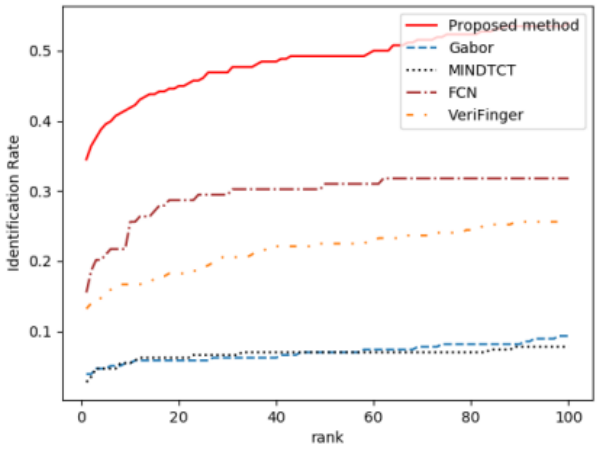


Figura 5-7. Curva CMC de diferentes algoritmos de extracción de minutiae comparados con FingerNet [32].

5.2.2 OrieNet

Esta red fue presentada por Zhenshen Qu *et al*. para la vigesimoséptima edición de la International Conference on Artificial Neural Networks celebrada en Rhodes, Grecia, en octubre del 2018 [33]. Su código se puede encontrar en: https://github.com/juniorliu95/OrieNet.

Esta red, bajo la misma premisa de FingerNet, combina una primera etapa en la que hace uso de métodos tradicionales para el pre-procesamiento de las imágenes con una red neuronal destinada a la estimación del OF.

Esta primera etapa está compuesta por un bloque de *Local Total Variation* (LTV), usado para descomponer la imagen en bloques de 64x64 píxeles, un filtro paso de banda y un filtro de Gabor. Este último emplea las dos orientaciones con mayor respuesta de entre un conjunto de 12 filtros direccionales, obtenidos mediante un método de optimización externo. A continuación, la imagen de huella realzada será entregada a una red del tipo DRNN (*Deep Regression Neural Network*).

Para la comparación de resultados se tomó la red FingerNet, la cual fue re-entrenada y testeada con la misma base de datos de esta red, la cual volvió a superar el rendimiento de los mejores algoritmos hasta la fecha.

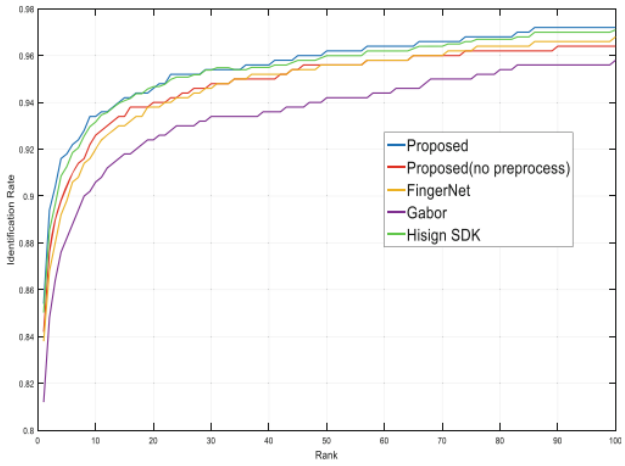


Figura 5-8. Curva CMC de diferentes algoritmos y OrieNet (azul) [33].

Ambas redes cumplen con nuestros requerimientos, pues a parte de dar un desempeño actualizado a la orden del estado de arte actual su código se puede encontrar de manera pública en *github*. Sin embargo, el campo de orientación se obtiene por ambas redes en forma de imagen, lo que haría necesario encontrar o en su defecto programar un método para obtener las direcciones principales del OF a partir de su imagen. Esta tarea excede los límites de un trabajo de este tipo, por lo que definitivamente se elegirá algún algoritmo centrado únicamente en el realce.

5.3 Realce de imágenes de huellas dactilares

Si bien es cierto que la literatura al respecto de esta materia también es amplia, resulta aún más complicado encontrar el código asociado a los artículos. Esto se debe a que el realce de las imágenes de huellas constituye en casi todos los casos la primera etapa de un algoritmo y no un objetivo final. Los algoritmos propuestos para realce buscan, por tanto, mejorar el rendimiento de otros para la estimación de OF, extracción de *minutiae* o en definitiva el que suele ser el objetivo principal de todo el desarrollo en este campo: la mejora en la robustez y resolución de los AFIS en su software de identificación.

De esta forma se ha elegido la red desarrollada por Sukesh A. y Jayanthi S. llamada FPD-M-net [34].

5.3.1 FDP-M-Net

El nombre de esta red hace referencia, tal y como explica el título de su artículo asociado, a una red de restauración (**Inpainting**) y eliminación de ruido (**Denoising**). Para ello se adapta una red convolucional existente, llamada M-Net [35].

Esta red fue diseñada para la segmentación de imágenes 3D de resonancias magnéticas cerebrales (MRI), obteniendo muy buenos resultados. FPD-M-Net elimina por tanto el primer bloque encargado de la conversión de 3D a 2D, reordenan las capas de algunos bloques y sustituye la función *softmax* en la capa final por una *sigmoid*, quedando finalmente una estructura como la que se muestra en la siguiente figura 5-9:

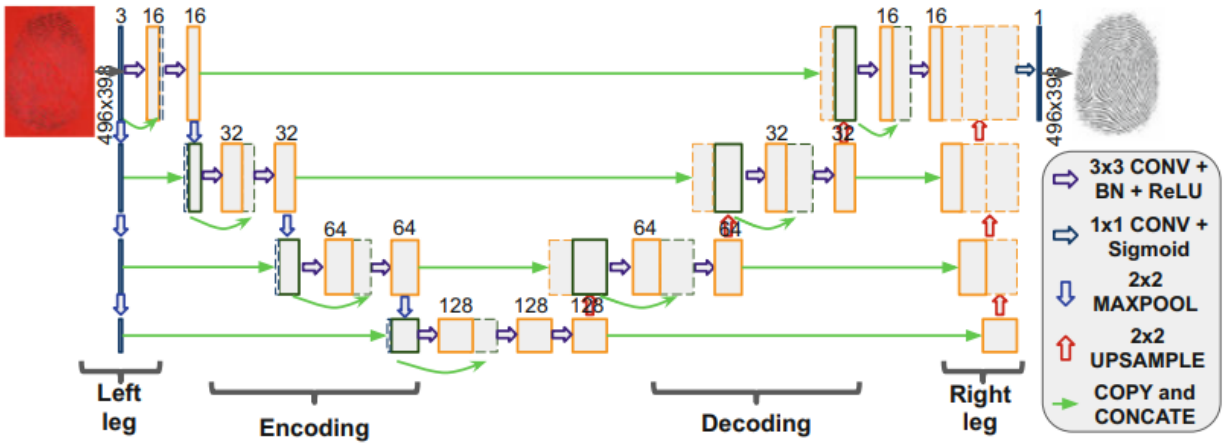


Figura 5-9. Arquitectura de FPD-M-Net [34].

En esta figura las cajas amarillas continuas representan bloques CONV-BN-ReLU, esto es una capa convolucional y una de *Batch Normalisation* seguidas de una función de activación del tipo ReLU. Entre cada dos de estos bloques se introduce una capa de *dropout* con probabilidad de 0.2. Las capas de BN se utilizan únicamente durante el proceso de entrenamiento para hacerlo más estable y rápido y las de *dropout* para prevenir un fenómeno llamado sobreentrenamiento u *overfitting*, el cual ocurre si no se entrena el modelo de forma adecuada y este no es capaz de ganar generalidad a la hora de realizar nuevas estimaciones cuando se dan nuevos datos de entrada distintos a los de entrenamiento. Las capas de *dropout*, por tanto, prueban a descartar capas enteras bajo una cierta probabilidad en cada una de las iteraciones del proceso de entrenamiento [36].

Figura 5-10. Ejemplos de resultado de FPD-M-Net.

En la figura anterior se muestra un ejemplo de ejecución de la red. En ella las figuras de la izquierda muestran las imágenes originales y las de la derecha su resultado tras pasar por ésta. Puede verse como se realizan las funciones tanto de eliminación de ruido como de restauración de las huellas.

6 IMPLEMENTACIÓN DEL PROYECTO

Una vez se han explicado los conceptos básicos de DL necesarios para entender el funcionamiento de FPD-M-Net, la red escogida a la cual entregaremos imágenes de lienzos para que las realce, detallaremos como ha sido la implementación de este proyecto. En un primer lugar mencionaremos las plataformas y software empleados y a continuación, la aportación de este trabajo al código original y el proceso mediante el cual se adaptó para que cumpliera con nuestras necesidades, así como para que superara ciertas adversidades que fuimos encontrando.

6.1 Entorno

El código original de la red se descargó desde **GitHub**. Esta página se trata del mayor repositorio de código online, con más de 31 millones de usuarios registrados en la actualidad [37]. A parte de albergar infinidad de tutoriales y códigos de uso libre, la característica más interesante de GitHub reside en que está basada en un sistema de control de versiones Git. Mediante este se guardan copias de todas las versiones de un programa, siendo todas ellas accesibles durante el desarrollo de los proyectos, lo que facilita la cooperación de forma remota, así como la depuración de errores.

Figura 6-1. Logo de GitHub [37].

Como intérprete para ejecutar este código se ha elegido también otro recurso ampliamente conocido como es **Jupyter Notebook**. Este basa su funcionamiento en cuadernos, fáciles de compartir por medio de Google Drive y de ejecutar, pues se hace desde el propio navegador web. Jupyter soporta más de 40 lenguajes de programación como R, Julia, Scala y Python, siendo este último el más usado actualmente por ingenieros e investigadores de datos y DL y por tanto en el que está escrito nuestra red.

Asimismo, haremos uso de Google Colaboratory (conocido comúnmente como **Google Colab** o solo Colab) para abrir el repositorio de GitHub y ejecutar la red. La gran ventaja que nos aporta Colab es que permite gratuitamente hacer uso de CPUs o GPUs potentes de forma remota, esto es esencial cuando se trabaja con redes neuronales como se ha comentado en los capítulos introductorios.

Figura 6-2. Logo de Jupyter [38].

Finalmente, pues ya se ha adelantado que el software de programación empleado será Python, mencionaremos brevemente las principales librerías empleadas.

- ***Keras***: Se trata de uno de los *frameworks* de aprendizaje automático más usados en la actualidad. ***Tensorflow,*** otro *framework* de código abierto, es compatible con éste y entre los dos reúnen todos los métodos y herramientas con las que está desarrollada y entrenada la red.
- ***os* y *sys*:** Son módulos de sistema. La librería *os* nos permite ejecutar órdenes del sistema operativo tales como abrir, cambiar, crear o eliminar directorios. Con sys podemos obtener parámetros y funciones específicas del sistema, como pueden ser los argumentos entregados por línea de comandos al lanzar la ejecución del programa.
- ***glob:*** Empleado para la búsqueda de rutas a archivos bajo condiciones específicas.
- ***numpy:*** Módulo ampliamente conocido, sirve para trabajar con vectores y matrices multidimensionales y realizar operaciones matemáticas sobre ellos.
- ***PIL, imageio y cv2:*** Serán los módulos empleados para trabajar con imágenes.

6.2 Implementación

6.2.1 Cuaderno de Google Colab: TFG.ipynb

El primer paso para la implementación de este trabajo será la creación de un cuaderno de colab desde el cual se pueda ejecutar la red. Este nos servirá como entorno de trabajo y en el realizaremos los cambios pertinentes como crear las carpetas necesarias, añadir nuestras propias imágenes o eliminar archivos innecesarios, para que el trabajo funcione correctamente. El código completo se puede consultar en el **Anexo A** de este trabajo.

```
[1]  !git clone https://github.com/adigasu/FDPMNet.git

     Cloning into 'FDPMNet'...
     remote: Enumerating objects: 65, done.
     remote: Total 65 (delta 0), reused 0 (delta 0), pack-reused 65
     Unpacking objects: 100% (65/65), done.

[2]  #Importamos el modulo del sistema operativo (os):
     import os

[3]  os.chdir("FDPMNet/")
```

Código 6-1. Clonación del repositorio de GitHub de la red original.

Con la ejecución de los 3 primeros bloques, mostrados en el **código 6-1**, clonamos el repositorio de github de los autores de la red y nos situamos dentro de éste (capeta FDPMNet). Cabe destacar que al clonar un repositorio se crea una copia local en el entorno de ejecución que estemos usando en ese momento en Google Colab, que se borrará al desconectarse de éste.

Al ejecutar el archivo test.py se toman las imágenes de su carpeta homónima y se entregan a la red, los resultados (imágenes realzadas por la red) se guardarán en la carpeta *Results*. A continuación, eliminaremos el archivo test.py para reemplazarlo por otro de igual nombre, pero actualizado para trabajar bajo nuestras condiciones, así como la carpeta *Results*, que contiene un resultado de ejemplo. Esta carpeta se genera con la ejecución del programa y nosotros la usaremos como guía para validar que nuestros cambios no afecten al desempeño original de la red.

```
[4]  #Borramos los archivos obsoletos.
     if (os.path.isfile("test.py")):
       print("Borramos archivo test.py")
       os.remove("test.py")
     else:
       print("Archivo test.py no existe")

     Borramos archivo test.py

[5]  #Borramos carpeta de Results (para comprobar que se está ejecutando bien el código de la red)
     os.chdir("test/")

     import shutil
     if (os.path.exists("Results")):
       print("Borramos carpeta Results")
       shutil.rmtree("Results")
     else:
       print("Carpeta Results no existe")

     #Volvemos a la carpeta FDPMNet:
     os.chdir("../")

     Borramos carpeta Results
```

Código 6-2. Para eliminar el archivo test.py y carpeta *Results* originales.

Finalmente, para trabajar de forma local, se copian desde Google Drive las imágenes de los cuadros a una carpeta de nombre *ImagenesTFG* dentro de la carpeta *test*, pues de ahí tomará el programa las imágenes a

la hora de pasarlas por la red.

```
[6]  #Montamos el Drive para la carga de las imágenes y los archivos actualizados:
     from google.colab import drive
     drive.mount("/content/gdrive/", force_remount=True)

     Mounted at /content/gdrive/

[7]  #Movemos el archivo test.py actualizado:
     origen_arch  = "../gdrive/MyDrive/TFG/test.py"
     destino_arch = "../FDPMNet"
     shutil.copy(origen_arch, destino_arch)

     '../FDPMNet/test.py'

[8]  #Cuando pueda usar las imagenes: Las copiamos todas en la carpeta para que puedan ejecutarse.
     #Se crea la carpeta: ImagenesTFG
     os.chdir("test/")
     if os.path.exists("ImagenesTFG"):
       print("Carpeta ya existe")
     else:
       os.mkdir("ImagenesTFG")
     os.chdir("../")

     origen_arch  = "../gdrive/MyDrive/TFG/Imagenes"
     destino_arch = "../FDPMNet/test/ImagenesTFG"

     files = os.listdir(origen_arch)
     for fname in files:
       shutil.copy(os.path.join(origen_arch,fname), destino_arch)
```

Código 6-3. Para hacer una copia local de archivos de Drive.

Una vez ejecutadas todas estas celdas estará preparado el entorno de trabajo con nuestro archivo test.py actualizado y nuestras imágenes de cuadros cargadas. Un ejemplo de los cambios en el entorno nada más ser clonado de GitHub y tras la ejecución del cuaderno se muestra en la siguiente figura:

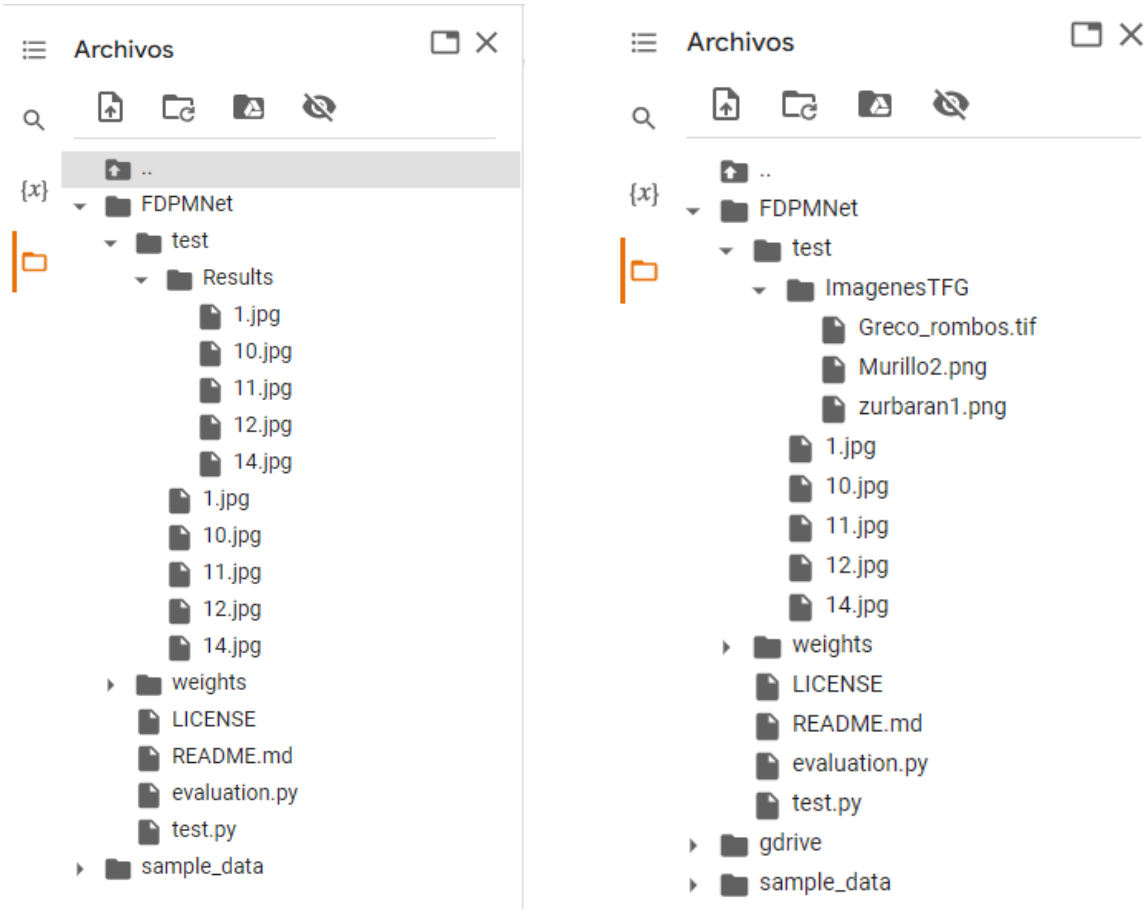


Figura 6-3. Entorno en colab (TFG.ipynb) tras clonar el repositorio de GitHub (izquierda) y después de ejecutar el código (derecha).

6.2.2 Cambios en el código de la red: *test.py*

En este apartado detallaremos los cambios realizados sobre el código *test.py* original en el orden lógico en el que se fueron implementando, en función de las necesidades que fueron apareciendo, hasta que se obtuvieron resultados competentes.

Para ello será necesario tener una idea global de cómo funciona la red en su versión original, prestando especial interés en las dimensiones de las imágenes que admite a la entrada y el tratamiento que éstas reciben, pues adelantamos serán objetivo de gran parte de las modificaciones que tendremos que realizar. Un esquema del flujo de la red en forma de diagrama de bloques se muestra en la siguiente figura:

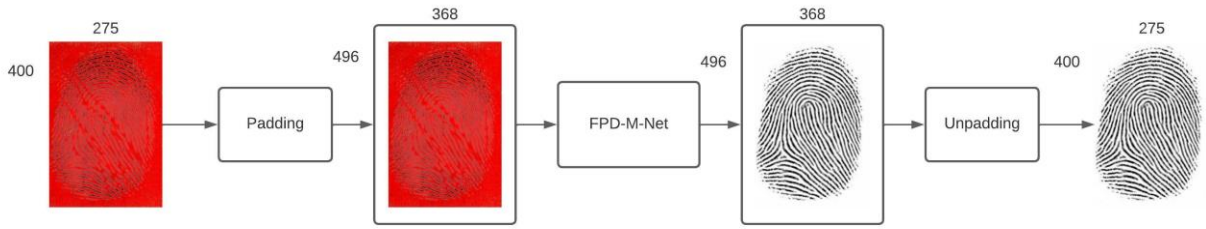


Figura 6-4. Diagrama de flujo del programa original.

Atendiendo a dicho diagrama, vemos que las imágenes de prueba proporcionadas tienen unas dimensiones de 275x400 píxeles (todas pertenecen a misma la base de datos empleada tanto para el entrenamiento como la validación, la cual es la *Chalearn LAP Inpainting Competition Track 3* [39], de la cual hablaremos más adelante) a las cuales se aplica un ***padding*** (con la llamada a la función ***padData***), de modo que las dimensiones que admite la red a su entrada son de 368x496 píxeles.

Una vez se obtiene la imagen de huella dactilar realzada se realiza un **unpadding** de forma que se obtiene una imagen final a la salida acorde a las dimensiones de entrada (275x400 de nuevo).

A continuación, iremos presentando los cambios que se fueron haciendo a partir de esta base. El código completo se puede encontrar en el **Anexo B**.

6.2.2.1 Primera actualización: parcheado de imágenes grandes

A la hora de introducir nuestras imágenes en la red cabía esperar que sus dimensiones no coincidieran con aquellas para las que ésta estaba diseñada, es por ello que parece lógico que la primera actualización al programa sea añadir métodos que nos ayuden a dividir estas imágenes (que catalogaremos como "grandes") en parches cuyo tamaño sí admita la red.

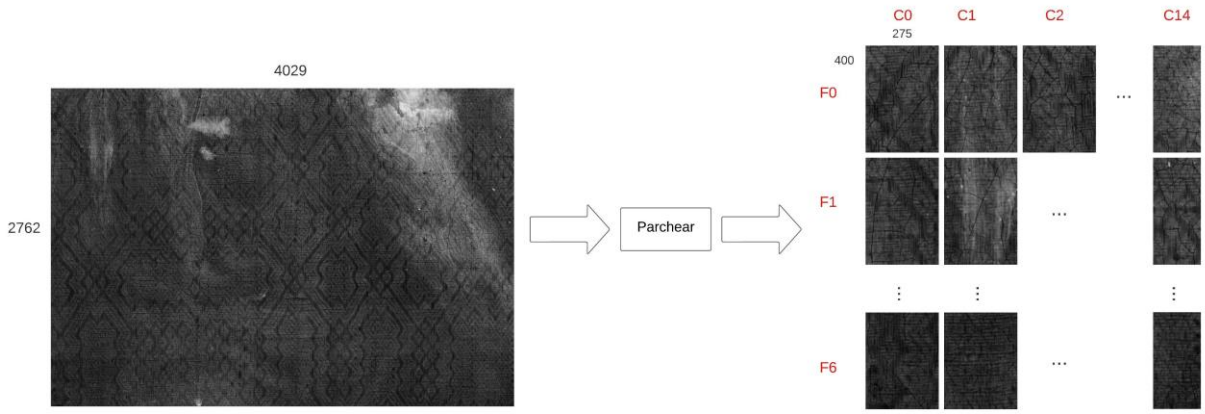


Figura 6-5. Parcheado de la imagen Murillo2.jpg.

El razonamiento es bastante inmediato: dividimos cada imagen en parches de tamaño 275x400 píxeles al igual que los de la base de datos empleada, a los cuales se aplicarían las mismas técnicas de relleno (padding). Con esto no solo conseguimos mantener las mismas dimensiones de 368x496 píxeles con las que trabaja la red original, sino que lo haga correctamente al centrar las imágenes tal y como explican los autores del artículo en [34].

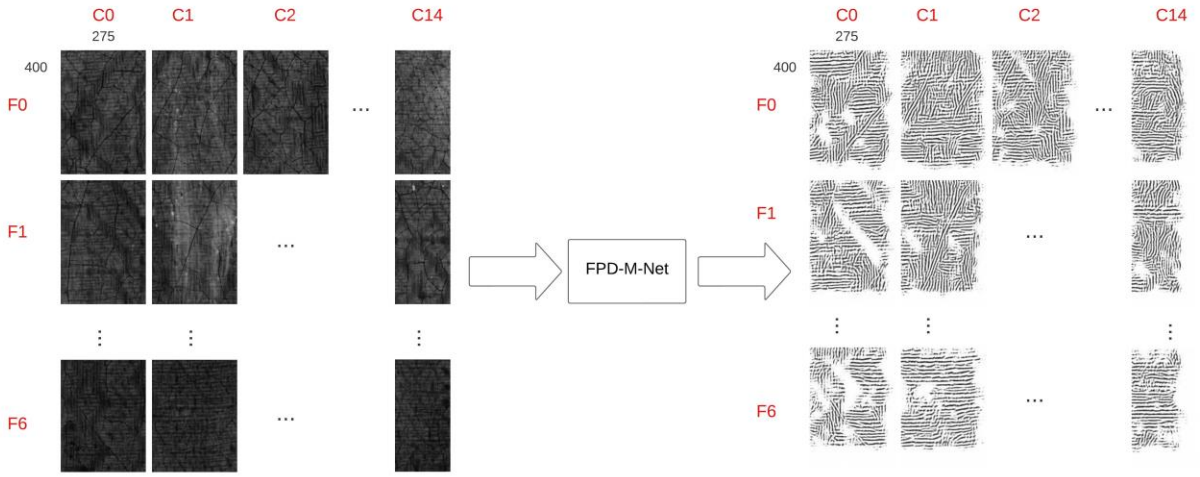


Figura 6-6. Parches de la imagen Murillo2.jpg al pasar por la red.

Finalmente, bastaría con unir todas versiones realzadas de los parches en una nueva imagen, la cual sería equivalente a haber pasado la imagen grande directamente por la red.

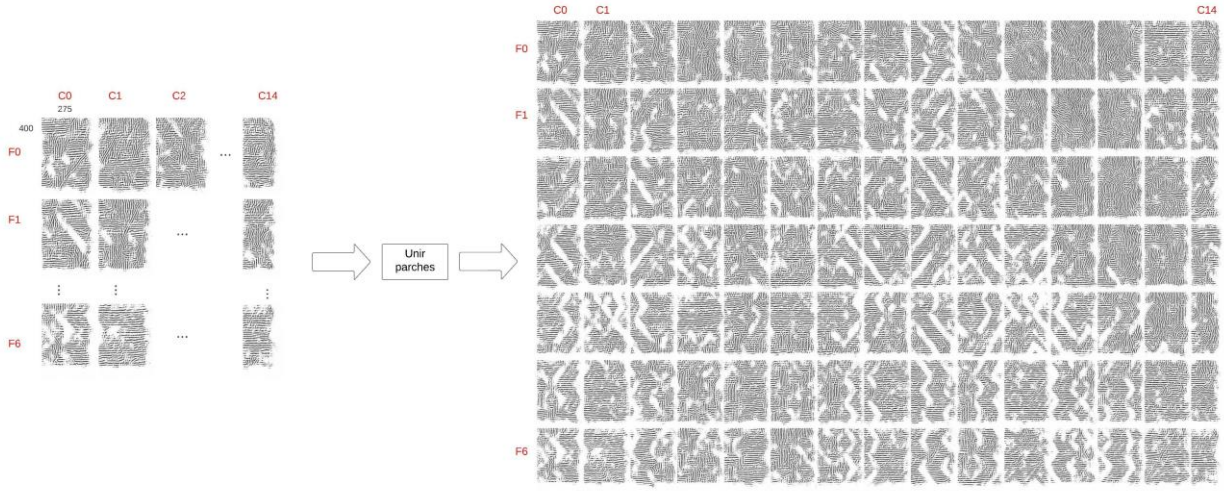


Figura 6-7. Unión de los parches de Murillo2.jpg en la imagen grande realzada.

Para lograr todo esto se crearon las siguientes nuevas funciones:

- **ConvertToJPG**: Simplemente se encarga de convertir las imágenes importadas a formato .jpg, de modo que puedan ser tratadas con normalidad por el resto del programa.
- **parchear**: Esta función realiza el parcheado de la imagen proporcionada en una carpeta nueva con su mismo nombre.
- **unirParches**: Une todas las imágenes de una carpeta en una nueva imagen, aunque se utilizará únicamente para la reconstrucción de la imagen realzada final.

Además de editar la función *Main* del programa *test.py* para que difiera si la imagen a tratar es grande o

no, en cuyo caso realizaría una ejecución normal sin hacer uso de ninguna de las actualizaciones mencionadas.

Con esta nueva versión del programa, se probaron tres imágenes de cuadros, que llamaremos Greco_rombos, Murillo2 y zurbaran1. Algunos datos destacables sobre estas imágenes "grandes" se muestran en la siguiente tabla:

Nombre de la imagen	**Alto (Height)**	**Ancho (Width)**	**Nº de parches**	**Nº de filas**	**Nº de columnas**
Greco_rombos.jpg	2048	2048	48	6	8
Murillo2.jpg	2762	4029	105	7	15
zurbaran1.jpg	1956	1453	30	5	6

Tabla 6-1. Imágenes grandes empleadas en la primera actualización del programa.

Tras la ejecución del programa, el entorno de ejecución quedó como se muestra en la Figura 6-8. Las imágenes realzadas obtenidas como resultado se muestran en las figuras posteriores:

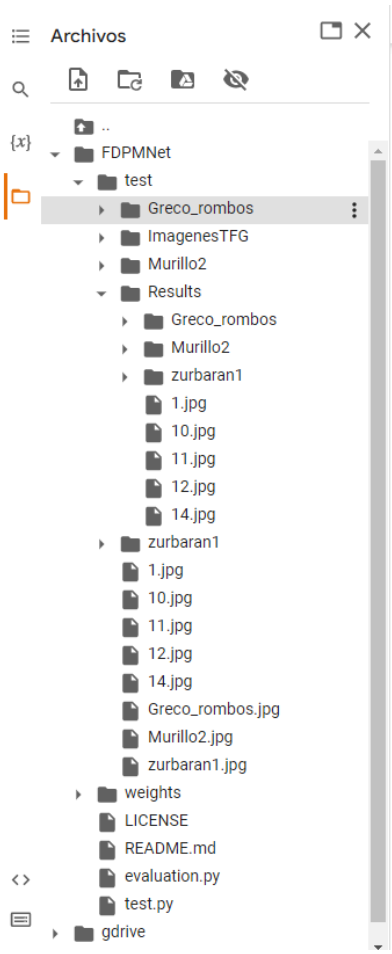


Figura 6-8. Entorno en colab (TFG.ipynb) tras ejecutar el código de la primera actualización.

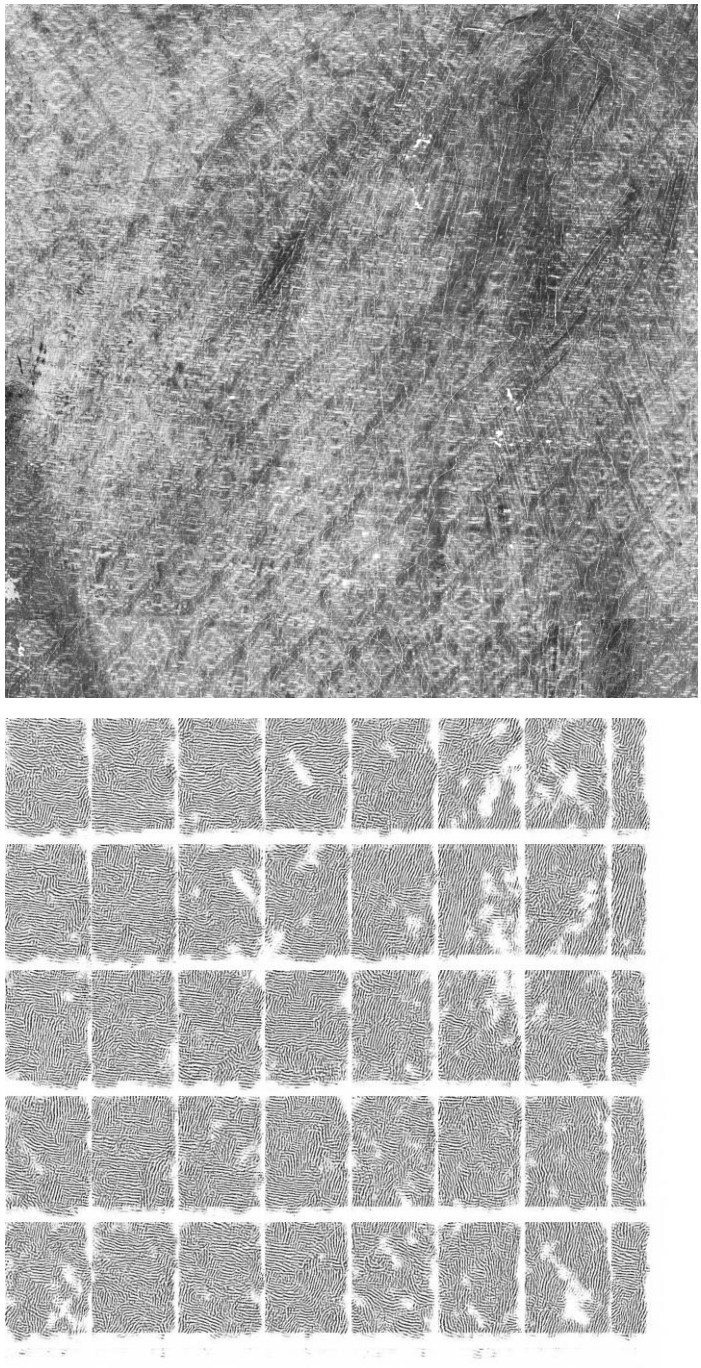


Figura 6-9. Imagen Greco_rombos original (arriba) y tras ser realzada por la red (abajo).

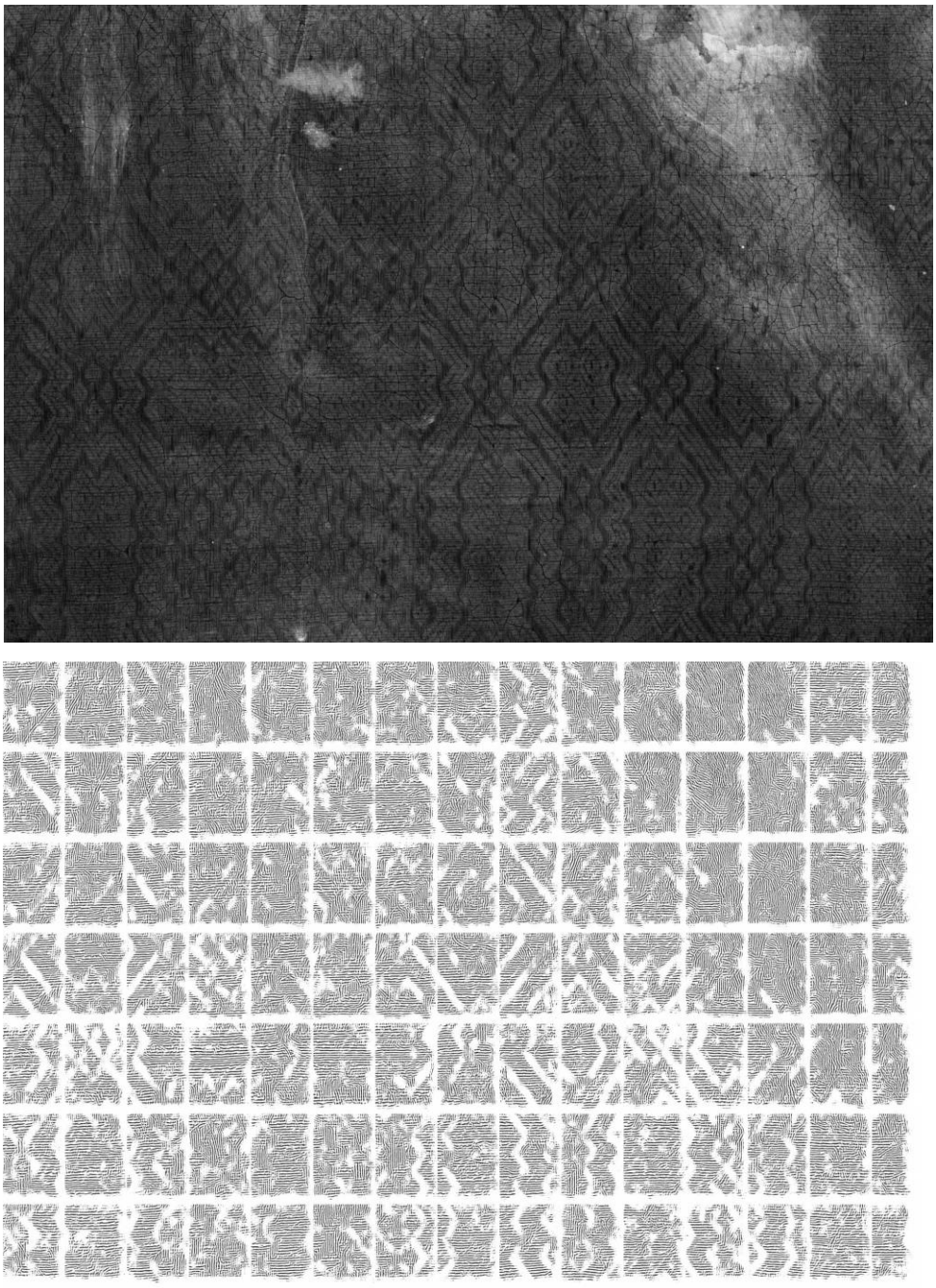


Figura 6-10. Imagen Murillo2 original (arriba) y tras ser realzada por la red (abajo).

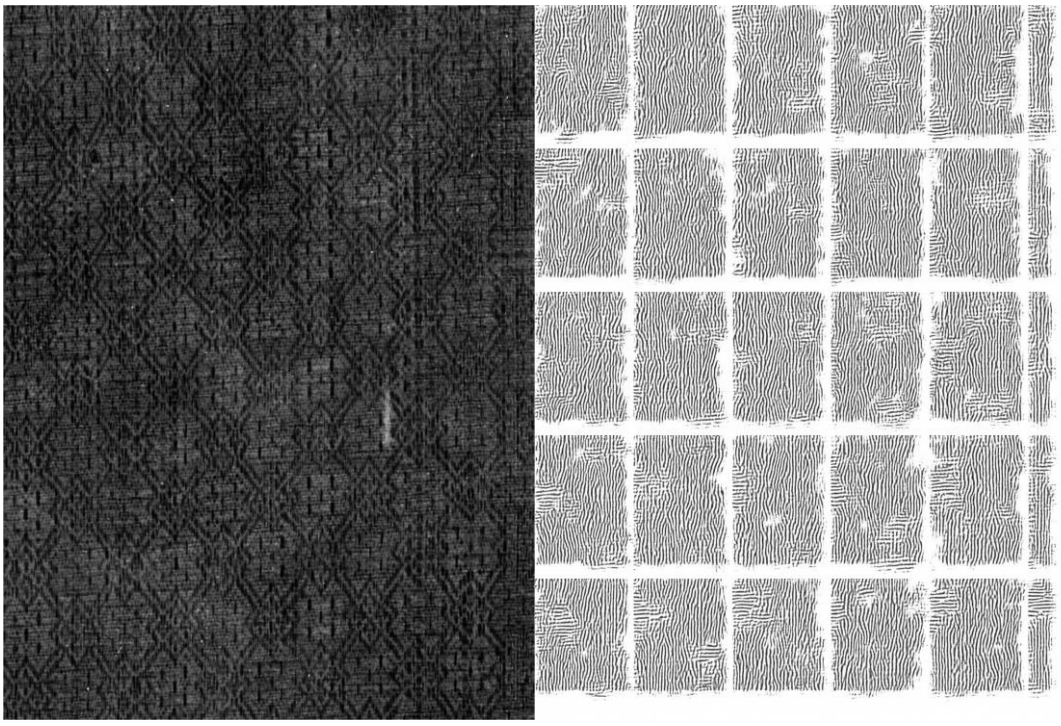


Figura 6-11. Imagen zurbaran1 original (izquierda) y tras ser realzada por la red (derecha).

En vista a estos resultados podemos arrojar un par de conclusiones acerca de esta primera actualización del código de la FPD-M-Net:

- Puesto que las dimensiones de los parches son fijas es posible, y de hecho ocurre para estas tres imágenes, que el último parche en cada fila y columna ocupe una región muy pequeña del cuadro original. Esto se contempla en el programa a la hora de realizar el *padding*, donde en caso de recibir un parche de dimensiones menores a 275x400 aplica un doble relleno: primero añadiendo filas y columnas de ceros hasta llegar a dichas dimensiones, después el que recibiría cualquier parche en todos sus bordes hasta los 368x496 que admite la red.

  Aunque el hecho de que la red, al recibir muy poca información en estos casos no sea capaz de realzar el parche de forma totalmente correcta, esto no supone un problema ya que será suficiente con el resto del cuadro para calcular la información sobre las direcciones principales. Se podrá o bien descartar esos parches o recortar las bandas blancas, producto de ese primer relleno comentado.

- Lo que sí que parece un problema son las bandas blancas que aparecen entre los parches, pues podemos afirmar que se está perdiendo información sistemáticamente en esas zonas. También podemos afirmar que no se debe a una mala aplicación del proceso de *unpadding*, pues por ejemplo en el primer parche de la imagen zurbaran1 (ver Figura 6-11) se puede apreciar que no es una banda blanca sólida, que se podría arreglar con un simple recorte. En este caso sólo una pequeña parte es interpretada por la red como “huella” y por tanto realzada, mientras que el resto es interpretado como ruido y consecuentemente descartado.

  Analizando el origen de esto bajo la suposición de que se encuentre en el proceso de entrenamiento de la red llegamos al *dataset* empleado [39]. En él podemos ver que, aunque las huellas sufran transformaciones aleatorias, tales como rotaciones de los píxeles o rotaciones totales de 10º, la gran mayoría de estas quedan confinada en el centro de la imagen (ver Figura 6-

12). Es razonable por tanto pensar que, aunque la red rinda con el *dataset* entregado para el concurso para la que fue diseñada, no se realcen de igual forma los bordes hasta el punto de obtener bandas en las que la imagen es totalmente ignorada.

Figura 6-12. Ejemplos del dataset empleado por la red [39].

Para solucionar esto se supuso que al tomar los parches con cierto solape se lograría entregar toda la información del cuadro por la red, al repetir aquella información que la red descarta sistemáticamente en todos los parches. Bajo esta premisa se implementó el siguiente cambio en el programa.

6.2.2.2 Segunda actualización: solapado entre parches

La idea detrás de esta actualización es simple: añadir un nivel de solapado entre los parches igual a las franjas en las cuales la red no funciona correctamente. Las dimensiones de solapado, (medidas en píxeles repetidos tanto en filas como columnas con los parches anteriores) si bien pudieron haber sido calculadas de forma óptima con alguna nueva función, se obtuvieron tras varias iteraciones en un proceso de ensayo y error.

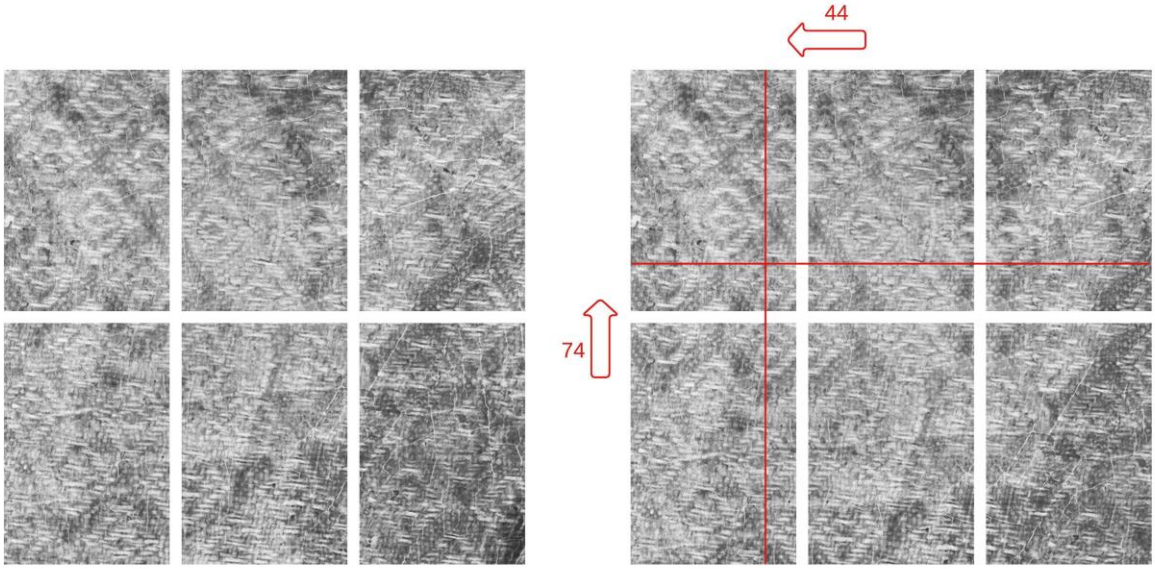


Figura 6-13. Parches de la imagen Greco_rombos sin solape (izquierda) y con solape (derecha.

Los cambios más remarcables en el código fueron en la función **parchear**, que ahora admite dos variables nuevas: *solap_x* y *solap_y*, que denotarán el número de píxeles solapados entre columnas y filas respectivamente. Con estos cambios, añadiendo un solapado de **44 píxeles entre columnas y 74 entre filas**, un análisis de las imágenes a tratar por la red se muestra en la siguiente tabla:

Nombre de la imagen	**Nº de parches**	**Nº de filas**	**Nº de columnas**
Greco_rombos.jpg	63	7	9
Murillo2.jpg	162	9	18
zurbaran1.jpg	42	6	7

Tabla 6-2. Imágenes grandes empleadas en la segunda actualización del programa.

Y los resultados fueron los siguientes:

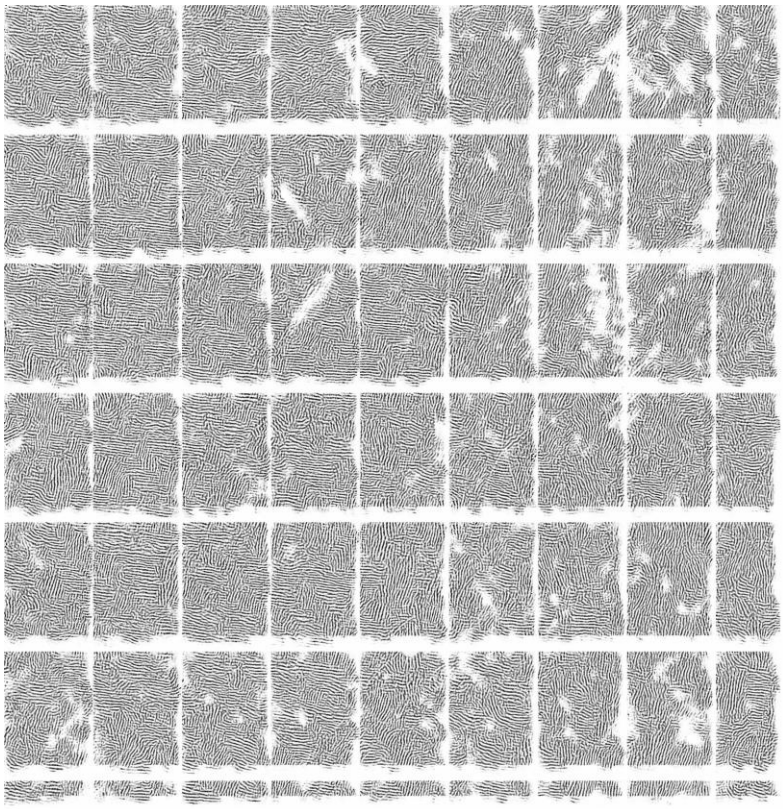


Figura 6-14. Imagen Greco_rombos realzada tras la segunda actualización del programa.

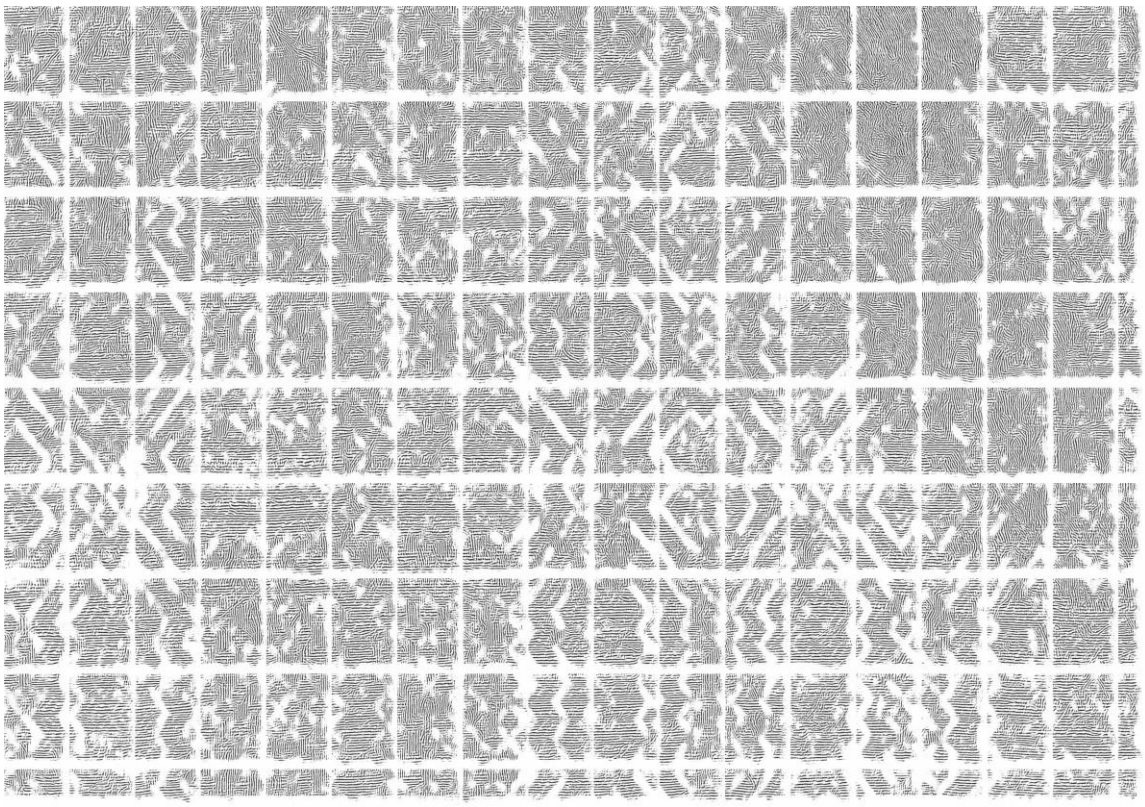


Figura 6-15. Imagen Murillo2 realzada tras la segunda actualización del programa.

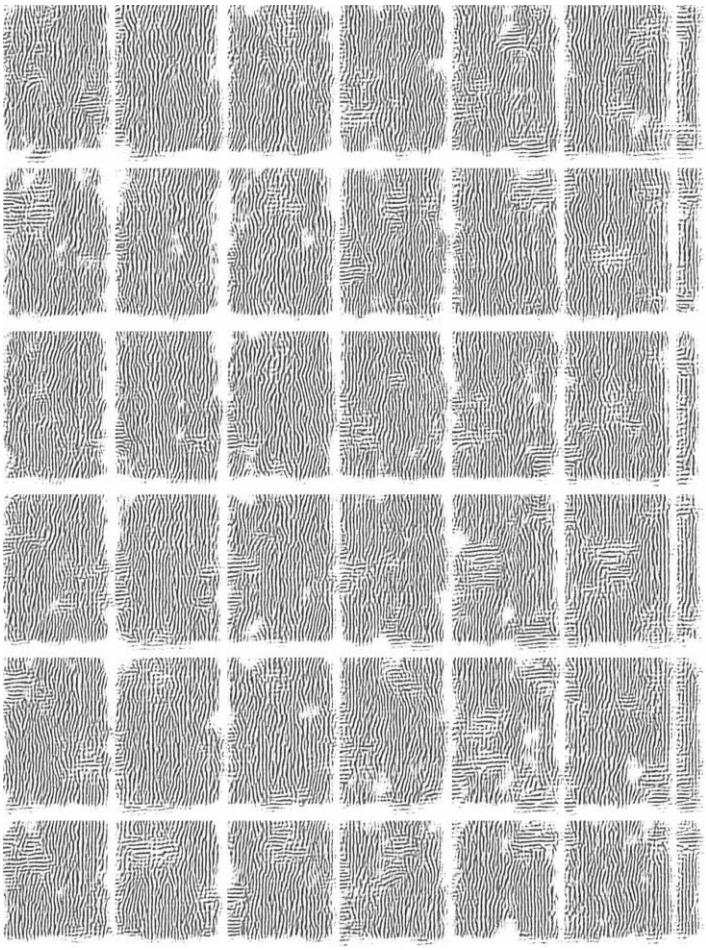


Figura 6-16. Imagen zurbaran1 realzada tras la segunda actualización del programa.

Aunque tras las nuevas adiciones al programa podamos suponer que las imágenes estarán siendo realzadas, tal y como se ve en las figuras anteriores las bandas blancas entre parches no dejan de aparecer. Esto tiene sentido, pues hemos cambiado la forma en la que se entregan las imágenes a la red, no la forma que esta tiene de analizarlas, que seguirá siendo deficiente en los bordes tanto derechos como inferiores de cada parche como habíamos analizado.

La solución final, por tanto, será eliminar esos bordes manualmente ya que ahora no estaríamos perdiendo la información del cuadro en esa zona al ser replicada en los parches mediante el solape.

6.2.2.3 Tercera actualización: eliminación de bandas blancas

Puesto que los parches en la actualización se diseñaron con medidas tales que se contrarrestaran las zonas en las que la red no trabajaba de forma correcta, resulta razonable eliminar tantas filas y columnas al final de cada parche como aquellas que se hayan replicado a la hora de formarlos.

Esto se hará al final de la ejecución del programa, donde a la hora de unir los parches ya realzados se descartarán el mismo número de filas y columnas que los que se hayan indicado para hacerlos (estas son las variables *solap_y* y *solap_x*).

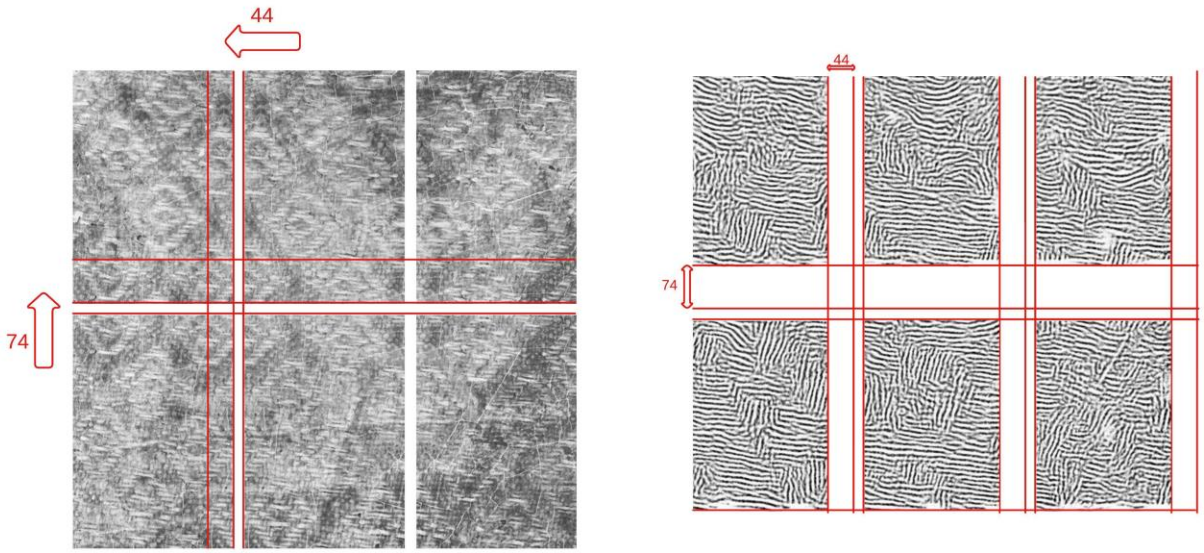


Figura 6-17. Recorte de las zonas donde la red trabaja de forma ineficiente.

Se aumentó el nivel de solapado a **92 filas y 50 columnas**. Las imágenes realzas quedan, tras la incorporación de todos los cambios mencionados, de la siguiente manera:

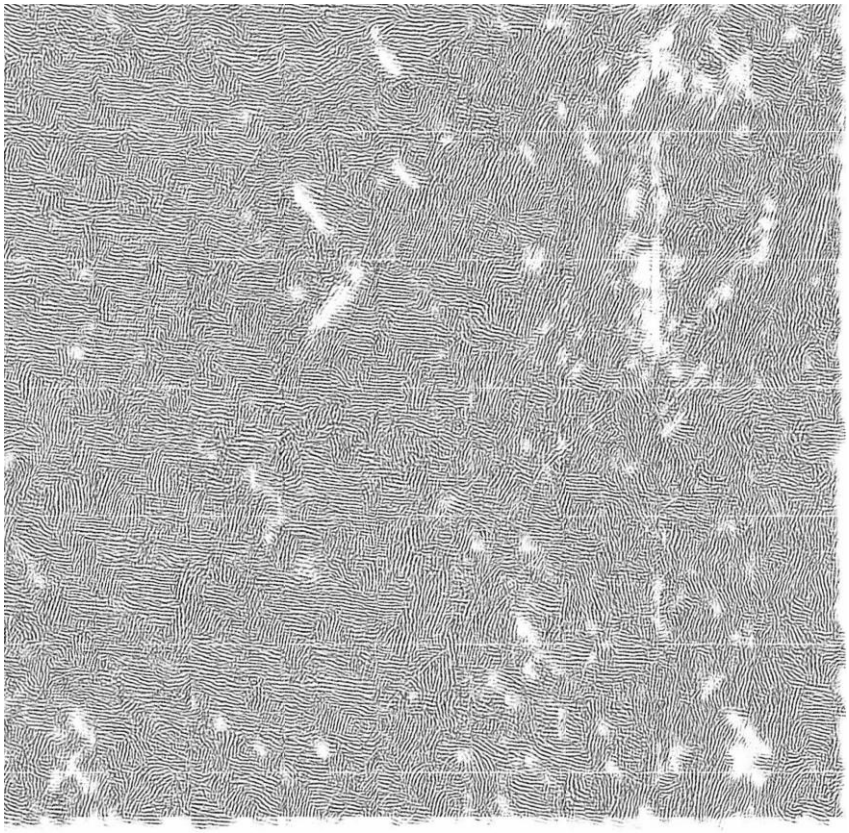


Figura 6-18. Imagen Greco_rombos realzada tras la tercera actualización del programa.

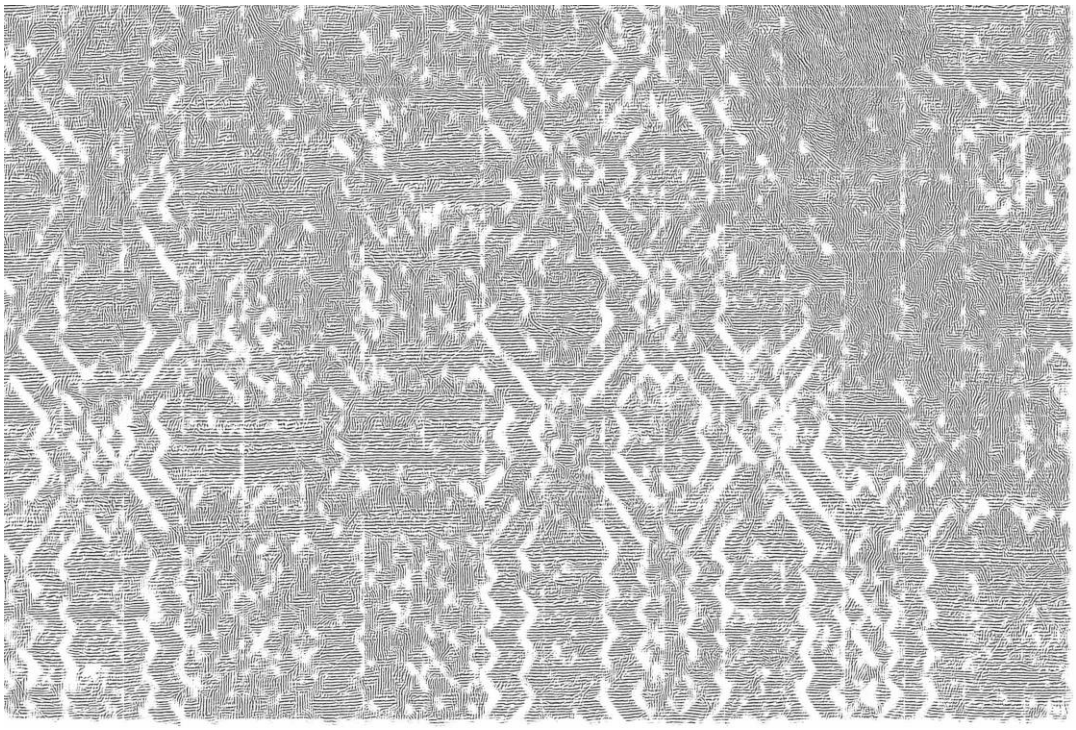


Figura 6-19. Imagen Murillo2 realzada tras la tercera actualización del programa.

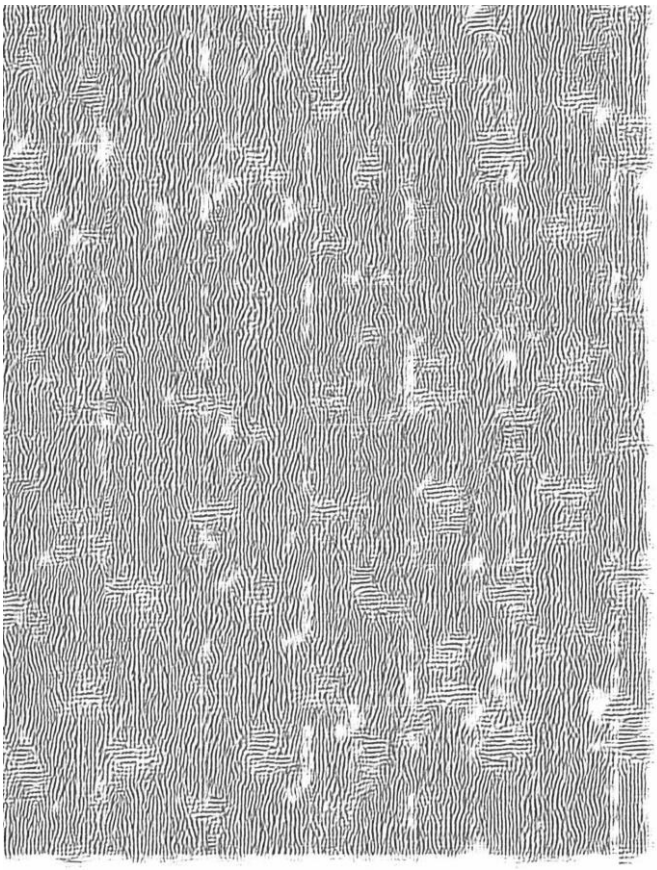


Figura 6-20. Imagen zurbaran1 realzada tras la tercera actualización del programa.

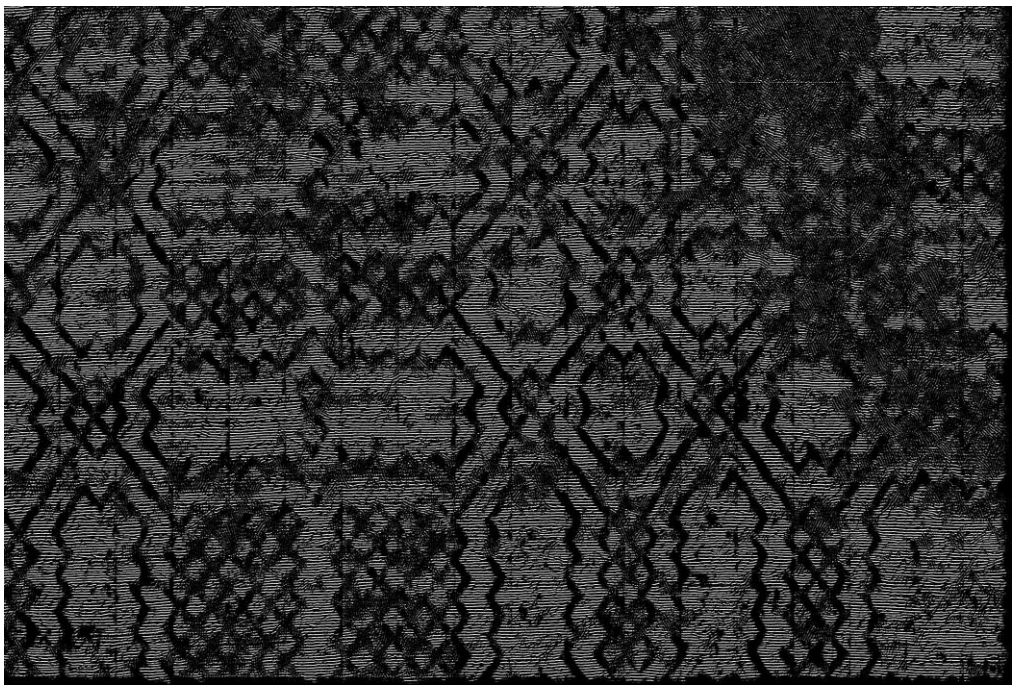
Como podemos observar, aun se pueden diferenciar a simple vista pequeñas bandas blancas en algunas zonas de las imágenes. Si bien esto no tiene por qué condicionar el resultado del algoritmo final al

incorporarlas, al compararlas con las imágenes binarias de huella dactilar del trabajo original puede ser un indicativo de que no se vaya a conseguir mejorar su desempeño. Es por ello que como actualización final se probarán algunas técnicas de pos-procesado, con el objetivo de aportar una imagen con información sobre el patrón formado por los hilos lo suficientemente característica.

6.2.2.4 Cuarta actualización: pos-procesado de la imagen realzada

Puesto que los hilos de los cuadros se distribuyen o bien en dirección horizontal, o bien en vertical, el último paso será aplicar alguna técnica de pos-procesado a la imagen realzada final para así realzarlo en alguna de estas componentes.

Primero se probó la aplicación de operaciones morfológicas típicas, empleadas comúnmente en la detección de bordes tales como la **erosión** y la **dilatación**. También la combinación de éstas, conocidas como **opening**, donde primero se aplica la erosión y a continuación la dilatación y **closing**, que sigue el orden inverso. Un ejemplo del resultado de este pos-procesado es el que se muestra en la siguiente figura:



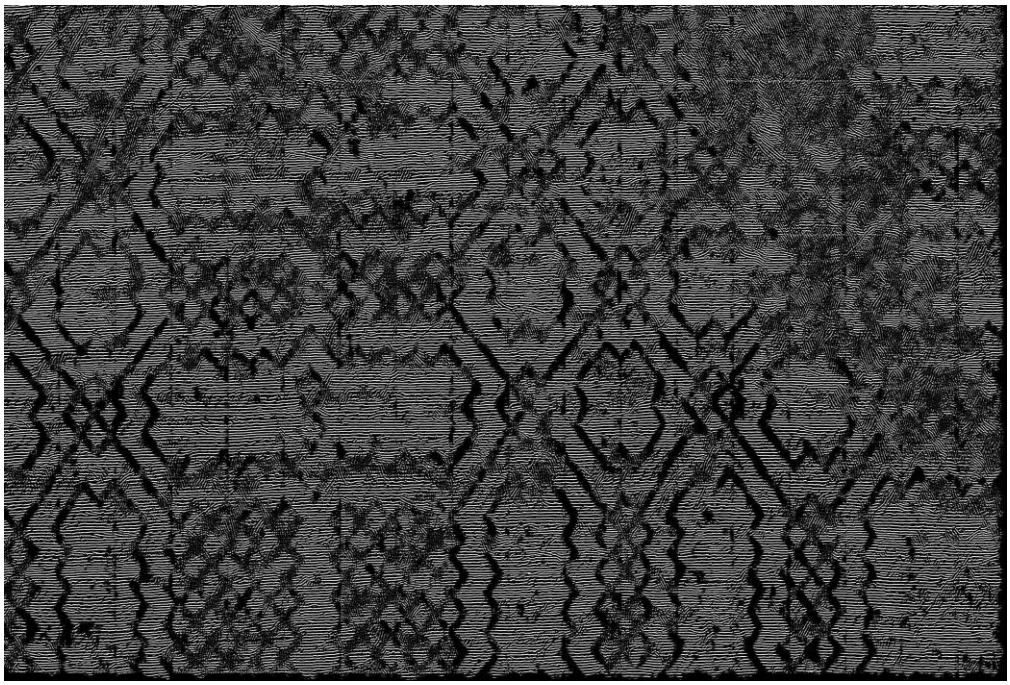


Figura 6-21. Imagen Murillo2 después de la operación open (arriba) y close (abajo).

Sin embargo, el resultado más convincente se obtuvo tras la aplicación de un filtro u **operador de Sobel**. Este también es ampliamente conocido y empleado en la detección de bordes, pues realiza una estimación del gradiente local de la imagen, tanto en dirección horizontal como vertical, tal y como buscábamos.

El filtro de Sobel se define como un kernel del siguiente tipo, siendo el kernel definido en la ecuación 6.1 el del filtro horizontal y el de la ecuación 6.2 el vertical [40]:

$$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \tag{6.1}$$

$$S_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{6.2}$$

Los resultados tras aplicarlo en nuestras imágenes realzadas por la red fueron los siguientes:

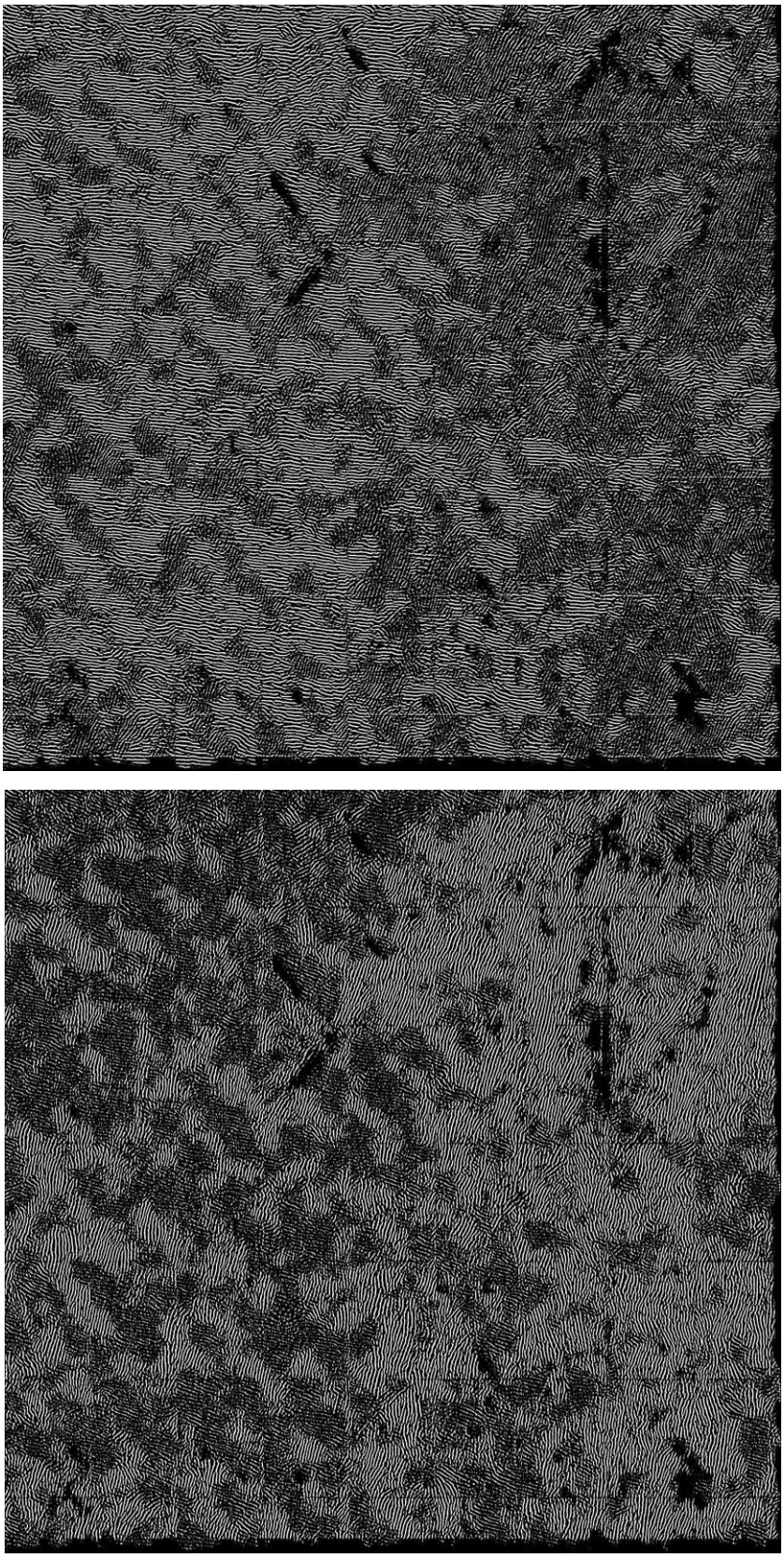


Figura 6-22. Imagen Greco_rombos después del filtrado de Sobel horizontal (arriba) y vertical (abajo).

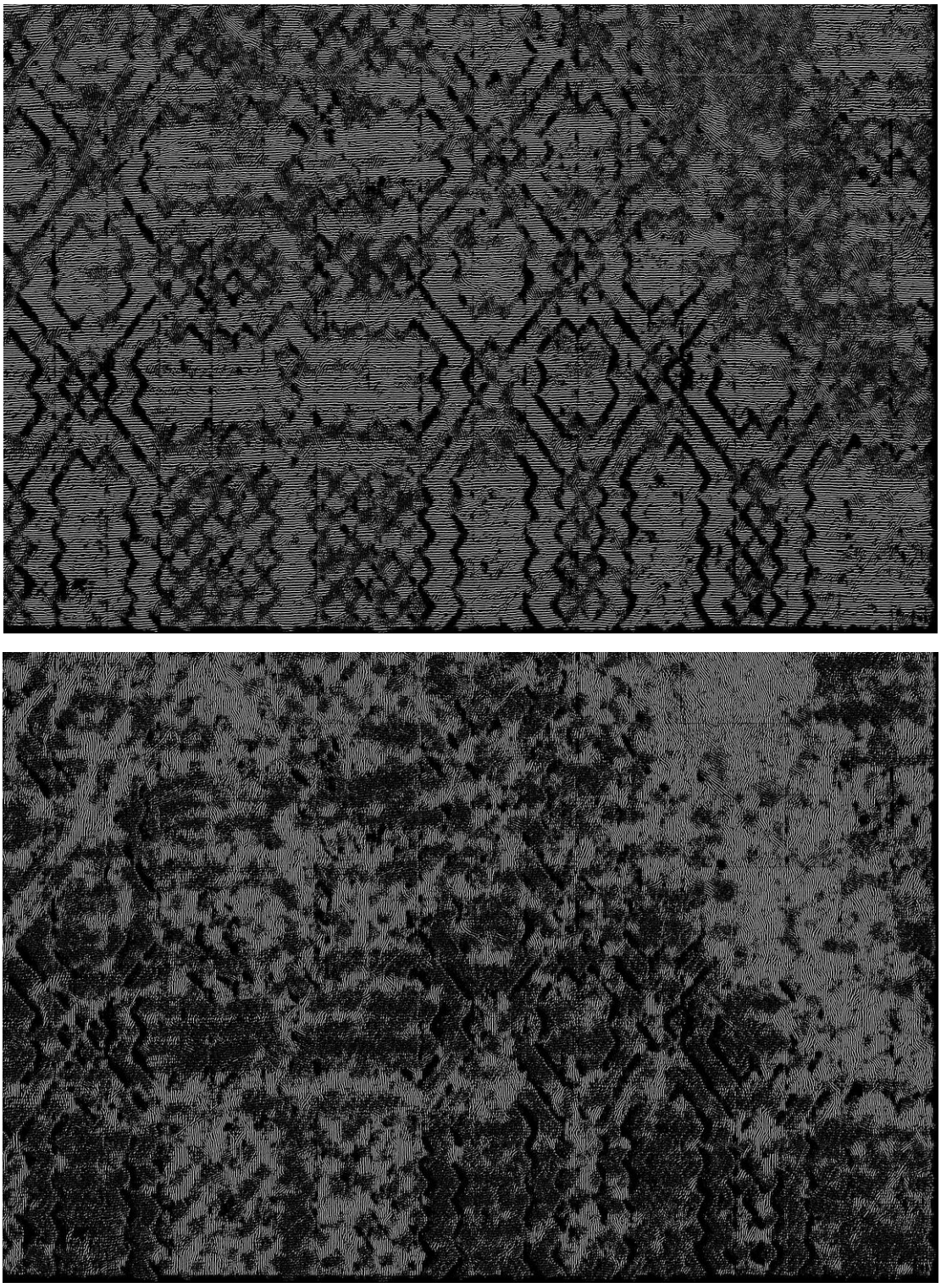


Figura 6-23. Imagen Murillo2 después del filtrado de Sobel horizontal (arriba) y vertical (abajo).

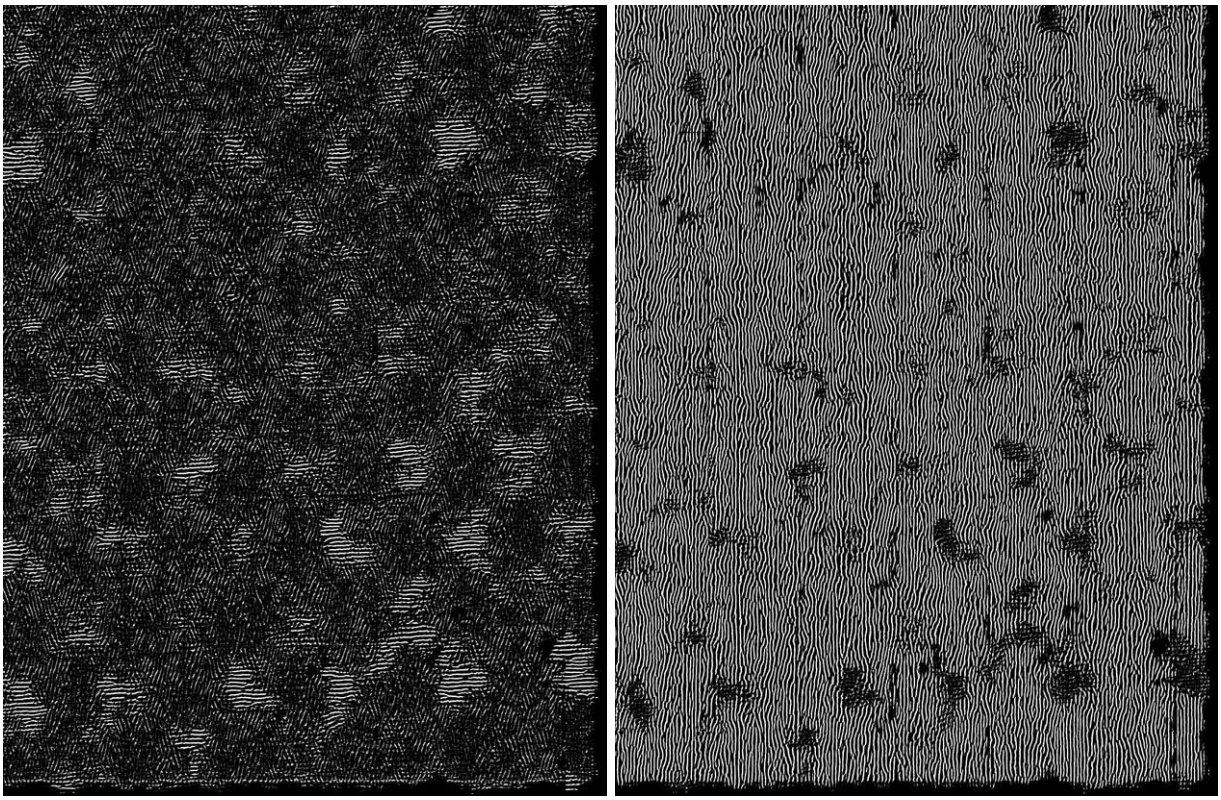


Figura 6-24. Imagen zurbaran1 después del filtrado de Sobel horizontal (izquierda) y vertical (derecha).

En vista de estos resultados podemos concluir que:

- Parece que el uso de filtros de Sobel da un resultado más convincente a simple vista, pues al eliminar una de las componentes y realzar la otra se ofrece una imagen más característica del patrón general que siguen los hilos del cuadro. Sin embargo, es posible que para ciertas aplicaciones, como es el caso del algoritmo de extracción de patrones para el que estaba destinado, sea más útil la operación de open, que da como resultado una mayor separación entre lo que podríamos caracterizar como figuras romboidales de hilos.

  Se probó a filtrar la imagen y luego hacer un opening, pero el resultado destruía el patrón por completo.

- Comparando la aplicación del filtrado en ambas direcciones, parece que el **mejor resultado se da al realzar sus componentes horizontales**. Mientras que en el algoritmo original se realiza todo un procedimiento para decidir qué direcciones realzar [1], el propuesto en este trabajo está limitado a una única de éstas. Si bien en términos de modularidad y escalado del problema esto puede ser considerado una ventaja, supone una limitación si se utiliza con cuadros cuyos patrones se deban encontrar mediante el análisis de sus hilos verticales.

  El origen de esto se encuentra en el propio funcionamiento de la red, pues no se debe olvidar estaba diseñada para el realce de huellas dactilares. Al entregarle nuestras imágenes, parte de los hilos verticales no son interpretados como surcos y son consecuentemente eliminados en el realce.

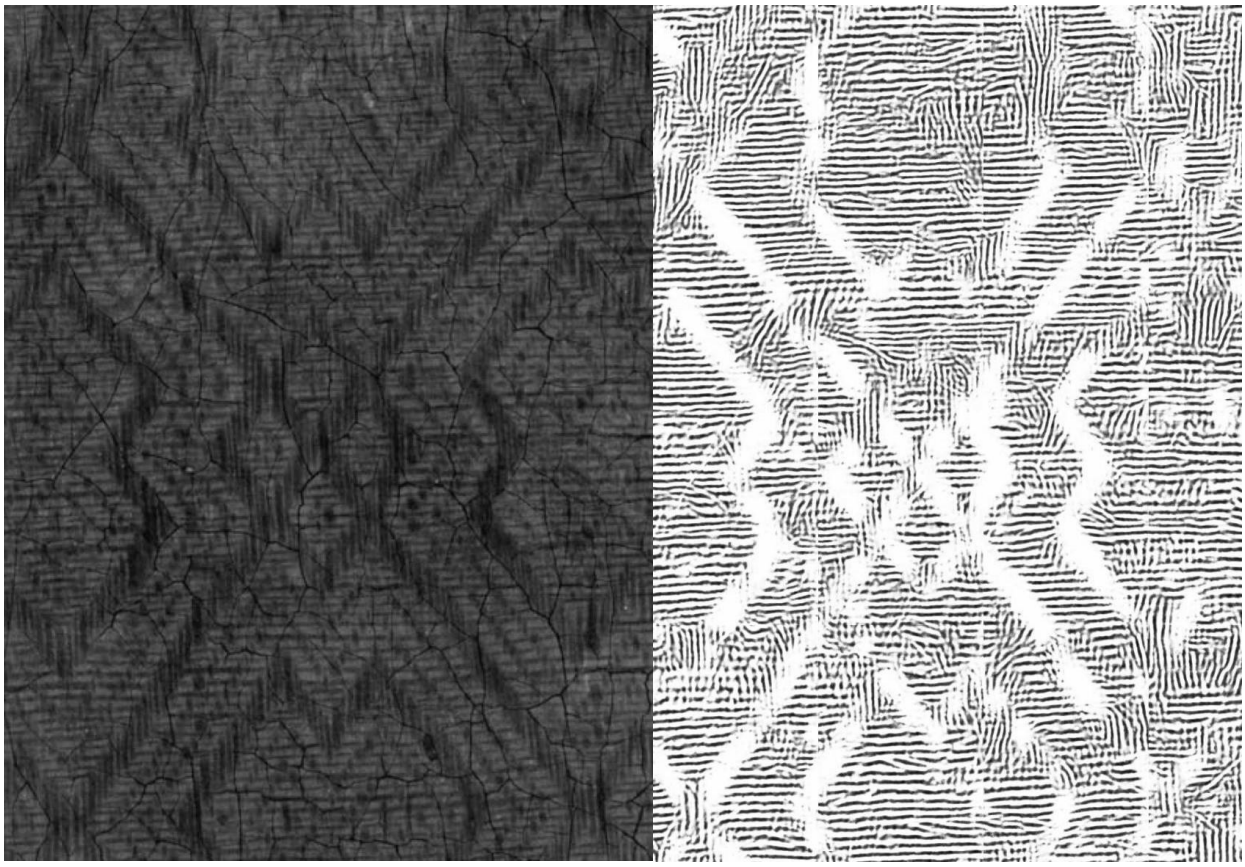


Figura 6-25. Problema del realce de hilos verticales en la imagen Murillo2.

6.2.3 Resultados

Llegados a este punto solo faltaría sustituir las imágenes resultantes en el algoritmo de [1]. De este modo el algoritmo tomaría estas nuevas imágenes a la hora de calcular las direcciones principales, y sucesivamente hasta proporcionar toda la información referente al patrón del cuadro.

Los resultados obtenidos tras la fase de identificación de direcciones principales fue el siguiente:

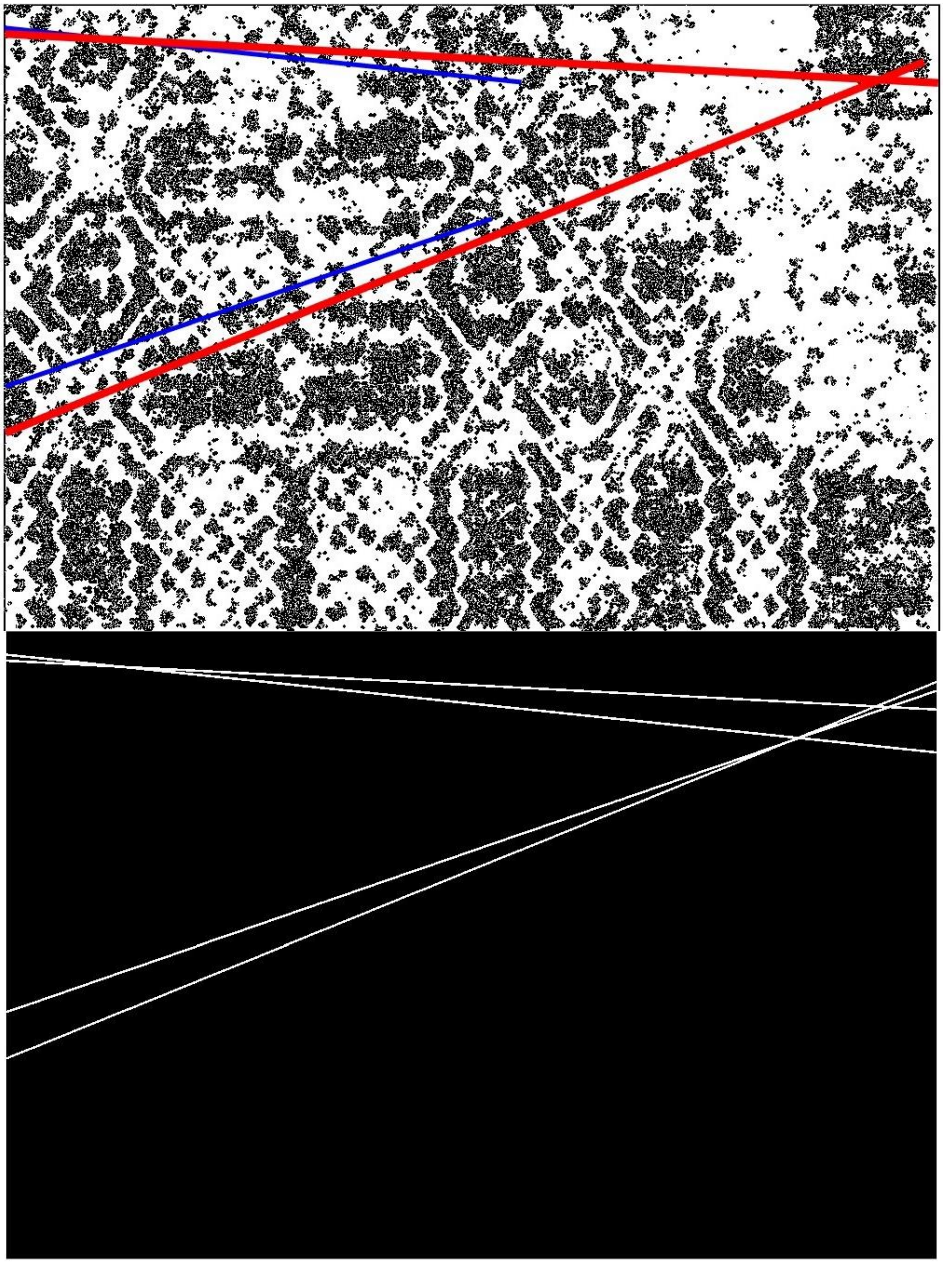


Figura 6-26. Imagen Murillo2: arriba, líneas encontradas con información relevante. Abajo: Máscara formada a partir de éstas.

Tal y como podía preverse en los apartados anteriores, comparando el resultado aportado por el nuevo algoritmo con los de la Figura 4-2, este resulta insuficiente, de modo que la etapa del algoritmo de [1] basado en la transformada de Hough no es capaz de calcular las direcciones principales del cuadro, haciendo imposible la ejecución de los pasos posteriores.

7 CONCLUSIONES Y LÍNEAS FUTURAS

El objetivo planteado en este trabajo era el de probar el funcionamiento de una red neuronal para mejorar la técnica de realce de hilos verticales y horizontales de un algoritmo existente. Desgraciadamente, tal y como se explica en el apartado 6.2.3, el tratamiento que propone este trabajo es incompatible con el algoritmo que se pretendía mejorar, dando no solo un peor resultado que el del original sino totalmente nulo.

Sin embargo, el procedimiento presentado produce en gran medida una imagen del patrón de los hilos del cuadro, especialmente cuando éstos están distribuidos en su mayoría en direcciones horizontales. Esto plantea la semilla de lo que podría ser la solución a una de las mayores debilidades del algoritmo de [1], que es su limitación a la hora de poder ser modularizado en otros algoritmos o escalado a bases de datos de más cuadros o incluso de otros museos, pues este requiere fuertemente de la interacción de un profesional que decida qué clase de procesado mejora el resultado en su primera fase.

También cabe destacar la que ha sido la mayor adversidad a la hora de realizar este trabajo, y que sin duda ha tenido repercusión en todo el desarrollo de éste que ha sido la escasez de códigos de redes neuronales de uso libre en el campo de las huellas dactilares. Como se comenta en el apartado 5.2, aunque la literatura de aplicaciones de DL en este ámbito es muy extensa, se tuvo que utilizar prácticamente la única red disponible de realce de huellas dactilares.

7.1 Líneas futuras

Puesto que ningún trabajo de investigación supone nunca un punto y final, a continuación se proponen varias ideas que puedan dar origen a trabajos futuros:

- Si bien el trabajo no ha tenido el resultado al que se aspiraba en un primer momento, todo el código desarrollado para el parcheado y unión de parches de imágenes puede ser empleado en cualquier contexto en el que se esté trabajando con redes neuronales convolucionales. Puesto que éstas suelen tener entradas y salidas de dimensiones fijas en función de la base de datos que empleen para su entrenamiento y para la aplicación para la que estén destinadas, será necesario algún tipo de pre-procesamiento en caso de querer trabajar con imágenes que no cumplan con estas dimensiones.
- Podría estudiarse la utilización de otras redes con el mismo objetivo de este trabajo, más concretamente de aquellas que calculen el OF de una imagen de huella dactilar. En los apartados 5.2.1 y 5.2.2 de este trabajo se presentan FingerNet y OrieNet.

  Para ello sería necesaria la realización de un nuevo algoritmo que calculara los ángulos de los vectores del OF, los cuales aportarían una descripción mucho más exacta de las direcciones principales de los hilos del cuadro.
- Por último, tomando la red FPD-M-Net utilizada en este trabajo, podría aplicarse un proceso de Transfer Learning para conseguir que ésta funcione correctamente al recibir imágenes de rayos X de cuadros y así mejorar su bajo rendimiento tanto en los bordes como con los hilos verticales. Esto, sin embargo, podría no ser posible en función de la cantidad de recursos de este tipo de imágenes, pues su uso debería ser solicitado a aquellas instituciones dueñas de éstas.

REFERENCIAS

[1] M. d. M. Velasco Montero, Extracción de patrones en telas de cuadros del siglo, Sevilla: Universidad de Sevilla, 2016.

[2] A. G. Serrano, INTELIGENCIA ARTIFICIAL. Fundamentos, práctica y aplicaciones, RC Libros, 2016.

[3] C. A. Khanzode y R. D. Sarode, «ADVANTAGES AND DISADVANTAGES OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING: A LITERATURE REVIEW,» *International Journal of Library & Information Science (IJLIS)*, vol. 9, 2020.

[4] J. Torres, Python Deep Learning, Introducción práctica con Keras y TensorFlow 2, MARCOMBO, 2020.

[5] P. Norvig y S. Russel, Artificial Intelligence: A Modern Approach, Pearson, 2021.

[6] M. E. Celebi, Supervised and Unsupervised Learning for Data Science, Springer, 2020.

[7] R. Raj, «Supervised, Unsupervised, And Semi-Supervised Learning With Real-Life Usecase,» [En línea]. Available: https://www.enjoyalgorithms.com/blogs/supervised-unsupervised-and-semisupervised-learning.

[8] F. Berzal, Redes neuronales and deep learning, Granada, 2018.

[9] U. García, «Introducción a las redes neuronales Pt. I,» 2019. [En línea]. Available: https://futurelab.mx/redes%20neuronales/inteligencia%20artificial/2019/06/25/intro-a-redes-neuronales-pt-1/.

[10] Z.-H. Zhou, Neural Networks, Springer, 2021.

[11] C. Xiao y J. Sun, Introducction to Deep Learning for Healthcare, Springer, 2021.

[12] J. Feng, «Commonly used activation functions,» 2019. [En línea]. Available: https://www.researchgate.net/figure/Commonly-used-activation-functions-a-Sigmoid-b-Tanh-c-ReLU-and-d-LReLU_fig3_335845675.

[13] S. Saxena, «Introduction to Softmax for Neural Network,» 2021. [En línea]. Available: https://www.analyticsvidhya.com/blog/2021/04/introduction-to-softmax-for-neural-network/.

[14] I. Goodfellow, Y. Bengio y A. Courville, Deep Learning (Adaptive Computation and Machine Learning series), vol. 2, 2016.

[15] S. Khan, H. Rahmani, S. A. Ali Shah y M. Bennamoun, A Guide to Convolutional Neural Networks for Computer Vision, Springer, 2018.

[16] H. Kumar, «Autoencoder: Downsampling and Upsampling,» [En línea]. Available: https://kharshit.github.io/blog/2019/02/15/autoencoder-downsampling-and-upsampling.

[17] Tashmit, «Convolution layer, Padding, Stride, and Pooling in CNN,» [En línea]. Available: https://www.codingninjas.com/codestudio/library/convolution-layer-padding-stride-and-pooling-in-cnn.

[18] K. Patel, «Convolutional Neural Networks — A Beginner's Guide,» 09 2019. [En línea]. Available: https://towardsdatascience.com/convolution-neural-networks-a-beginners-guide-implementing-a-mnist-hand-written-digit-8aa60330d022.

[19] P. Kovesi, «MATLAB and Octave Functions for Computer Vision and Image Processing.,» [En línea]. Available: https://www.peterkovesi.com/matlabfns/#fingerprints.

[20] Hong, Lin, Y. Wan y A. Jain, «Fingerprint Image Enhancement: Algorithm and Performance Evaluation,» *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 20, 1998.

[21] P. Lara, «Apple presenta el iPhone 5s - el teléfono inteligente más avanzado del mundo,» [En línea]. Available: https://www.apple.com/es/newsroom/2013/09/10Apple-Announces-iPhone-5s-The-Most-Forward-Thinking-Smartphone-in-the-World/.

[22] D. Maltoni, D. Maio, A. Jain y S. Prabhakar, Handbook of Fingerprint Recognition, Springer, 2009.

[23] A. Jain, S. Prabhakar y S. Pankanti, «On the similarity of identical twin Fingerprints,» *Pattern Recognition 35 (2002) 2653 – 2663,* 2001.

[24] P. İrtem, DEEP LEARNING IN FINGERPRINT ANALYSIS, İZMİR, 2020.

[25] A. Jain y S. Pankanti, The Essential Guide to Image Processing, 2009.

[26] Z. Yang, Y. Xu y G. Lu, «Efficient Method for High-Resolution Fingerprint Image Enhancement Using Deep Residual Network,» *2020 IEEE Symposium Series on Computational Intelligence (SSCI),* 2020.

[27] L. Chen, H. Yang Yin, T. Wang, H. Xu y M. Song Tong, «An Improved Algorithm for Enhancing Fingerprint Image Quality,» *2018 Progress In Electromagnetics Research Symposium (PIERS — Toyama),* 2018.

[28] N. Yager y A. Amin, «Fingerprint verification based on minutiae features: a review,» *Pattern Anal Applic (2004) 7: 94–113,* 2003.

[29] W. Bian, X. Deqin, Y. Cheng, B. Jie y X. Ding, «A Survey of the Methods on Fingerprint Orientation Field Estimation,» *IEEE Access,* vol. 7, 2019.

[30] B. Dorizzi, R. Cappelli, M. Ferrara, D. Maio, D. Maltoni, N. Houmani, S. Garcia-Salicetti y A. Mayoue, «Fingerprint and On-Line Signature Verification Competitions at ICB 2009,» *Proceedings of the International Conference on Biometrics (ICB)*, pp. 725-732, 2009.

[31] LivDet, «LivDet: 2023,» [En línea]. Available: https://livdet.diee.unica.it/index.php.

[32] Y. Tang, F. Gao, J. Feng y Y. Liu, «FingerNet: An Unified Deep Network for Fingerprint Minutiae Extraction,» *IEEE International Joint Conference on Biometrics (IJCB)*, 2017.

[33] Z. Qu, J. Liu, Y. Liu, Q. Guan, C. Yang y Y. Zhang, «OrieNet: A Regression System for Latent Fingerprint Orientation Field Extraction,» *Artificial Neural Networks and Machine Learning – ICANN 2018*, 2018.

[34] S. Adiga y J. Sivaswamy, «FPD-M-net: Fingerprint Image Denoising and Inpainting Using M-net Based Convolutional Neural Networks,» de *Impainting and Denoising Challenges, The Springer Series on Challenges in Machine Learning,*, Hyderabad, India, Springer, 2019.

[35] R. Mehta y J. Sivaswamy, «M-NET: A CONVOLUTIONAL NEURAL NETWORK FOR DEEP BRAIN STRUCTURE SEGMENTATION,» *IEEE*, 2017.

[36] X. Ying, «An Overview of Overfitting and its Solutions,» *IOP Conf. Series: Journal of Physics*, 2019.

[37] Y. Fernández, «Qué es Github y qué es lo que le ofrece a los desarrolladores,» Xataka Basics, [En línea]. Available: https://www.xataka.com/basics/que-github-que-que-le-ofrece-a-desarrolladores.

[38] «Jupyter.org,» [En línea]. Available: https://jupyter.org/.

[39] S. Escalera, «Fingerprint inpainting and denoising (WCCI'18, ECCV'18),» ChaLearn, 2018. [En línea]. Available: https://chalearnlap.cvc.uab.cat/dataset/32/description/#.

[40] Rvziel, «Filtros para la deteccion de bordes de una imagen con Python 3 (Parte 2),» Python's Eyes, 20 junio 2017. [En línea]. Available: https://pythoneyes.wordpress.com/2017/07/28/filtros-para-la-deteccion-de-bordes-de-una-imagen-con-python-3-parte-2/.

ANEXO A

Código del cuaderno de Google Colab, empleado para cargar las imágenes de los cuadros, el código Python desarrollado en este trabajo para su pre y pos-procesado y la ejecución de la red neuronal.

[1]
```
!git clone https://github.com/adigasu/FDPMNet.git
```

[2]
```
#Importamos el modulo del sistema operativo (os):
import os
```

[3]
```
os.chdir("FDPMNet/")
```

[4]
```
#Borramos los archivos obsoletos.
if (os.path.isfile("test.py")):
  print("Borramos archivo test.py")
  os.remove("test.py")
else:
  print("Archivo test.py no existe")
```

[5]
```
#Borramos carpeta de Results (para comprobar que se está ejecutando bien el
código de la red)
os.chdir("test/")

import shutil
if (os.path.exists("Results")):
  print("Borramos carpeta Results")
  shutil.rmtree("Results")
else:
  print("Carpeta Results no existe")

#Volvemos a la carpeta FDPMNet:
os.chdir("../")
```

[6]
```
#Montamos el Drive para la carga de las imágenes y los archivos
actualizados:
from google.colab import drive
drive.mount("/content/gdrive/", force_remount=True)
```

[7]
```
#Movemos el archivo test.py actualizado:
origen_arch  = "../gdrive/MyDrive/TFG/test.py"
destino_arch = "../FDPMNet"
shutil.copy(origen_arch, destino_arch)
```

[8]	#Copiamos todas las imágenes en la carpeta para que puedan ejecutarse.
	#Se crea la carpeta: ImagenesTFG
	os.chdir("test/")
	if os.path.exists("ImagenesTFG"):
	  print("Carpeta ya existe")
	else:
	  os.mkdir("ImagenesTFG")
	os.chdir("../")
	
	origen_arch = "../gdrive/MyDrive/TFG/Imagenes"
	destino_arch = "../FDPMNet/test/ImagenesTFG"
	
	files = os.listdir(origen_arch)
	for fname in files:
	  shutil.copy(os.path.join(origen_arch,fname), destino_arch)

[9]	#Ejecutamos la red FPD-M-Net !python test.py './test'

ANEXO B

Código del archivo Python test.py tras ser actualizado con las funciones necesarias para cumplir con los objetivos de este trabajo.

```
from __future__ import print_function
from typing import Dict
#@inproceedings{adiga2018fpdmnet,
#  title={FPD-M-net: Fingerprint Image Denoising and Inpainting Using M-Net
Based Convolutional Neural Networks},
#  author={Adiga, Sukesh V and Sivaswamy, Jayanthi},
#  booktitle={arXiv preprint arXiv:1812.10191},
#  year={2018},
#}

"Fingerprint Image denoising using M-net"
#Código usado en el T.F.G de Manuel Manzano (@manmanher, US, 2022) para el
realce de las imágenes de cuadros de Museo del Prado
#usando la red neuronal convolucional FPD-M-net de Sukesh Adiga y Jayanthi
Sivaswamy.

import os
import glob
import numpy as np
import PIL
from PIL import Image
import datetime
import scipy as sp
import imageio
import sys
import keras as keras
from keras.models import Model
from keras.layers import Input, Activation, Conv2D, MaxPooling2D, UpSampling2D,
Dropout
# from keras.layers.merge import concatenate
from keras.layers import concatenate
#from keras.layers.normalization import BatchNormalization
from keras.layers import BatchNormalization
from keras import backend as K
from tensorflow.keras.optimizers import SGD
```

```
import math
import cv2
# image_data_format = channels_last
K.set_image_data_format('channels_last')

#Necesario para que no piense que estamos haciendo un Decompression bomb DOS
attack:
#Establecemos el tamaño maximo de pixeles en el mayor de todas nuestras
imagenes:
PIL.Image.MAX_IMAGE_PIXELS = 248500000

# Parameter
################################################################
modelName = 'FPD_M_net_weights'
lrate = 0.1
decay_Rate = 1e-6
height = 400
width = 275
padSz = 88
d = 8
padX = (d - height%d)
padY = (d - width%d)
ipDepth = 3
ipHeight = height + padX + padSz
ipWidth = width + padY + padSz
outDepth = 1
is_zeroMean = False
################################################################

# Loss function
def my_loss(y_true, y_pred):
    l1_loss = K.mean(K.abs(y_pred - y_true))
    return l1_loss


# Define the neural network
def getFPDMNet(patchHeight, patchWidth, ipCh, outCh):

    # Input
    input1 = Input((patchHeight, patchWidth, ipCh))

    # Encoder
    conv1 = Conv2D(16, (3, 3), padding='same')(input1)
    conv1 = BatchNormalization()(conv1)
    conv1 = Activation('relu')(conv1)
    conv1 = Dropout(0.2)(conv1)

    conv1 = concatenate([input1, conv1], axis=-1)
```

```
    conv1 = Conv2D(16, (3, 3), padding='same')(conv1)
    conv1 = BatchNormalization()(conv1)
    conv1 = Activation('relu')(conv1)
    pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)

    #
    input2 = MaxPooling2D(pool_size=(2, 2))(input1)
    conv21 = concatenate([input2, pool1], axis=-1)

    conv2 = Conv2D(32, (3, 3), padding='same')(conv21)
    conv2 = BatchNormalization()(conv2)
    conv2 = Activation('relu')(conv2)
    conv2 = Dropout(0.2)(conv2)

    conv2 = concatenate([conv21, conv2], axis=-1)
    conv2 = Conv2D(32, (3, 3), padding='same')(conv2)
    conv2 = BatchNormalization()(conv2)
    conv2 = Activation('relu')(conv2)
    pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)

    #
    input3 = MaxPooling2D(pool_size=(2, 2))(input2)
    conv31 = concatenate([input3, pool2], axis=-1)

    conv3 = Conv2D(64, (3, 3), padding='same')(conv31)
    conv3 = BatchNormalization()(conv3)
    conv3 = Activation('relu')(conv3)
    conv3 = Dropout(0.2)(conv3)

    conv3 = concatenate([conv31, conv3], axis=-1)
    conv3 = Conv2D(64, (3, 3), padding='same')(conv3)
    conv3 = BatchNormalization()(conv3)
    conv3 = Activation('relu')(conv3)
    pool3 = MaxPooling2D(pool_size=(2, 2))(conv3)

    #
    input4 = MaxPooling2D(pool_size=(2, 2))(input3)
    conv41 = concatenate([input4, pool3], axis=-1)

    conv4 = Conv2D(128, (3, 3), padding='same')(conv41)
    conv4 = BatchNormalization()(conv4)
    conv4 = Activation('relu')(conv4)
    conv4 = Dropout(0.2)(conv4)

    conv4 = concatenate([conv41, conv4], axis=-1)
    conv4 = Conv2D(128, (3, 3), padding='same')(conv4)
    conv4 = BatchNormalization()(conv4)
    conv4 = Activation('relu')(conv4)
    conv4 = Dropout(0.2)(conv4)
```

```
conv4 = Conv2D(128, (3, 3), padding='same')(conv4)
conv4 = BatchNormalization()(conv4)
conv4 = Activation('relu')(conv4)

# Decoder
conv5 = UpSampling2D(size=(2, 2))(conv4)
conv51 = concatenate([conv3, conv5], axis=-1)

conv5 = Conv2D(64, (3, 3), padding='same')(conv51)
conv5 = BatchNormalization()(conv5)
conv5 = Activation('relu')(conv5)
conv5 = Dropout(0.2)(conv5)

conv5 = concatenate([conv51, conv5], axis=-1)
conv5 = Conv2D(64, (3, 3), padding='same')(conv5)
conv5 = BatchNormalization()(conv5)
conv5 = Activation('relu')(conv5)

#
conv6 = UpSampling2D(size=(2, 2))(conv5)
conv61 = concatenate([conv2, conv6], axis=-1)

conv6 = Conv2D(32, (3, 3), padding='same')(conv61)
conv6 = BatchNormalization()(conv6)
conv6 = Activation('relu')(conv6)
conv6 = Dropout(0.2)(conv6)

conv6 = concatenate([conv61, conv6], axis=-1)
conv6 = Conv2D(32, (3, 3), padding='same')(conv6)
conv6 = BatchNormalization()(conv6)
conv6 = Activation('relu')(conv6)

#
conv7 = UpSampling2D(size=(2, 2))(conv6)
conv71 = concatenate([conv1, conv7], axis=-1)

conv7 = Conv2D(16, (3, 3), padding='same')(conv71)
conv7 = BatchNormalization()(conv7)
conv7 = Activation('relu')(conv7)
conv7 = Dropout(0.2)(conv7)

conv7 = concatenate([conv71, conv7], axis=-1)
conv7 = Conv2D(16, (3, 3), padding='same')(conv7)
conv7 = BatchNormalization()(conv7)
conv7 = Activation('relu')(conv7)

# Final
```

```
    conv81 = UpSampling2D(size=(8, 8))(conv4)
    conv82 = UpSampling2D(size=(4, 4))(conv5)
    conv83 = UpSampling2D(size=(2, 2))(conv6)
    conv8 = concatenate([conv81, conv82, conv83, conv7], axis=-1)
    conv8 = Conv2D(outCh, (1, 1), activation='sigmoid')(conv8)

    ############
    model = Model(inputs=input1, outputs=conv8)

    sgd = SGD(lr=lrate, decay=decay_Rate, momentum=0.75, nesterov=True)
    model.compile(optimizer=sgd, loss=my_loss)

    return model

######################## Load & Preprocess Data ########################

# pad Data
# Editada por @manmanher: si la imagen a paddear, despues del pad normal sigue
sin ser del tamaño que admite la red (408x280 al pasar a multiplo de 8),
# se fuerza relleno hasta cumplir esos valores, esto generara un borde blanco a
la derecha y abajo del todo en la imagen resultante.
def padData(img):
    # For grayscale image
    if len(img.shape) == 3:
        H, W, nB = img.shape
        img = np.transpose(img, (2,1,0))
        img = np.reshape(img, (nB, W, H, 1))

    # Pad X, Y such that it will be divisible by 8
    nBi, Hi, Wi, Di = img.shape
    temp = np.zeros((nBi, Hi + padX, Wi + padY, Di))
    temp[:, :Hi, :Wi, :] = img
    temp2 = temp
    nB, H, W, D = temp2.shape

    #Prueba: pad extra si es el ultimo parche y hacen falta pixeles para ser
408x280
    if H < 408 and W < 280:
      temp2 = np.zeros((nB, 408, 280, D))

    elif H < 408:
      temp2 = np.zeros((nB, 408, W, D))

    elif W < 280:
      temp2 = np.zeros((nB, H, 280, D))

    temp2[:, :Hi, :Wi, :] = img
    nB, H, W, D = temp2.shape
    # Pad extra for network
```

```
    img = np.zeros((nB, H + padSz, W + padSz, D))
    for i in range(0, nB):
        for j in range(0, D):
            img[i,:,:,j] = np.lib.pad(temp2[i,:,:,j], (int(padSz/2)), 'edge')

    img = (img.astype('float32'))
    return img

#Editada por @manmanher: si la imagen a cargar es grande, no se añade padding:
se añadirá después a sus parches.
def load_data(dataPath):
    imgsX = imread(dataPath)
    isalargeimg = False
    #print(imgsX.shape)
    if imgsX.shape[1] > 400 or imgsX.shape[2] > 275:
        isalargeimg = True

    if is_zeroMean:
        imgsX = np.array(imgsX)/127.5 - 1.
    else:
        imgsX = np.array(imgsX)/255.0

    # Pad images
    if not isalargeimg:
        imgsX = padData(imgsX)
    return imgsX

def imread(path, is_gt=False):
    temp = imageio.imread(path).astype(float)
    if is_gt:
        temp = np.expand_dims(temp, -1)
    temp = np.expand_dims(temp, 0)
    return temp

######################## Funciones añadidas por @manmanher
########################

#Creamos una funcion que convierta los archivos a formato .jpg (es el que acepta
la red):
def ConvertToJPG(dataPath):
    origen_arch = os.path.join(dataPath, "ImagenesTFG")
    files = os.listdir(origen_arch)
    for fname in files:
        file_name     = fname.split('.')[0]
        file_ext      = fname.split('.')[1]

        new_file_name = file_name + '.jpg'
        new_destino   = os.path.join(dataPath, new_file_name)
```

```
        ubi_img = os.path.join(origen_arch, fname)
        img = Image.open(ubi_img)

        if file_ext == "tif":
            img = cv2.imread(ubi_img)
            cv2.imwrite(new_destino, img, [int(cv2.IMWRITE_JPEG_QUALITY), 100,
int(cv2.IMWRITE_PNG_COMPRESSION),0])
        else:
            rgb_im = img.convert('RGB')
            rgb_im.save(new_destino, compress_level=0)

#Funcion que parchea una imagen grande en parches de dimension H x W,
#con un solape de solap_x píxeles columnas y solap_y píxeles entre filas:
def parchear(H, W, imgDataFile, dataPath, solap_x, solap_y):
    imTest = imgDataFile
    #Creamos carpeta donde se almacenaran todos los parches.
    file_name     = os.path.basename(imgDataFile)
    file_name     = file_name.split('.')[0]
    savePath_temp = os.path.join(dataPath, file_name)

    if not os.path.exists(savePath_temp):
        os.makedirs(savePath_temp)

    imTest = cv2.imread(imgDataFile, 0) #añadir arg 0 para obtener componente de
gris solo
    it_i = 0
    it_j = 0
    head_i = '_'
    head_j = '_'
    numParches = 0
    ultimo_fin_y = 0
    firsttime = True

    while ultimo_fin_y <= imTest.shape[0]:
        it_j  = 0
        decenas_i = it_i/10
        decenas_i_f = int(math.floor(decenas_i))
        for aux in range(0,decenas_i_f):
            head_i = head_i + '_'

        ultimo_fin_x = 0

        while ultimo_fin_x <= imTest.shape[1]:
            if it_i == 0 and it_j == 0:
                parche = imTest[0:H, 0:W]
                ultimo_fin_x = (it_j+1)*W

            elif it_i == 0 and it_j != 0:
```

```
                parche = imTest[0:H, (ultimo_fin_x - solap_x):(ultimo_fin_x -
solap_x + W)]
                ultimo_fin_x = ultimo_fin_x - solap_x + W

            elif it_j == 0 and it_i != 0:
                if firsttime:
                    ultimo_fin_y += solap_y
                    firsttime = False
                parche = imTest[(ultimo_fin_y - solap_y):(ultimo_fin_y - solap_y
+ H), 0:W]
                ultimo_fin_x = W
            else:
                parche = imTest[(ultimo_fin_y - solap_y):(ultimo_fin_y - solap_y
+ H), (ultimo_fin_x - solap_x):(ultimo_fin_x - solap_x + W)]
                ultimo_fin_x = ultimo_fin_x - solap_x + W

            decenas_j = it_j/10
            decenas_j_f = int(math.floor(decenas_j))

            for aux in range(0,decenas_j_f):
                head_j = head_j + '_'
            file_name     = os.path.basename(imgDataFile)
            file_name     = file_name.split('.')[0]+ head_i + str(it_i) + head_j
+ str(it_j) + '.jpg'
            Image.fromarray(np.uint8(parche)).save(os.path.join(savePath_temp,
file_name), compress_level=0)
            it_j += 1
            head_j = '_'
            numParches += 1

        ultimo_fin_y = ultimo_fin_y - solap_y + H
        it_i += 1
        head_i = '_'

#Funcion que une todas las imagenes de una carpeta y coloca el resultado en la
carpeta Results:
#H, W: altura y ancho de las imagenes que se unen, respectivamente.
#numFilas, numCols: numero de parches en cada fila y columna, respectivamente.
def unirParches(imTest, savePath_temp, imgDataFile, H, W, numFilas, numCols):
    itt  = 0
    salida = np.zeros((H*numFilas, W*numCols))
    files = os.listdir(savePath_temp)
    files.sort()

    while itt < len(files):
      for fil in range(0,numFilas):
        for col in range(0,numCols):
          fname= files[itt]
```

```
            sp_temp = os.path.join(savePath_temp, fname)
            img = imageio.imread(sp_temp)
            salida[(fil*H):((fil+1)*H), (col*W):((col+1)*W)] = img
            itt+=1

    dir_uni_parches = os.path.join(savePath_temp, os.path.basename(imgDataFile))
    Image.fromarray(np.uint8(salida)).save(dir_uni_parches, compress_level=0)

    # Aplicaremos filtro de Sobel para resaltar las componentes horizontales y
verticales
    #de la imagen realzada final.
    #Primero horizontal (dx=0)
    edge_sobel = cv2.Sobel(src=salida, ddepth=-1, dx=0, dy=1, ksize=3)
    file_name     = os.path.basename(imgDataFile)
    file_name     = file_name.split('.')[0]+ '_Sobel_Hor' + '.jpg'
    dir_uni_parches = os.path.join(savePath_temp, file_name)
    Image.fromarray(np.uint8(edge_sobel)).save(dir_uni_parches,
compress_level=0)

    #Ahora en vertical (dy=0)
    edge_sobel = cv2.Sobel(src=salida, ddepth=-1, dx=1, dy=0, ksize=3)
    file_name     = os.path.basename(imgDataFile)
    file_name     = file_name.split('.')[0]+ '_Sobel_Vert' + '.jpg'
    dir_uni_parches = os.path.join(savePath_temp, file_name)
    Image.fromarray(np.uint8(edge_sobel)).save(dir_uni_parches,
compress_level=0)

##############################################################################
#####

def test_FPDMNet(dataPath):
    #PRIMERO: Convertimos todas las imagenes añadidas por nosotros a formato
.jpg:
    ConvertToJPG(dataPath)
    H = 400
    W = 275
    solap_x = 50  #Num de cols  repetidas entre parches
    solap_y = 92  #Num de filas repetidas entre parches

    # Get FPD_M_Net
    loadWeightsPath = os.path.join("./weights", modelName + ".hdf5")
    fpDenNet = getFPDMNet(ipHeight, ipWidth, ipDepth, outDepth)

    # load weights
    fpDenNet.load_weights(loadWeightsPath)

    imgDataFiles     = glob.glob(os.path.join(dataPath, '*.jpg'))
    imgDataFiles.sort()
```

```
    savePath = os.path.join(dataPath, "Results")
    if not os.path.exists(savePath):
        os.makedirs(savePath)

    for i in range(0, len(imgDataFiles)):

        print('----------------------------> Test: ', i, ' <------------------
----------')
        print ("                    Se procesará la imagen :",
os.path.basename(imgDataFiles[i]))

        # Read fundus data from imgDataFiles
        imTest = load_data(imgDataFiles[i])
        print("Imagen cargada: ", imTest.shape)

        if imTest.shape[1] > 496 or imTest.shape[2] > 368:
            #Creamos directorio donde iran los parches, dentro de /Results:
            file_name     = os.path.basename(imgDataFiles[i])
            file_name     = file_name.split('.')[0]
            savePath_temp = os.path.join(savePath, file_name)
            print(savePath_temp)
            if not os.path.exists(savePath_temp):
              os.makedirs(savePath_temp)

            parchear(H, W, imgDataFiles[i], dataPath, solap_x, solap_y)

            #Creamos una carpeta por cada imagen grande donde se guardaran sus
parches:
            file_name     = os.path.basename(imgDataFiles[i])
            file_name     = file_name.split('.')[0]
            savePath_temp = os.path.join(savePath, file_name)
            if not os.path.exists(savePath_temp):
                os.makedirs(savePath_temp)

            it_i = 0
            it_j = 0
            head_i = '_'
            head_j = '_'
            numParches=0
            ultimo_fin_y = 0
            firsttime = True

            while ultimo_fin_y <= imTest.shape[1]:
                it_j  = 0
                decenas_i = it_i/10
                decenas_i_f = int(math.floor(decenas_i))

                for aux in range(0,decenas_i_f):
```

```
                    head_i = head_i + '_'

                ultimo_fin_x = 0

                while ultimo_fin_x <= imTest.shape[2]:
                    if it_i == 0 and it_j == 0:
                      parche = imTest[:, 0:H, 0:W,:]
                      ultimo_fin_x = (it_j+1)*W

                    elif it_i == 0 and it_j != 0:
                      parche = imTest[:, 0:H, (ultimo_fin_x -
solap_x):(ultimo_fin_x - solap_x + W),:]
                      ultimo_fin_x = ultimo_fin_x - solap_x + W

                    elif it_j == 0 and it_i != 0:
                      if firsttime:
                        ultimo_fin_y += solap_y
                        firsttime = False

                      parche = imTest[:, (ultimo_fin_y - solap_y):(ultimo_fin_y
- solap_y + H), 0:W,:]
                      ultimo_fin_x = W

                    else:
                      parche = imTest[:, (ultimo_fin_y - solap_y):(ultimo_fin_y
- solap_y + H), (ultimo_fin_x - solap_x):(ultimo_fin_x - solap_x + W),:]
                      ultimo_fin_x = ultimo_fin_x - solap_x + W

                    decenas_j = it_j/10
                    decenas_j_f = int(math.floor(decenas_j))
                    for aux in range(0,decenas_j_f):
                      head_j = head_j + '_'

                    #Hay que añadir a los parches de 400x275 el padding que usa
la red:
                    parche = padData(parche)
                    imPred = fpDenNet.predict(parche, batch_size=1)
                    imPred = np.squeeze(imPred)

                    #Eliminamos bordes en los que consideramos que la red no
tiene buen desempeño
                    #Para la primera actualizacion: no eliminamos bordes del
tamaño del solape al no haber solape:
                    if solap_x > 0 or solap_y > 0:
                        unpad_x = int(solap_x/2)
                        unpad_y = int(solap_y/2)
                        imPred = imPred[unpad_y:-unpad_y, unpad_x:-unpad_x]

                    unpad = int(padSz/2)
```

```
                    imPred = imPred[unpad:-unpad, unpad:-unpad]
                    imPred = imPred[:-padX, :-padY]
                    print('After unpad: ', imPred.shape)

                    hei,wid=imPred.shape
                    file_name     = os.path.basename(imgDataFiles[i])
                    file_name     = file_name.split('.')[0]+ head_i + str(it_i)
+ head_j + str(it_j) + '.jpg'
                    Image.fromarray(np.uint8(imPred *
255)).save(os.path.join(savePath_temp, file_name), compress_level=0)
                    it_j += 1
                    head_j = '_'
                    numParches += 1

                ultimo_fin_y = ultimo_fin_y - solap_y + H
                it_i += 1
                head_i = '_'

            unirParches(imTest, savePath_temp, imgDataFiles[i], hei, wid, it_i,
it_j)

        else:
            imPred = fpDenNet.predict(imTest, batch_size=1)
            imPred = np.squeeze(imPred)

            #Descarte de bordes: como prueba para ver la zona en la que la red
es efectiva
            #unpad_x = int(solap_x/2)
            #unpad_y = int(solap_y/2)
            #imPred = imPred[unpad_y:-unpad_y, unpad_x:-unpad_x]
            unpad = int(padSz/2)
            imPred = imPred[unpad:-unpad, unpad:-unpad]
            imPred = imPred[:-padX, :-padY]
            Image.fromarray(np.uint8(imPred * 255)).save(os.path.join(savePath,
os.path.basename(imgDataFiles[i])))

if __name__ == '__main__':
    dataPath = sys.argv[1]
    #dataPath = './test'
    test_FPDMNet(dataPath)
```