Trabajo Fin de Máster
Máster Universitario en Ingeniería de Telecomunicación

Auto-codificadores para detección y procesado de imágenes

Autor: Francisco Jesús Marchal Cebador
Tutor: Juan José Murillo Fuentes

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

Sevilla, 2019

Trabajo Fin de Máster
Máster Universitario en Ingeniería de Telecomunicación

Auto-codificadores para detección y procesado de imágenes

Autor:
Francisco Jesús Marchal Cebador

Tutor:
Juan José Murillo Fuentes
Prof. Catedrático de Universidad

Dpto. de Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2019

Trabajo Fin de Máster: Auto-codificadores para detección y procesado de imágenes

Autor: Francisco Jesús Marchal Cebador

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, 2019

El Secretario del Tribunal

A mi familia

A mis amigos

A mis maestros

Agradecimientos

La realización de este proyecto significa el final de una etapa, de la que no solo me llevo conocimientos, sino también buenos momentos y grandes amistades. Es complicado recoger en unas líneas a todas las personas que me han acompañado en este camino y a las que agradecer que todo esto haya sido posible.

A mi tutor, Juan José Murillo, por darme la oportunidad de profundizar en este fascinante campo, por guiarme y ayudarme en todo lo necesario para que este trabajo saliera adelante.

A mis amigos y compañeros que me han acompañado durante todos estos años, con los que he pasado tantas horas y por todas las historias que hemos compartido. Tanto a los que siguen cerca como a los que se han marchado lejos en busca de un futuro profesional mejor. En especial a Francisco José Pérez y Daniel López, por acompañarme de nuevo en este máster y que a pesar de los momentos difíciles hayamos conseguido juntos llegar hasta el final. Gracias por estar ahí y facilitar este camino que comenzamos en primero de carrera. No quiero dejar de mencionar a Javier Soriano y Adrián Vélez, por todos esos ratitos después de clase o trabajo y por seguir manteniendo un hueco en la agenda para vernos de vez en cuando.

A mi familia, en especial a mis padres, Concha Cebador y Francisco R. Marchal, por dejarse la vida para que pueda cumplir este sueño. Gracias por vuestro esfuerzo y dedicación, por ayudarme, aconsejarme y acompañarme día tras día y en los momentos más difíciles. Sin vosotros esto no hubiera sido posible.

Francisco Jesús Marchal Cebador

Sevilla, 2019

Resumen

El procesado de imágenes de placas de Rayos X efectuadas sobre obras de arte permite obtener información para realizar estudios que son relevantes en el campo de la Historia del Arte. Poco se ha investigado acerca de aplicar técnicas de aprendizaje profundo sobre estas placas.

El objetivo de este proyecto es el estudio de los auto-codificadores, un tipo de red neuronal artificial de aprendizaje no supervisado, para aplicarlos sobre el procesado de imágenes. A partir de un análisis del fundamento teórico, se diseñan las bases del auto-codificador aplicándolo sobre detección de imágenes de dígitos manuscritos. Posteriormente, se diseña una nueva arquitectura para adaptarla a las placas de Rayos X de cuadros. Dicha arquitectura comprende el preprocesado de las imágenes, su introducción en el auto-codificador y la posterior reconstrucción de la salida del mismo. La idea es poder reducir el ruido que contienen las radiografías de forma que se puedan realizar futuros análisis más profundos. Los resultados se han comparado con PCA, otro algoritmo de aprendizaje máquina para la reducción de la dimensionalidad.

Las imágenes que se emplean en este trabajo han sido cedidas por el Museo Nacional del Prado y por el *RKD* (Instituto de Historia del Arte de los Países Bajos).

Abstract

Image processing of artwork X-rays allows to obtain information for relevant studies in Art History field. There are not many researches about applying Deep Learning techniques over these X-rays images.

The purpose of this project is a research of autoencoders, a kind of artificial neural network of non-supervised learning, for using them in image processing. From theoretical basis analysis, a basic autoencoder has been created for applying it over image detection of handwritten digits. Afterwards, a new architecture has been designed for adapting it to X-rays artworks processing. This architecture contains images preprocessing, autoencoder execution and its outputs reconstruction. The idea is to be able to reduce noise contained in X-rays for making deeper analysis of the paintings in a future. The results have been compared with PCA, another Machine Learning algorithm for dimensionality reduction.

The images used in this project have been transferred by *Museo Nacional del Prado* and by *RKD* (Netherlands Institute for Art History).

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

Todo lo que puedas imaginar es real.

- Pablo Picasso -

La inteligencia artificial (*AI*) y dentro de sus múltiples técnicas el concepto *Deep Learning* o "aprendizaje profundo" se encuentran hoy en día entre los términos más utilizados cuando hablamos de tecnología. El intento por conseguir emular el comportamiento del cerebro humano para que las máquinas puedan tomar sus propias decisiones y ayudarnos con infinidad de tareas se ha convertido en uno de los principales objetivos de las grandes organizaciones. Esto se ha visto incrementado con la revolución de los datos que nos ha llegado con términos como *Big Data* e *IoT*, en la que las empresas generan y almacenan una gran cantidad de información, la cual necesita ser procesada.

Dentro de este "aprendizaje profundo" existen multitud de arquitecturas que simulan la estructura de la red de neuronas del cerebro. Estas redes neuronales artificiales (*ANN*) se pueden implementar de diversas formas, entre las que se encuentran los *autoencoders* o "auto-codificadores", que no es más que un tipo de red neuronal no supervisada que aprende a producir a la salida la misma información que tiene a la entrada tras haber sido comprimida.

Las grandes compañías tecnológicas emplean las ANN para todo tipo de aplicaciones, desde recomendaciones basadas en las búsquedas, reconocimiento de patrones en textos e imágenes hasta asistentes personales, por mencionar algunos ejemplos. Pero cada vez más el *Deep Learning* es empleado en ámbitos totalmente diferentes como la banca, medicina, educación, etc. El abanico de oportunidades que se abre es inmenso. Tal es así que a día de hoy se inviertan grandes cantidades de dinero para investigar y fomentar los avances en este campo.

El campo de la Historia del Arte requiere de la interpretación de las pinturas y sus materiales para estudiar el origen de los lienzos, de forma que se puedan obtener características sobre la tela como por ejemplo el número de hilos o el reconocimiento de los patrones. Esto resulta importante de cara a restauraciones de las obras de arte o a descubrir otras obras del mismo autor.

Hasta hace poco este estudio de los cuadros se realizaba de forma visual y manual, pero gracias a la tecnología y al procesado de las imágenes se va automatizando y simplificando. Se pueden encontrar noticias recientes acerca de descubrimientos no visibles al ojo humano tras analizar las radiografías con nuevas técnicas digitales, como por ejemplo que Picasso al parecer pintó una de sus obras (*La miséreuse accroupie*) sobre otro cuadro[1], o el descubrimiento del retrato de una mujer oculta bajo la obra *Retrato de una niña* del pintor

[1] Los rayos X revelan dos obras ocultas en un cuadro de Picasso. Febrero 2018. Fuente: Business Insider España

Amadeo Modigliani[2].

Sin embargo, se ha investigado poco acerca de realizar este procesado de los Rayos X de cuadros de forma no supervisada mediante ANN. Esta idea ha motivado la realización de este proyecto, que se basa en el estudio de los auto-codificadores aplicados al procesado de imágenes.

El objetivo es que a partir de haber obtenido unas imágenes de Rayos X de los lienzos, estos puedan ser procesados con los auto-codificadores de forma que se intente obtener las características básicas y se pueda realizar una reconstrucción de las imágenes de Rayos X eliminando el ruido implícito en la radiografía, como pueden ser grietas, cortes, desgarros, etc. Entre las características a obtener, se pretende que se puedan visualizar de la mejor forma posible los hilos que forman el tejido del lienzo, para un futuro análisis de conteo o análisis de los patrones de entrelazado de éstos.

El estudio realizado en este proyecto se centra en un primer momento en entender correctamente el funcionamiento de un auto-codificador, sus características y tipos de arquitectura. Posteriormente se realiza un análisis de las herramientas software a utilizar para su implementación, y se hacen las primeras pruebas con imágenes de dígitos manuscritos. Esto nos va a dar una primera aproximación de la solución.

Tras ello, se procede a aplicar dicha implementación en las imágenes de Rayos X anteriormente mencionadas, cedidas por el Museo Nacional del Prado y por el *RKD* (Instituto de Historia del Arte de los Países Bajos).

[2] Descubierto un retrato de Modigliani debajo de una de sus obras maestras. Marzo 2018. Fuente: El País

2 FUNDAMENTO TEÓRICO

Todo el mundo y todo a tu alrededor es tu maestro.

- Ken Keyes -

El aprendizaje profundo (*Deep Learning*) es una técnica del aprendizaje máquina (*Machine Learning*) en la que varias capas se interconectan entre sí, de forma que cada capa está "profundamente" conectada a la anterior y toma las decisiones en función la salida de su capa previa. La forma más común de implementar esto es mediante las redes neuronales artificiales (ANN).

En este capítulo se expone de manera resumida el concepto de ANN, y a partir de ahí se introducen los fundamentos de los autoencoders, los cuales son el hilo conductor de este proyecto.

2.1 Redes neuronales artificiales

Una red de neuronas es una herramienta matemática que intenta simular el comportamiento de las neuronas que tiene el cerebro humano. En una neurona biológica, se reciben los impulsos (entradas) por las dendritas y genera una salida a través de su axón. Este axón tiene unas terminaciones que puede dividirse múltiples veces y conectarse mediante sinapsis a las dendritas de otras neuronas.

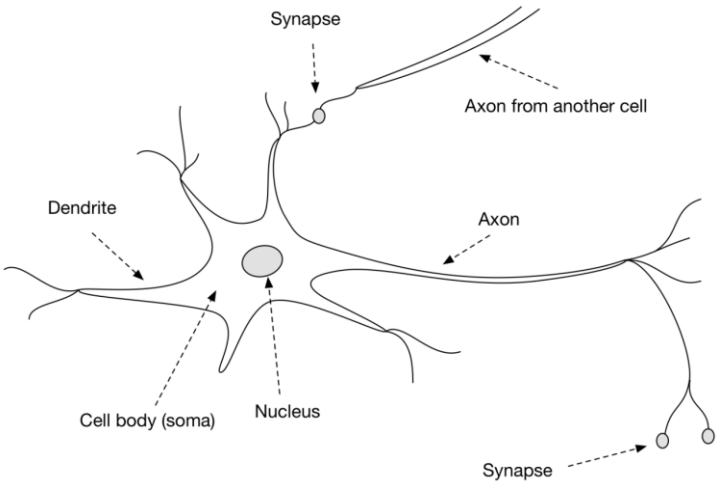

Figura 2-1. Neurona biológica (Fuente [1])

Extrapolando este concepto a nivel computacional, para cada neurona, a partir de unos valores de entrada (equivalente a las dendritas), se realiza una serie de operaciones matemáticas con dichas entradas dando lugar a otros valores a la salida (axón). Este modelo es denominado *perceptrón*.

En un perceptrón las dendritas llevan las distintas señales o valores de entradas hasta el núcleo, donde son sumados cada uno multiplicado por un peso w_i. También se le puede sumar un sesgo o *bias* (b). La salida es el resultado de pasar dicha suma por una función de activación, que en el caso más simple (perceptrón) suele ser una función escalón o función signo. La idea es que la conexión sináptica se consigue a partir de la influencia de una neurona sobre otra, mediante la actualización o aprendizaje de los pesos y el sesgo.

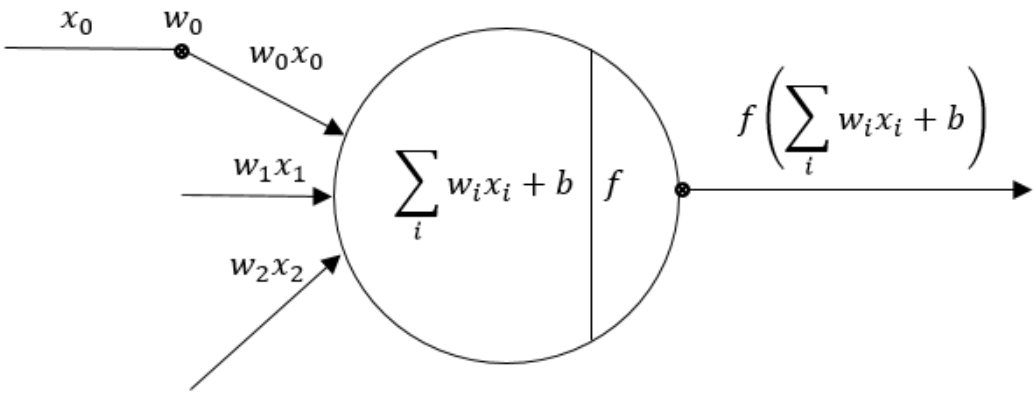

Figura 2-2. Esquema básico del perceptrón

En el modelo básico la salida sólo es capaz de reproducir clasificadores binarios, además de no poder resolver problemas no lineales. Para generalizar el modelo del perceptrón, aparece el concepto de redes de neuronas (también conocido como *perceptrón multicapa*), que no es más que un conjunto de neuronas conectadas.

Una red neuronal está organizada en capas, y generalmente está formada por una capa de entrada, varias capas ocultas y una capa de salida.

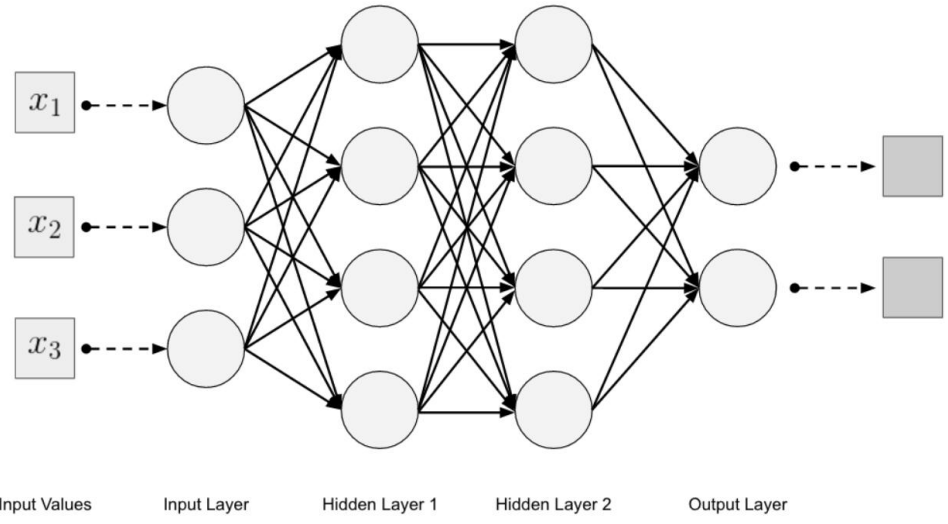

Figura 2-3. Ejemplo de red neuronal con capas completamente conectadas (Fuente [1])

En las ANN, las neuronas pueden usar funciones de activación diferentes a la función escalón o la función signo (utilizadas en el perceptrón), por lo que se tiene la posibilidad de implementar comportamientos no lineales. De forma general, se suelen utilizar funciones no lineales para las capas ocultas, mientras que para la capa de salida pueden no tener función de activación (salida lineal).

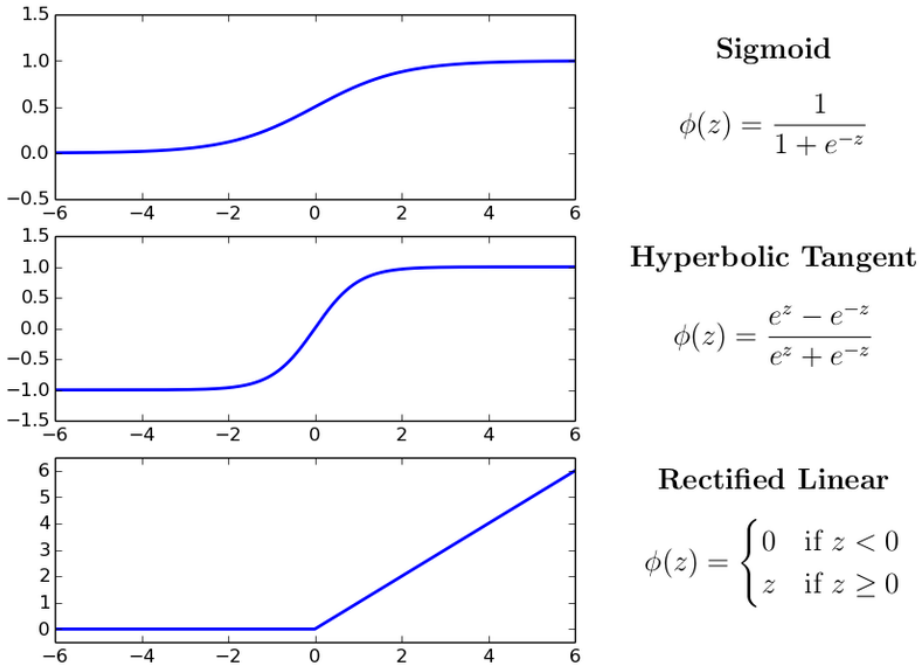

Figura 2-4. Funciones de activación no lineales comunes (Fuente: researchgate.net)

El proceso de aprendizaje para estas redes neuronales pasa por realizar varias iteraciones en las que se vayan actualizando los pesos y los sesgos, de forma que se reduzca el error, minimizando la función de coste.

El método de la retropropagación (*backpropagation*) es uno de los más utilizados y se aplica a las redes neuronales del tipo *feedforward*, es decir, que tienen una arquitectura en la que las conexiones entre las neuronas no forman bucles (este es el modelo más simple). Este procedimiento aplica el método de descenso de gradiente, es decir, emplear la regla de la cadena para distribuir el error a la salida de la red entre los diferentes parámetros de esta. El algoritmo de este proceso de aprendizaje puede encontrarse en el capítulo 2 del libro "*Deep learning: a practitioner's approach*" [1].

Un hiperparámetro importante de la retropropagación es la tasa de aprendizaje (*learning rate*), que controla la velocidad de aprendizaje de la red neuronal. Si es muy alto, los valores de los parámetros dan grandes saltos (alta probabilidad de saltarse la solución óptima), mientras que si es bajo los valores evolucionan muy lentamente (se tarda mucho en alcanzar la solución y se puede quedar en mínimos locales).

La función de coste a minimizar durante este proceso de aprendizaje va a depender de la tarea deseada. Un ejemplo de coste frecuente es el error cuadrático medio (*MSE*), muy utilizado en *backpropagation*. Dadas N muestras, denotamos la función de coste del MSE como:

$$L = \frac{1}{N}\sum_{i=1}^{N}\left(\hat{Y}_i - Y_i\right)^2 \tag{2–1}$$

siendo (X, Y) el conjunto de entrada valor-etiqueta, y $(\hat{X}, \hat{Y})$ el conjunto de valores de salida. En este caso para la función de coste solo se tiene en cuenta los valores de las etiquetas (Y), ya que se comprueba el error entre lo etiquetado durante el aprendizaje y la etiqueta real.

Este es un ejemplo para un tipo de aprendizaje supervisado, pero existen otras muchas funciones de coste según el uso como pueden ser regresión, clasificación multiclase, u otras que son aplicables también a métodos no supervisados.

2.1.1 Redes neuronales convolucionales

Hay aplicaciones en las que las redes neuronales tradicionales no se ajustan correctamente. Es el caso del procesado de imágenes, en el que si se aplicara una ANN se tendrían que tomar como datos de entrada cada uno de los píxeles de la imagen, lo que haría que cuando se introdujeran en la red generaría un número inmenso de parámetros que hacen muy difícil que se pueda trabajar con ellas.

Las redes neuronales convolucionales (CNN) aparecen para dar respuesta a la necesidad de disminuir el número de parámetros de la red, teniendo en cuenta la disposición espacial de los datos de entrada.

Este tipo de redes están siendo muy utilizadas para aplicaciones de visión artificial tales como coches autónomos, robots, drones, y tratamiento de imágenes de diverso tipo como por ejemplo para pruebas médicas.

Cada neurona tiene una disposición en 3 dimensiones: altura, anchura y activación (profundidad). Para el caso de la capa de entrada, estas dimensiones coinciden con la de una imagen en la que se tiene altura y anchura, y la profundidad se corresponde con los canales RGB (en el caso de imágenes en blanco y negro sólo tendrá una capa de profundidad).

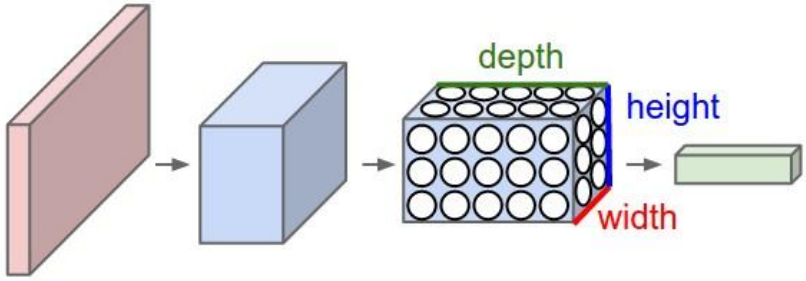

Figura 2-5. Neurona de 3 dimensiones de una CNN (Fuente [2])

Tras la capa de entrada, se tienen una serie de capas para la extracción de características de los datos introducidos. Se pueden distinguir dos tipos de capas principalmente, las convolucionales y las de sub-muestreo (*pooling*).

En las capas convolucionales cada neurona forma un filtro que se aplica a una parte concreta de la imagen. El filtro se va a ir desplazando ("stride") sobre los datos de entrada, realizando un producto convolucional entre el fragmento de datos de entrada y los pesos del filtro.

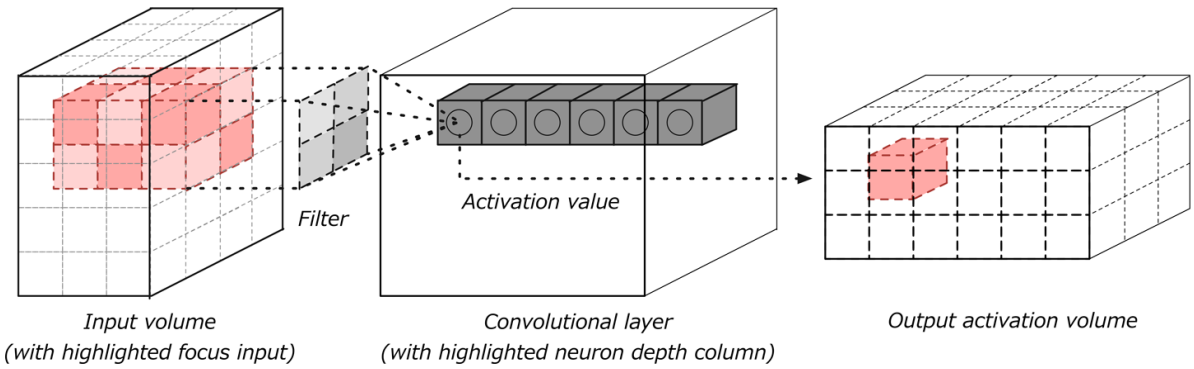

Figura 2-6. Capa convolucional (Fuente [1])

Existe una serie de hiperparámetros o decisiones que se pueden tomar sobre estas capas convolucionales, como por ejemplo el *stride* o paso de desplazamiento, que se detallarán en el apartado 3.1.2 y que hace que la red tenga un cierto comportamiento. Esto será el objetivo principal de las pruebas realizadas cuando se implemente el autoencoder, de forma que se busque modificando dichos parámetros la mejor solución tras entrenar el modelo.

La capa de *pooling* suele introducirse entre dos capas convolucionales. El objetivo es reducir el tamaño de los datos de salida de la capa convolucional para que a la salida de la red se pueda emplear una capa completamente conectada para obtener el resultado deseado de la red neuronal. Esto es muy típico por ejemplo en problemas de clasificación y se verá luego que para un *autoencoder* se emplea también esta capa para la parte del "codificador".

En esta capa no se reduce la profundidad de los datos, sino el alto y ancho de la imagen. Para hacer este sub-muestreo suelen emplearse dos métodos diferentes: aplicar la función máximo entre los valores (*max-pooling*) o realizar un promedio de ellos.

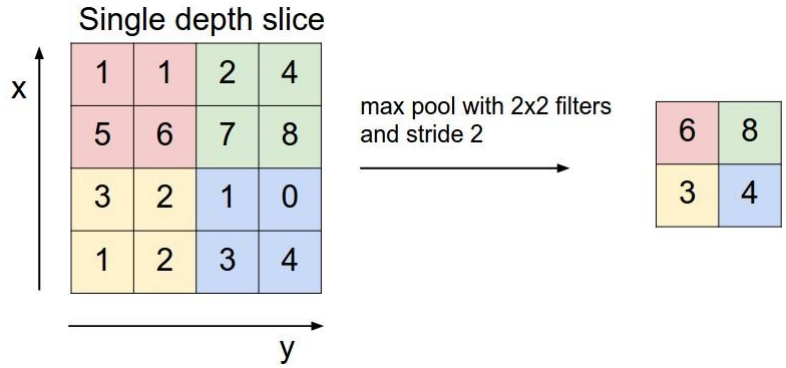

Figura 2-7. Max Pooling de 2x2 (Fuente [2])

Con esto, la estructura general de una red neuronal convolucional está compuesta por la capa de entrada, las capas de extracción de características (formadas por capa convolucional, función de activación y capa de pooling) y por último una capa completamente conectada que generará la salida.

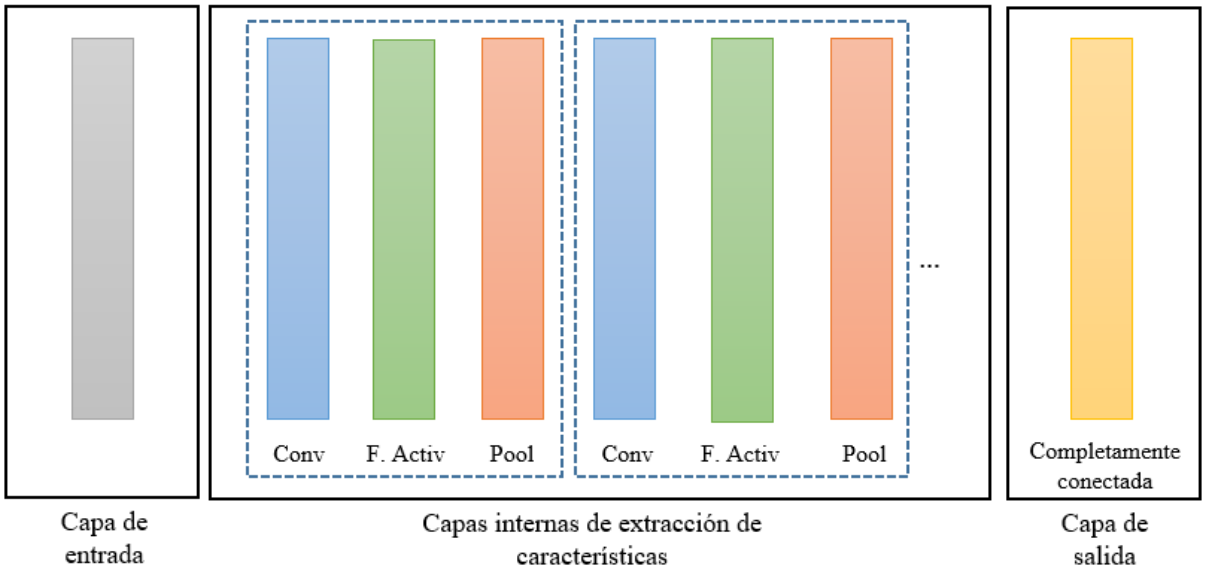

Figura 2-8. Estructura general de CNN

Esta estructura es la base para diseñar el modelo de red convolucional, que será modificada en función de los resultados que se quieran obtener. Al final del siguiente apartado se puede observar un ejemplo de estructura para un auto-codificador convolucional, siguiendo el modelo que se implementará posteriormente.

2.2 Auto-codificadores

Las redes neuronales hasta ahora descritas son aplicables a métodos de aprendizaje supervisado, donde los datos de entrenamiento incluyen etiquetas para su posterior aprendizaje para una clasificación.

Un auto-codificador es un algoritmo de aprendizaje no supervisado, el cual aprende de forma autónoma a producir a la salida la misma información que recibe a la entrada. Para ello, comprime la entrada a un espacio de variables oculto (código), y posteriormente reconstruyendo a la salida a partir de la información adquirida de la imagen comprimida.

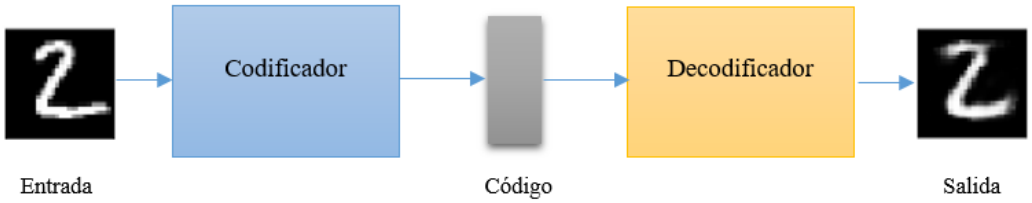

Figura 2-9. Funcionamiento básico de un autoencoder

Se puede intuir que el propósito de un auto-codificador es aprender una aproximación de la función identidad. Esto parece ser algo trivial, pero en realidad en función de los requisitos con los que se configure la red se puede descubrir en el código una estructura interesante que nos proporcione información importante acerca de los datos de entrada.

El objetivo principal de un auto-codificador es la reducción de la dimensionalidad. Sin embargo, el hecho de que el código retenga datos relevantes sobre la entrada hace que sea de gran utilidad para otras aplicaciones como por ejemplo la reducción de ruido en imágenes o la clasificación a partir de dicha estructura obtenida. A esto se une la ventaja de que sea un método no supervisado, es decir, no requiere que se le especifiquen etiquetas para ser entrenado, sólo se le pasan los datos de entrada.

La red de un autoencoder consta de dos partes:

- Codificador (encoder): Comprime los datos de entrada, con el objetivo de obtener las características principales de dichos datos. La salida del codificador es la denominada capa oculta del autoencoder o código (h). Para la entrada x, con una matriz de pesos W y un sesgo b:

$$h = f(x) = f(Wx + b) \tag{2–2}$$

- Decodificador (decoder): Reconstruye la entrada a partir de las características recogidas en el codificador. En este caso la entrada del decodificador será h (salida del codificador), con una matriz de pesos W' y un sesgo b'.

$$r = g(\big(f(x)\big) = g(W'h + b') \tag{2–3}$$

Los autoencoders se entrenan con las mismas técnicas que una red neuronal tradicional, mediante back-propagation (método del descenso de gradiente).

De forma general, existen tres hiperparámetros a tener en cuenta a la hora de diseñar un autoencoder:

- Número de capas: hasta ahora se ha presentado el autoencoder como una red de tres capas. Sin embargo, la capa oculta puede ser tan profunda como se quiera.
- Número de neuronas por capa: Esto se verá en las diferentes pruebas que se realicen a nivel práctico. En función de este número la red se comportará de una manera u otra. Habitualmente, el número de neuronas va disminuyendo o se mantiene igual conforme se avanza en las capas del codificador, y luego se reestablecen en el mismo orden para el decodificador. Para el caso de las neuronas en el

código (capa oculta central), cuanto menor tamaño mayor compresión.

- Función de pérdidas: Esto se verá en el siguiente sub-apartado para cada tipo de arquitectura. Algunas de las más usadas son el error cuadrático medio (MSE) o la entropía cruzada binaria.

2.2.1 Tipos de arquitectura

La dimensión de la capa oculta va a diferenciar dos grandes grupos en la clasificación de estas redes: uno en el que la dimensión de *h* es inferior a la entrada y la salida (*undercomplete autoencoder*), y otro donde el código *h* es superior a las otras dos capas (*regularized autoencoder*).

2.2.1.1 Undercomplete autoencoder

La dimensión de la capa oculta (código o *h*) es de menor dimensión que la entrada: $h < x$. Este es el caso más básico de autocodificador, en el que las neuronas del código son menores que la entrada y por tanto se produce una compresión que permite que genere una estructura con la información más relevante.

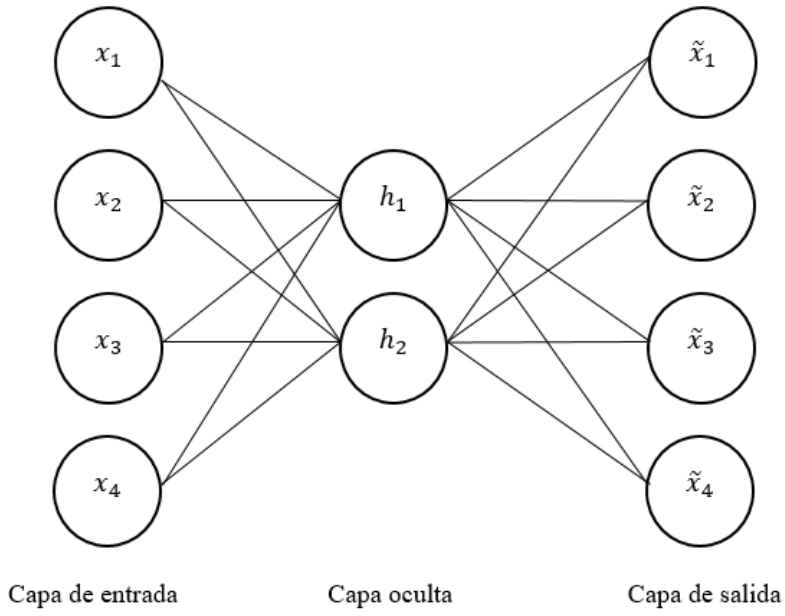

Figura 2-10. Esquema undercomplete autoencoder

En el gráfico anterior, la salida será: $r = \tilde{x}_i$, siendo $\tilde{x}_i$ la neurona equivalente a la entrada x_i tras ser decodificada. Esta estructura básica de tres capas puede extenderse a un autoencoder multicapa, en la que la capa oculta se extiende a un número mayor de capas para lograr una mejor generalización. Las neuronas de cada una de las capas ocultas, al igual que en el caso de las redes neuronales tradicionales, tendrán sus respectivos pesos ($W_i^{(j)}$), los cuales serán aprendidos durante el entrenamiento de la red.

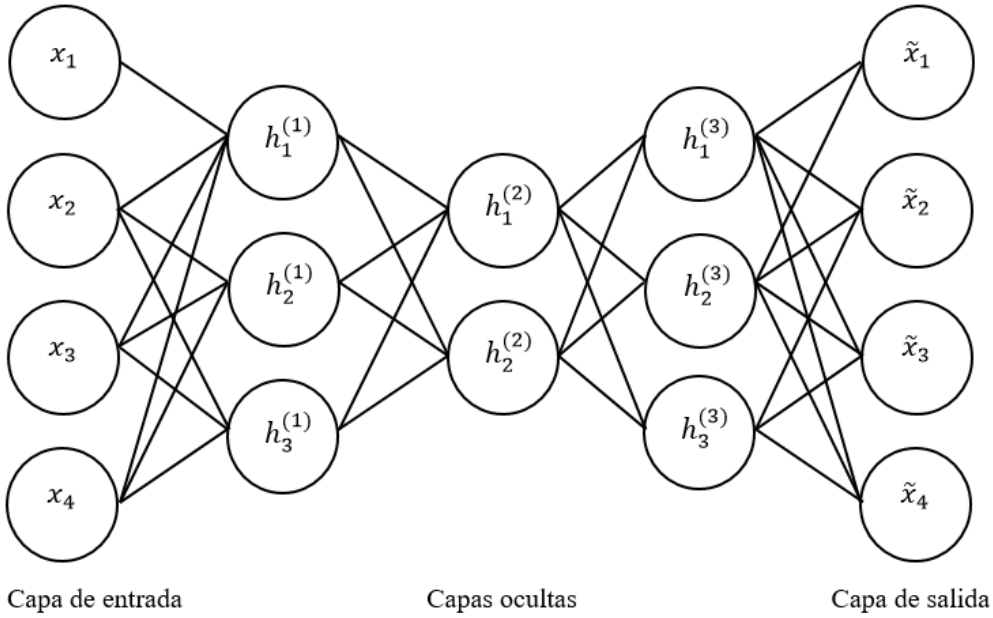

Figura 2-11. Esquema autoencoder multicapa

El objetivo de cualquier auto-codificador es minimizar una función de pérdidas:

$$\min L\left(x, g\left(f(x)\right)\right) \tag{2–4}$$

Para el caso del error cuadrático medio (MSE), en la que se mide la distancia entre la salida y la entrada:

$$L = \left|\left|x - r\right|\right|^2 \tag{2–5}$$

Cuando el decoder es lineal y *L* es el error cuadrático medio, el autoencoder aprende el mismo subespacio que si se utilizara PCA. El auto-codificador entrenado aprendería el subespacio principal de los datos de entrenamiento.

Si los datos son un vector de valores binarios o un vector de probabilidades entre 0 y 1, también se puede emplear la función de pérdidas de la entropía cruzada, en la que se calcula cuántos bits de información se conservan en la salida en comparación con la entrada:

$$L = -\sum_{k=1}^{d} x_k \log r_k + (1 - x_k) \log(1 - r_k) \tag{2–6}$$

2.2.1.2 Regularized autoencoder

Cuando la capa oculta tiene una dimensión igual o superior a la capa de entrada (*overcomplete autoencoder*), la red puede aprender a copiar sin extraer información útil sobre la distribución de los datos, es decir, funcionaría como una matriz identidad.

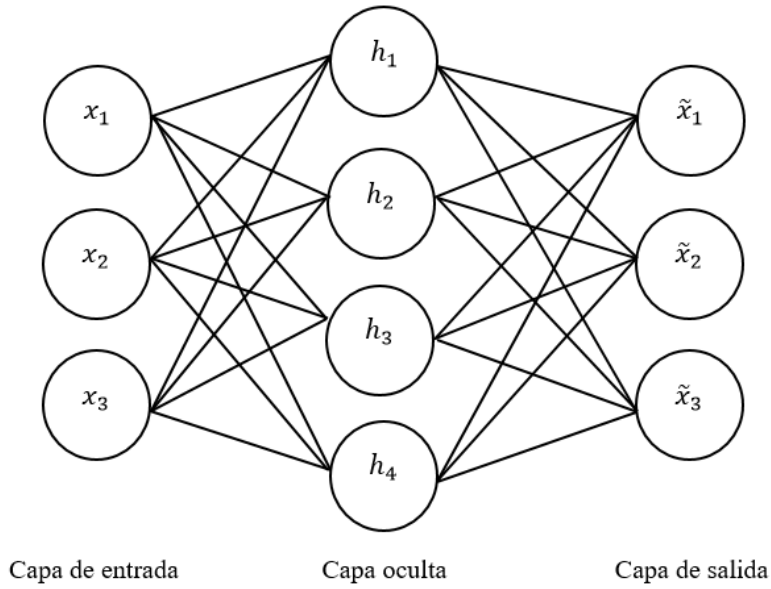

Figura 2-12. Esquema overcomplete autoencoder

El objetivo de los auto-codificadores regularizados es precisamente el de entrenar cualquier arquitectura con éxito, eligiendo la dimensión del código y la capacidad (complejidad) del codificador y del decodificador para que se ajuste correctamente, independientemente de si el tamaño del código es igual o superior a la entrada.

La regularización sirve de ayuda para controlar la complejidad del modelo, asegurando que es capaz de tomar mejores decisiones, es decir, conseguir que generalice lo mejor posible.

Este tipo de autoencoders usan una función de pérdidas que permiten al modelo tener otras propiedades además de la capacidad de copiar la entrada en la salida, como la dispersión de la representación, la disminución de la derivada de la representación y la robustez frente a ruido o entradas faltantes.

Existen dos tipos de auto-codificadores regularizados, los cuales se analizan a continuación.

2.2.1.2.1 Sparse autoencoder (SAE)

El funcionamiento de un autoencoder disperso (*sparse*) se basa en aplicar una restricción de dispersión.

La restricción a aplicar puede variar en función de dónde se aplique. Por un lado se tiene la técnica denominada como "dropout", en la que solo un número determinado de neuronas se activarán y se aplicará de forma aleatoria sobre las neuronas de la red.

Una neurona está "activa" si su valor de salida está cercano a 1, y por el contrario, estará "inactiva" si su salida es cercana a 0. Esto hace que se puedan emplear más neuronas en la capa oculta, pero no todas se utilizarán para el aprendizaje. Se evita así que la capa oculta funcione solo como una matriz identidad, que solo sirva para copiar la entrada. Esta técnica se expondrá posteriormente en el apartado 3.1.2.4.

Por otro lado, se puede aplicar la restricción sobre la función de pérdidas. A diferencia de otras regularizaciones, aquí la regularización depende directamente de los datos. El criterio de entrenamiento (función de pérdidas) de un auto-codificador disperso introduce una penalización o restricción de dispersión sobre la capa oculta (h), siendo h la salida del codificador: $h = f(x)$. Esto significa que la limitación añadida va a cambiar dependiendo de los valores de entrada.

$$L\left(x, g\left(f(x)\right)\right) + \Omega(h) \tag{2–7}$$

La restricción más utilizada es la basada en el concepto de divergencia de *Kullback-Leibler* (KL), que es una función estándar para medir cómo son dos distribuciones. En este caso, se mide la desviación entre una

variable aleatoria de Bernoulli con media ρ (parámetro de dispersión) y una variable aleatoria de Bernoulli con media $\hat{\rho}_j$ (activación media de la neurona en h). Será este parámetro $\hat{\rho}_j$ el que dependa del valor de entrada.

$$\Omega = \beta \sum_{j=1}^{s_h} KL(\rho||\hat{\rho}_j) = \beta \sum_{j=1}^{s_h} \rho \log\frac{\rho}{\hat{\rho}_j} + (1-\rho)\log\frac{1-\rho}{1-\hat{\rho}_j} \tag{2–8}$$

El término s_h es el número de neuronas en la capa oculta. Se añade también el parámetro β, que se encarga de controlar el peso que va a tener el parámetro de penalización dentro de la función de pérdidas.

Sin embargo, existen otras restricciones más sencillas, que serán las empleadas posteriormente en las pruebas realizadas. Dichas restricciones se denominan de nivel 1 ("L1") o nivel 2 ("L2") y se aplican directamente sobre los pesos de la red neuronal. La variable λ es un hiperparámetro que controla la "cantidad" de regularización que se está aplicando sobre la red.

Para el caso de "L1", la regularización se calcula como la suma de los valores absolutos de los pesos:

$$\Omega = \lambda R(W) = \lambda\left(\sum_i \sum_j |W_{i,j}|\right) \tag{2–9}$$

Para el caso de "L2", se calcula como la suma de pesos al cuadrado:

$$\Omega = \lambda R(W) = \lambda\left(\sum_i \sum_j W_{i,j}^2\right) \tag{2–10}$$

Además, este tipo de restricciones pueden aplicarse directamente sobre la salida de una capa concreta, es decir, la regularización se aplica sobre la función de activación de la capa en cuestión. Esto se denomina "regularización de actividad".

2.2.1.2.2 Denoising autoencoder (DAE)

En lugar de añadir una penalización Ω a la función de coste, se puede obtener un autoencoder que aprenda características útiles cambiando el término de la reconstrucción del error en la función de coste.

DAE reconstruye la entrada x a partir de una versión con ruido. Por tanto, la función a minimizar sería la mostrada a continuación, siendo $\hat{x}$ una copia de x que ha sido alterada por algún ruido.

$$L\left(x, g\left(f(\hat{x})\right)\right) \tag{2–11}$$

Por tanto, un DAE debe deshacer esta corrupción de los datos en lugar de simplemente copiar la información. Se fuerza a las funciones f y g a aprender implícitamente la estructura de los datos. El procedimiento de entrenamiento se puede observar en la siguiente figura.

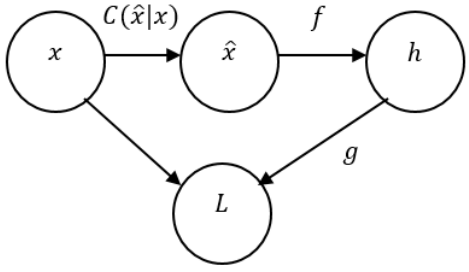

Figura 2-13. Proceso de entrenamiento DAE

Se introduce el ruido mediante el proceso $C(\hat{x}|x)$, que representa una distribución condicional sobre las muestras con ruido $\hat{x}$. El autoencoder es entrenado para reconstruir un punto x a partir de una versión $\hat{x}$ con ruido.

Esto se logra minimizando la función de pérdidas:

$$L = -\log p_{decoder}\big(x|h = f(\hat{x})\big) \tag{2–12}$$

Típicamente, $p_{decoder}$ es definido por una función decodificador $g(h)$.

Este tipo de auto-codificadores pretenden solucionar el problema de que cuando las capas ocultas son de mayor dimensión que la entrada se copie directamente la información sin obtener datos relevantes.

No obstante, un DAE permite eliminar el ruido que proviene de las muestras de entrada. Es por ello, que aunque aquí se haya detallado el proceso de incluirle ruido "artificialmente", esto es aplicable directamente a una capa de entrada con ruido ya implícito. Esto es lo que se hará posteriormente cuando se apliquen los autoencoders a imágenes de Rayos X.

2.2.1.3 Otras arquitecturas

Además de las arquitecturas más comunes que se han expuesto, existen otros tipos de auto-codificadores, los cuales se resumen a continuación.

2.2.1.3.1 Contractive autoencoder (CAE)

Al igual que en el caso de DAE, el objetivo de los auto-codificadores contractivos es hacer que la representación aprendida sea robusta frente a pequeños cambios en la entrada.

Para ello, se agrega un nuevo término de penalización a la función de pérdidas, correspondiente a la norma de Frobenius de la matriz jacobiana[3], la cual contiene una derivada parcial del valor de activación de una neurona con respecto al valor de entrada. Con esto, cuando se aumenta el valor de activación, aumentará el jacobiano, penalizando la representación.

$$L_{CAE} = \sum (L\left(x, g\big(f(x)\big)\right) + \lambda \|J_h(x)\|^2_F) \tag{2–13}$$

2.2.1.3.2 Variational autoencoder (VAE)

A diferencia de los usos típicos de una red neuronal como puede ser la clasificación o la regresión, los auto-codificadores variacionales son potentes modelos generativos, que actualmente está muy de moda en aplicaciones como por ejemplo generar rostros humanos falsos o producir música sintética.

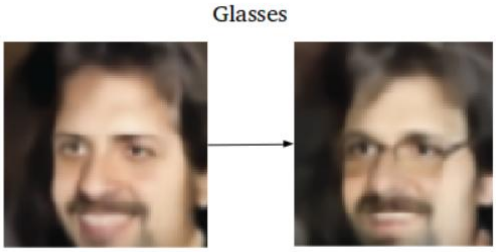

Figura 2-14. Ejemplo de modelo generativo (Fuente [3]]

[3] La norma de Frobenius de la matriz jacobiana es como una generación euclídea de la norma. Puede encontrarse más información en: Kristiadi A. Deriving Contractive Autoencoder and Implementing it in Keras [12]

Con un modelo generativo lo que se quiere es generar una salida aleatoria nueva, en función de los datos de entrenamiento. También se puede especificar una dirección para la salida, de forma que sea como se desee y no tan aleatoria.

El problema de usar los auto-codificadores normales para este tipo de aplicaciones es que el espacio codificado (la capa oculta) puede no ser continua o que permita una fácil interpolación.

En los VAE se busca que el espacio codificado permita un muestreo fácil e interpolación. Para ello, se construye una distribución probabilística para estimar la distribución de las características. Se generan dos vectores: un vector de medias () y otro vector de desviaciones típicas ().

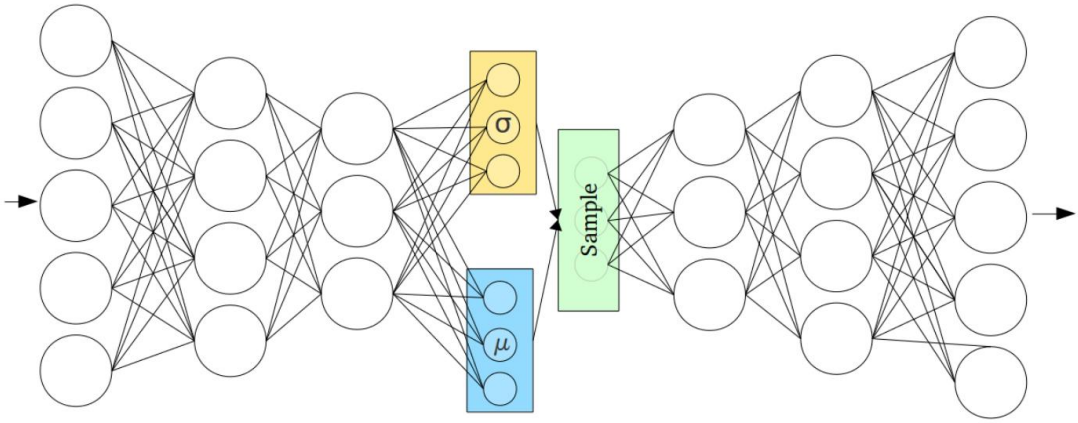

Figura 2-15. Esquema de VAE (Fuente [3])

2.2.1.3.3 Auto-codificadores apilados

A pesar de no ser una nueva estructura como tal, es necesario mencionar en este apartado el caso del apilado de auto-codificadores. Con un solo autoencoder podemos encontrar las características fundamentales (simples) de los datos de entradas. Pero si queremos que la red neuronal detecte conceptos más complejos (por ejemplo un rostro en una imagen), podría ser necesaria una red más compleja.

Es por ello que aparece el término de autoencoders apilados, que no es otra cosa que usar varios auto-codificadores, y entrenarlos uno a uno, de forma que la salida de un autoencoder sirva para entrenar el siguiente.

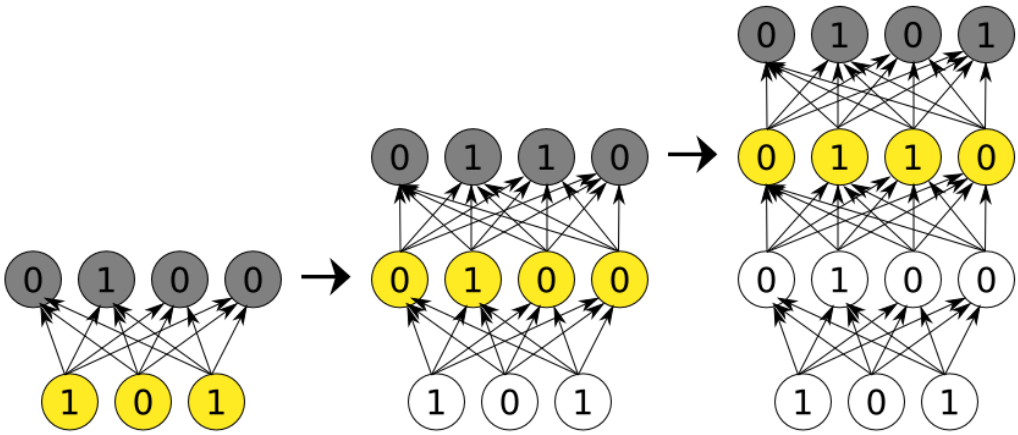

Figura 2-16. Ejemplo de auto-codificador apilado (Fuente [4])

2.2.2 Auto-codificadores convolucionales

Todos los conceptos y arquitecturas detalladas anteriormente para un auto-codificador pueden ser extrapolados a una red neuronal convolucional. El diseño de un auto-codificador convolucional es superior a los autoencoders apilados, incorporando relaciones espaciales entre los píxeles de las imágenes.

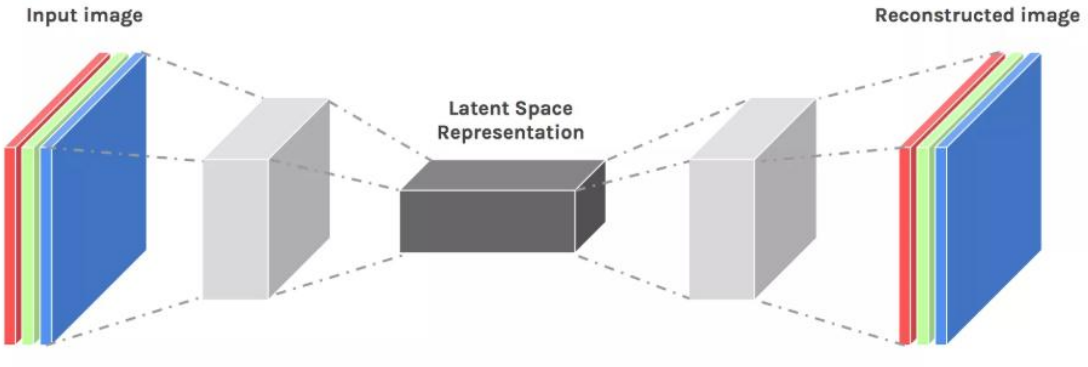

Figura 2-17. Auto-codificador convolucional (Fuente [5])

Este tipo de autoencoders serán los empleados principalmente para el procesado de imágenes (serán implementados posteriormente para la aplicación objeto de este proyecto).

Su estructura es la misma que la de una red neuronal convolucional, pero teniendo en cuenta que la red tendrá dos partes y debe ser simétrica: codificador y decodificador. El hecho de que la red sea simétrica significa que en la parte del decodificador, las capas internas deben de comportarse de forma inversa a como lo hacen en el codificador.

Para el codificador, las capas internas serán las que se expusieron en el apartado 2.1.1: capa de convolución y capa de *MaxPooling*. Con ellas se va comprimiendo la entrada hasta llegar a la capa central o código.

Como se ha indicado, a la hora de decodificar es necesario realizar el proceso inverso al realizado en el codificador. Para ello, lo lógico sería pensar que se debe realizar una especie de "deconvolución". En realidad lo que se quiere es poder realizar una convolución en la que se pase de una imagen de baja a alta resolución. Existen dos métodos para realizarlo, que suelen en ocasiones confundirse como el mismo.

El primero de los métodos consiste en emplear una capa de *Upsampling* y tras ella una capa de convolución clásica (esta será igual que la de convolución del codificador). En la siguiente figura puede verse un ejemplo del proceso de *Upsampling,* con el que se sobremuestrea la imagen repitiendo los píxeles.

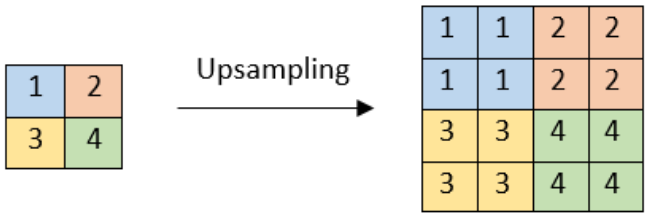

Figura 2-18. Ejemplo de Upsampling

El segundo método consiste en realizar lo denominado como convolución traspuesta (*Transpose Convolution*), que consiste en vez de sobremuestrear la imagen, se realiza un relleno con ceros de la matriz y se colocan los píxeles en la primera celda de cada sub-matriz (*unpooling*). Tras ello, igualmente se aplica una capa de convolución clásica. Este método es implementado directamente en algunas librerías y puede emplearse como una única capa. En la siguiente figura puede verse este *unpooling* en el mismo ejemplo.

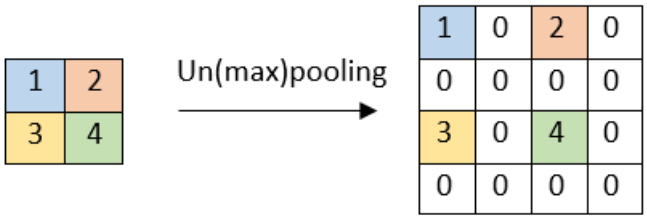

Figura 2-19. Ejemplo de Unpooling para convolución traspuesta

La convolución traspuesta es una aproximación del gradiente de la convolución, mientras que el primer caso podría considerarse como una suma de la convolución. Al final ambos métodos tienen el mismo objetivo, aunque con procedimientos diferentes. En el caso del *Upsampling* combinado con convolución se obtienen unos resultados algo más suaves que con el método de la convolución traspuesta.

En la imagen que se muestra a continuación se recoge un ejemplo de auto-codificador convolucional y su conjunto de capas. Se ha omitido en este grafo las capas correspondientes a las funciones de activación (que van a ir siempre de la mano de las capas convolucionales). El código (*h*) se ha representado como una capa adicional pero realmente es la salida del último pooling del codificador. La salida del autoencoder se trata de una última capa de convolución, que como se verá durante la implementación tendrá una función de activación diferente a las utilizadas a las anteriores, y viene a sustituir a la capa completamente conectada de las redes convolucionales tradicionales.

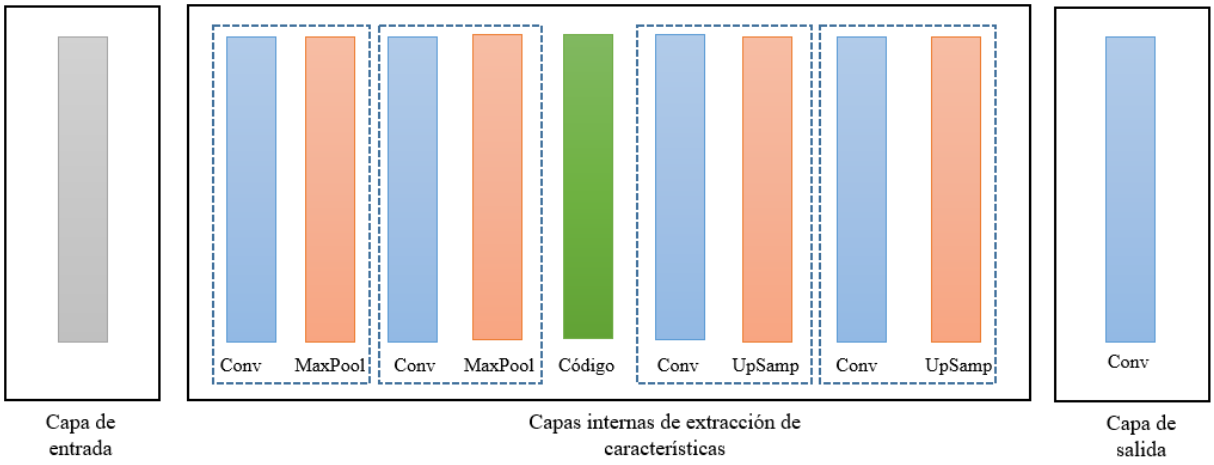

Figura 2-20. Estructura ejemplo de auto-codificador convolucional

2.2.3 Comparativa con PCA

Una de las principales propiedades de los auto-codificadores es la reducción de la dimensionalidad del espacio de características. Como se ha expuesto anteriormente, su objetivo es conseguir un código de dimensión menor con el que se obtengan las características fundamentales de los datos de entrada. Esto es necesario cuando se tiene un número elevado de dimensiones en los datos de entrada, porque la aplicación de algoritmos sobre ellos se vuelve difícil y costosa.

Existen diferentes métodos en Machine Learning para reducir la dimensionalidad, pero el más común es el conocido como Análisis de Componentes Principales (PCA). Su objetivo es disminuir el conjunto de datos (vector x) en otro de dimensionalidad menor (vector y) mediante una transformación ortogonal. La idea es poder encontrar un conjunto de ejes sobre los que los datos tienen mayor varianza, y usar dichos ejes para proyectar los valores.

El vector y será un conjunto de variables incorreladas, denominadas componentes principales. El primer componente constituye la mayor varianza de los datos y es igual al mayor autovalor de la matriz de

covarianza. Los componentes siguientes del vector representan las siguientes varianzas, todas en orden decreciente con respecto a los autovalores de la matriz de covarianza. Para el cálculo de autovalores y autovectores puede emplearse el método matemático de Descomposición en Valores Singulares (SVD).

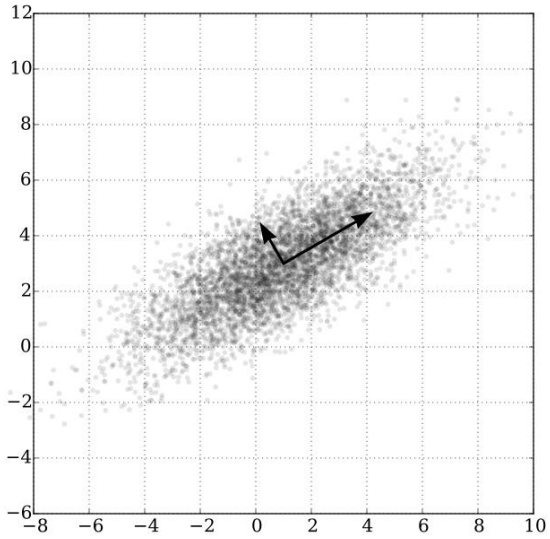

Figura 2-21. Ejemplo de PCA sobre una distribución Gaussiana multivariable (Fuente: Wikipedia)

Un auto-codificador como ya se ha visto tiene una capa de entrada, una capa oculta y otra de salida. La reducción de la dimensionalidad se produce precisamente en la capa oculta, en el que se reduce el tamaño de los datos de entradas. Cuando el AE se activa mediante funciones lineales su resultado se aproxima (casi equivalente) al que se puede dar usando PCA.

Sin embargo, las redes neuronales permiten introducir funciones de activación no lineales, lo que convierte esto en una ventaja del auto-codificador con respecto a PCA. La no linealidad permite explorar modelos más complejos y obtener buenos resultados.

Además, cuanto mayor es el número de características más lento será el procesado con PCA, ya que incrementa el cálculo a realizar. Por ejemplo, para una imagen de 256x256, la dimensión será de 65536. La matriz de covarianza será por tanto de 65536x65536, por lo que será excesivamente costoso encontrar SVD. En esto también mejora un auto-codificador, ya que las transformaciones son sencillas para conseguir el valor de los pesos. Esta es una de las principales características de las redes neuronales con respecto a otros algoritmos de *Machine Learning*.

Por otro lado, la función objetivo de PCA suele tener una solución óptima global, mientras que en los AE su función objetivo puede dar lugar a muchos óptimos locales. Esto realmente puede convertirse en un problema en el caso de los autoencoders, ya que el resultado obtenido tras el entrenamiento puede no ser el esperado, y que el algoritmo se haya quedado en un óptimo local.

3 PARÁMETROS Y HERRAMIENTAS DE DISEÑO

Vivo con esperanzas de todo, pero si no lucho nunca,
te aseguro que no llegará.

- Beret -

Para el diseño de un auto-codificador pueden emplearse varias soluciones que proporcionen herramientas para ejecutar algoritmos de Deep Learning. Además de poner en marcha dicha red neuronal, será necesario emplear otro tipo de software para tareas de tratamiento de las imágenes, etc. En este capítulo se detallan las tecnologías empleadas en el proyecto, en términos de herramientas y programas utilizados.

3.1 Decisiones de diseño

A la hora de diseñar una red neuronal se tiene una serie de herramientas matemáticas o parámetros modificables que hacen que el sistema se comporte de una manera u otra. Aunque esto ya se ha expuesto en el capítulo anterior durante la explicación de las redes neuronales, se van a detallar aquí las que han sido empleadas concretamente para este trabajo.

3.1.1 Tipos de no linealidad

Cada neurona que forma el auto-codificador va a tener una función de activación, que va a hacer que la salida se active o no en función del valor tomado. Como ya se ha comentado se pueden tener diversos tipos de funciones de activación, tanto lineales como no lineales.

A lo largo de las pruebas realizadas en este proyecto las funciones empleadas son no lineales. A pesar de la cantidad de funciones disponibles, se ha decidido tomar las dos que se muestran a continuación por su buen comportamiento en las pruebas realizadas así como por ser las más empleadas en las diferentes aplicaciones de ejemplo que se han ido analizando.

3.1.1.1 ReLU

ReLU (Rectified Linear Unit), también conocida como rectificador, es la típica función rampa que se define de la siguiente forma:

$$f(x) = \max(0, x) = \begin{cases} 0 & si \quad x < 0 \\ x & si \quad x \geq 0 \end{cases} \tag{3–1}$$

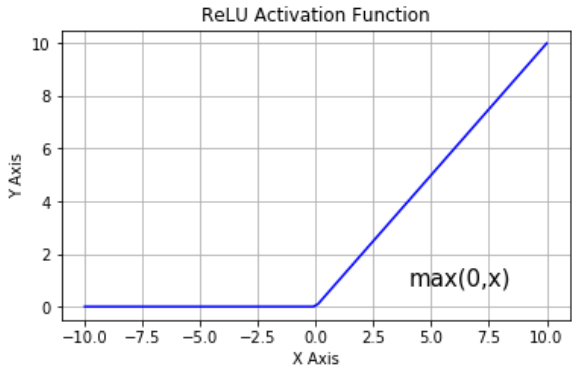

Figura 3-1. Función ReLU (Fuente: [Learn OpenCV])

La ventaja de utilizar ReLU frente a otras funciones es que es matemáticamente muy sencilla y de bajo coste computacional. El gradiente no satura en la parte positiva y hace que se acelere mucho el proceso de aprendizaje de la red neuronal al converger rápidamente. Sin embargo, tiene como inconveniente que el gradiente si cae en la parte negativa puede hacerse cero, y que no está centrada en cero.

Existen otras variaciones de esta función de activación, como por ejemplo *Leaky ReLU* y *Exponential Linear Unit* (ELU), en las que se busca evitar la anulación del gradiente en la parte negativa.

En el caso de la implementación del auto-codificador de este trabajo, se empleará ReLU a la salida de todas las capas internas de la red neuronal.

3.1.1.2 Sigmoide

La función sigmoide ha sido popular desde el comienzo de las ANN por tener respuestas similares a las neuronas biológicas. Lo que hace es que toma un valor y lo comprime entre 0 y 1. Su ecuación es la siguiente:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3–2}$$

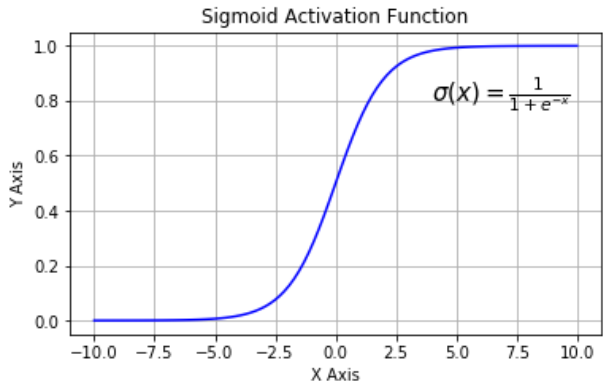

Figura 3-2. Función sigmoide (Fuente: [Learn OpenCV])

A pesar de ser una de las funciones de activación más utilizadas, tiene una serie de inconvenientes. Por un lado, la saturación casi anula al gradiente al realizar *backpropagation* a lo largo de la red neuronal. El gradiente tomará valores muy cercanos a 0 o a 1 y no se actualizarán demasiado los pesos. Por otro lado, al igual que en el caso de la función ReLU, no está centrada en cero. Esto provoca que se pueda producir inestabilidad en el gradiente. Además, al contener una exponencial hace que el coste computacional sea alto.

En nuestro caso se va a emplear únicamente en la salida del modelo, es decir, tras la última convolución realizada en el decodificador. Se usa precisamente esta en lugar de ReLU por la compresión entre 0 y 1 a la salida (de cara a los píxeles de la imagen resultante).

3.1.2 Hiperparámetros

Los hiperparámetros o decisiones que pueden ajustarse en una red neuronal son muy importantes para el diseño de la misma. Para los modelos convolucionales este número de valores de diseño se ve incrementado. En el caso de los auto-codificadores siguen empleando los mismos parámetros ajustables que en las redes convencionales (sean o no convolucionales).

Se resumen a continuación los hiperparámetros que posteriormente en las aplicaciones prácticas de este proyecto se van a ir modificando en cada una de las pruebas realizadas. Todos son enfocados a un autoencoder convolucional, al tratarse de una aplicación a imágenes.

3.1.2.1 Stride

Es el valor del paso con el que se va a ir desplazando el filtro sobre la imagen. Cuanto menor sea este valor mayor solapamiento existirá con las ventanas contiguas, y mayor profundidad del volumen a la salida.

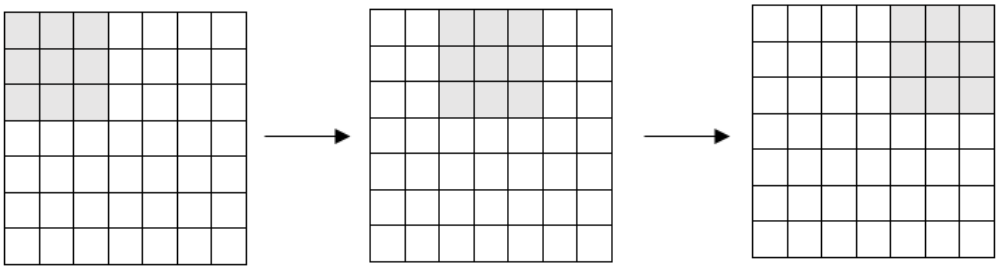

Figura 3-3. Ejemplo de stride con valor 2 para un filtro de 3x3

Sobre el ejemplo representado en la figura, la salida tendrá un tamaño de 3x3. De forma general, se puede calcular dicho tamaño de la siguiente forma (siendo W el tamaño de la imagen de entrada y F el tamaño del filtro aplicado):

$$tam_{salida} = \frac{W - F}{stride} + 1 \tag{3–3}$$

3.1.2.2 Zero-padding

Consiste en rellenar con ceros la salida para que conserve el tamaño deseado. A veces se quiere mantener el tamaño del volumen de entrada a la salida o también puede ocurrir porque el tamaño sea requerido por la capa siguiente.

Al añadir este relleno, se puede calcular el tamaño de salida de la capa convolucional como se muestra a continuación (siendo P el número de celdas de relleno a cada lado de la fila o columna a calcular):

$$tam_{salida} = \frac{W - F + 2P}{stride} + 1 \tag{3–4}$$

3.1.2.3 Batch-normalization

Es una normalización de los datos de entrada de la capa convolucional de forma que tengan una activación de salida media de cero y una desviación estándar de uno.

Se denomina normalización "por lotes" o “batch normalization” porque durante el entrenamiento, se normalizan las activaciones de la capa previa para cada lote.

Dicha transformación pasa por cuatro ecuaciones matemáticas, en las que se le pasa unos valores de entrada x_i y se aprenden dos parámetros durante el proceso de entrenamiento: γ y β. Existe otro valor que es ε que solo se emplea para evitar que se divida por cero. La salida se representará como y_i.

Los cuatro pasos o ecuaciones se realizan para cada uno de los denominados “mini-batch” o pequeño lote en los que se divide cada activación para estimar la media y la varianza de forma más sencilla.

$$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \tag{3–5}$$

$$\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \tag{3–6}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} \tag{3–7}$$

$$y_i \leftarrow \gamma\hat{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i) \tag{3–8}$$

Este es el proceso matemático que se encuentra detrás de la capa de Batch Normalization que posteriormente podrá ser empleada en herramientas software como Keras, simplemente llamando a una función como una capa más de la red neuronal.

3.1.2.4 Dropout

Consiste en una técnica de regularización en la que neuronas seleccionadas al azar son ignoradas durante el entrenamiento de la red. Con esto se evita que la red se adapte demasiado a las muestras utilizadas para el aprendizaje, evitando el sobreajuste (*overfitting*).

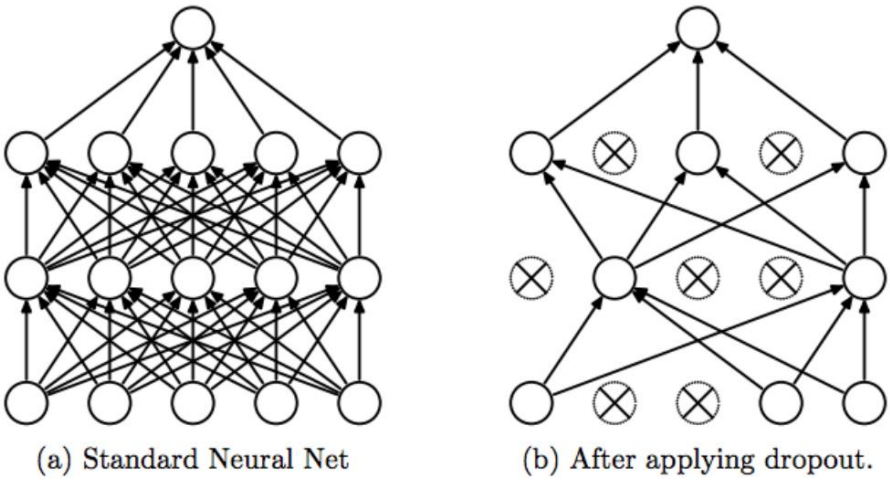

Figura 3-4. Ejemplo de aplicación de Dropout (Fuente [medium.com])

Se puede variar el valor de dicho dropout a partir de un rango entre 0 y 1. Por ejemplo, si añadimos un dropout de 0.2 (20%), significa que una de cada cinco neuronas se desactivará aleatoriamente en cada actualización.

Esta técnica será de bastante utilidad durante las pruebas a realizar posteriormente, ya que precisamente se querrá tener un modelo lo más general posible, que se adapte bien a cualquier muestra que sea introducida en éste, por lo que esta técnica evitará dicho sobreajuste durante el entrenamiento del autoencoder.

3.1.2.5 Batch size

Este parámetro define el número de muestras que se propagarán a través de la red neuronal, es decir, el número de muestras de entrenamiento que habrá en un pase hacia delante y hacia atrás.

Un ejemplo para que esto se entienda es que si se tienen un total de 1000 muestras de entrenamiento, y se toma un *batch-size* de 100, el algoritmo cogerá las 100 primeras muestras y entrenará la red, luego tomará las 100 siguientes empezando en la solución dada con las 100 muestras anteriores y volverá a entrenar la red, y así hasta llegar al final de las muestras.

Es importante destacar que este valor es diferente al número de iteraciones o "*epochs*" que se realizan durante el entrenamiento. Una "*epoch*" es el pase hacia delante y hacia atrás de todas las muestras de entrenamiento. Es decir, con el ejemplo anterior (con *batch-size* igual a 100), se necesitarán 10 iteraciones para completar una "*epoch*".

El valor de *batch-size* es destacable a la hora de realizar un entrenamiento, ya que dependiendo del mismo el aprendizaje se realizará con más o menos velocidad, y consumiendo una mayor o menor memoria. Esto es primordial dependiendo de las características del equipo donde se realicen las simulaciones.

Cuanto menor sea su valor menor memoria requerirá, ya que se entrenarán menos muestras a la vez. Sin embargo, cuanto menor sea el *batch* menor exactitud existirá a la hora de estimar el gradiente. Es por ello que se hace imprescindible encontrar un valor adecuado entre las especificaciones del equipo y los requisitos del modelo.

3.2 Software para auto-codificador

A la hora de seleccionar una herramienta software para implementar cualquier algoritmo de *Deep Learning* se encuentran numerosas opciones. En primer lugar, la decisión pasa por elegir el lenguaje de programación. Entre los más comunes para este objetivo, se encuentran varios tales como Python, R, Java, JavaScript, Matlab, C/C++, etc.

Para este proyecto, se ha seleccionado Python, por ser uno de los más utilizados. Esto se debe a su sencillez de uso y aprendizaje, y su gran potencial empleando librerías y herramientas que soportan gran cantidad de funciones desarrolladas para las redes neuronales y los algoritmos de *Machine Learning* en general.

A nivel de frameworks para *Deep Learning*, existen dos motores principales a destacar: Tensorflow y Theano. Tensorflow tiene detrás el potencial de Google, mientras que Theano está desarrollado por algunos de los investigadores más importantes en el área del aprendizaje profundo. Se ha elegido Tensorflow para este trabajo porque está en pleno crecimiento, siendo utilizado cada vez más por los desarrolladores, además de que tener el respaldo de una gran compañía como Google hace que haya un buen soporte y una mejora constante de las librerías[4].

Para la implementación del auto-codificador se ha empleado la API de Keras, que está construida sobre Tensorflow (también se puede usar sobre Theano). En los siguientes sub-apartados se detalla más información acerca de esto.

3.2.1 Tensorflow

Es una librería de software libre desarrollada por Google para realizar cálculos relativos al aprendizaje automático y en las redes neuronales profundas. Su funcionamiento se basa en realizar operaciones mediante diagramas de flujos de datos, es decir, se tienen una serie de nodos (operaciones matemáticas) que se comunican mediante tensores (matrices o conjunto de datos multidimensionales).

[4] Durante el desarrollo de este proyecto apareció también en versión estable PyTorch, una librería open-source para Python enfocada en Deep Learning desarrollada por Facebook.

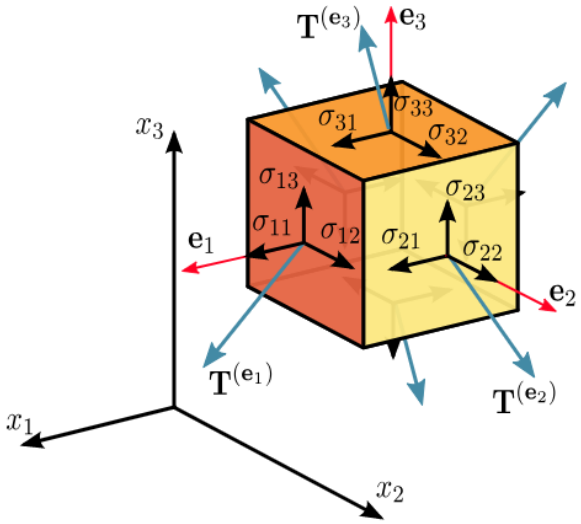

Figura 3-5. Representación gráfica de un tensor[5] (Fuente [Wikipedia])

Tensorflow tiene una arquitectura muy flexible, permitiendo un desarrollo multiplataforma (Python, C y Java), así como a nivel de rendimiento, pudiendo funcionar en un sistema con una o varias CPUs, GPUs o TPUs, entre otras plataformas hardware.

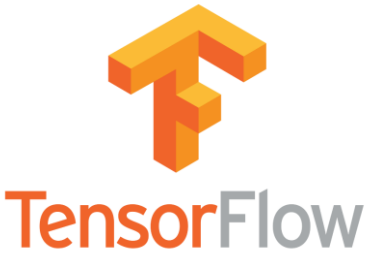

Figura 3-6. Logo de TensorFlow

Aunque se excede al alcance de este trabajo, es interesante destacar el término TPU, que se refiere a una unidad de procesamiento tensorial desarrollada por Google para ser utilizada por Tensorflow. Estas unidades están diseñadas para un gran volumen de cálculo como el necesario para entrenar complejos algoritmos de aprendizaje profundo.

Como se ha comentado, en nuestro caso sólo se empleará como una capa de nivel inferior a la API de Keras, funcionando como un motor. Es por ello que no se va a entrar en términos de desarrollo directo sobre Tensorflow.

3.2.1.1 TensorBoard

Los cálculos que se realizan durante un entrenamiento de una red neuronal con una gran cantidad de datos de entrada pueden generar resultados complejos y difíciles de visualizar.

Es por ello que aparece TensorBoard, una extensión de Tensorflow que incorpora una serie de herramientas de visualización de los datos. Entre ellas, la posibilidad de graficar diferentes métricas con los datos del entrenamiento y de la validación, histogramas, y realizar grafos de la red neuronal. Los grafos se pueden visualizar cuando se implemente la red directamente sobre Tensorflow, que no es nuestro caso, en el que emplearemos Keras para ello.

Para activar TensorBoard, es necesario incluir un *callback* en la función de entrenamiento del código de la red

[5] Para mayor información del fundamento matemático de un tensor empleado en Tensorflow: Cálculo Tensorial (Wikipedia)

neuronal. Este fragmento que se muestra a continuación es un ejemplo obtenido de una de las pruebas posteriores que se han realizado para el auto-codificador con Keras:

```
 h = autoencoder.fit(x_train, x_train,
                 epochs=20,
                 batch_size=128,
                 shuffle=True,
                 validation_data=(x_test, x_test),
                 callbacks=[TensorBoard(log_dir='/tmp/tb',  histogram_freq=0,
write_graph=False)])
```

Además, hay que activar TensorBoard desde una consola de comandos de la siguiente forma (se comporta como un servidor web local):

```
 #tensorboard --logdir=/tmp/tb
```

El acceso a la herramienta es mediante navegador web, y como en el ejemplo se pueden visualizar las gráficas de las métricas (pérdidas y "*accuracy*") que se han recogido.

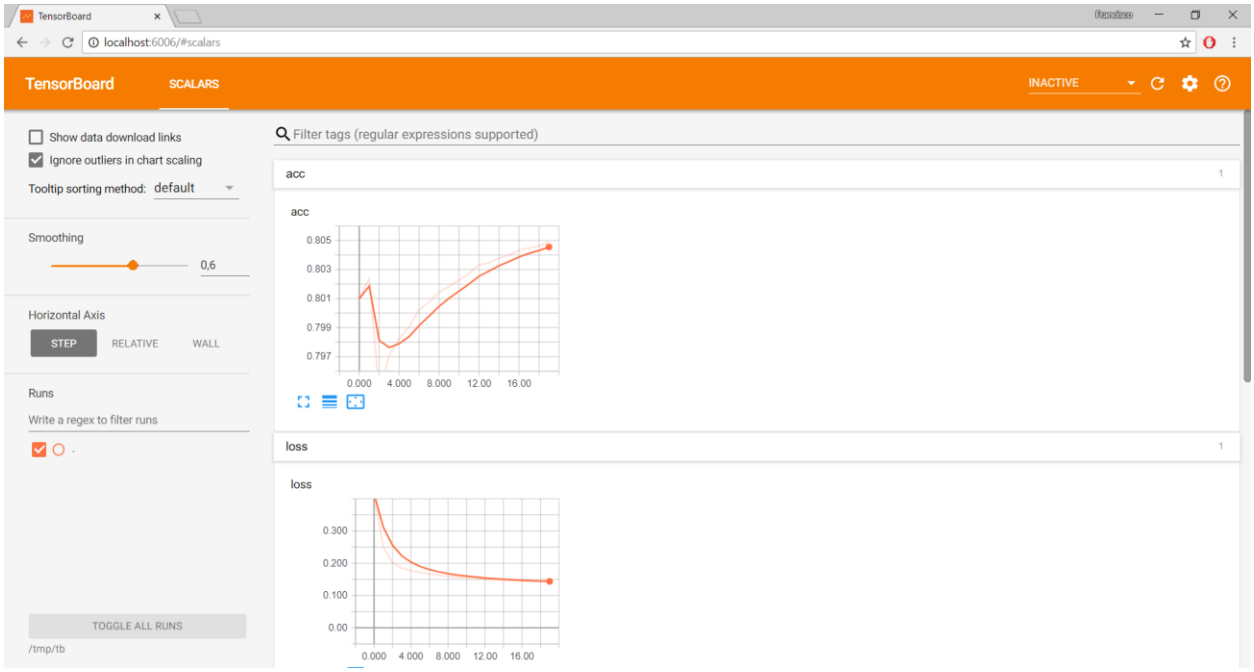

Figura 3-7. Ejemplo de uso de TensorBoard

3.2.2 Keras

Se trata de una API de alto nivel para redes neuronales escrita en Python y capaz de correr sobre TensorFlow, Microsoft Cognitive Toolkit o Theano.

Está diseñada para ofrecer una experiencia sencilla y rápida con las redes neuronales, muy a alto nivel y sin necesidad de profundizar en los entresijos de Tensorflow y el funcionamiento de sus tensores. Ha sido desarrollada por parte del equipo de investigación del proyecto ONEIROS (*Open-ended Neuro-Electronic Intelligent Robot Operating System*) y su principal autor es François Chollet.

Figura 3-8. Logo de Keras

Entre las principales ventajas de utilizar Keras para el diseño de redes neuronales se encuentra su fácil uso, ya que está diseñada para ser empleada por personas y no por máquinas; y su modularidad, ya que permite visualizar muy bien la estructura de capas de la red que se está diseñando, así como la gran cantidad de módulos y restricciones que se pueden añadir como funciones de activación, funciones de pérdidas, optimizadores, regularizadores, etc.

Además permite que pueda ser extendida con nuevos módulos, facilitando su creación o su integración con otras herramientas de Python.

La estructura de datos principal de Keras es un modelo, que es la forma de organizar las capas de la red neuronal. En el siguiente ejemplo se muestra la forma más sencilla de crear un modelo secuencial (*Sequential*) que forma una pila lineal de capas. Este ejemplo está sacado directamente de la documentación de Keras.

Ejemplo 3–1. *Creación de un modelo secuencial básico en Keras*

Se importa el modelo:

```
from keras.models import Sequential
model = Sequential()
```

Se añaden las capas a dicho modelo:

```
from keras.layers import Dense

model.add(Dense(units=64, activation='relu', input_dim=100))
model.add(Dense(units=10, activation='softmax'))
```

Se compila y se entrena:

```
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5, batch_size=32)
```

Se generan predicciones sobre datos nuevos:

```
classes = model.predict(x_test, batch_size=128)
```

Nótese que cuando se implemente el auto-codificador en capítulos posteriores, será ligeramente diferente la forma de añadir las capas. En lugar de hacerlo con la función `.add()` se realizará llamando directamente a las capas en la creación del modelo.

3.3 Software para procesado de imágenes

El tratamiento y procesado de imágenes resulta un punto fundamental en este proyecto. Para la aplicación en imágenes de Rayos X, estas tienen un tamaño muy grande que hacen inviable que puedan introducirse en la red neuronal (problemas de recursos en el equipo, etc.). Es por ello que se hace necesario emplear ciertas herramientas para diversas acciones sobre las imágenes: leerlas, guardarlas, recortarlas o aplicar previamente un preprocesado.

A continuación se resume el software empleado para dichas acciones para la realización del trabajo.

3.3.1 Scikit-image

Se trata de una librería *Open Source* de Python para el procesado de imágenes. Incluye entre otros algoritmos de segmentación, transformaciones, manipulación del color, análisis y filtrado, etc.

Scikit-image se puede entender como un toolkit de la librería SciPy, un ecosistema de paquetes para Python empleados para matemáticas, ciencias e ingeniería. Entre dichos paquetes de SciPy se encuentran NumPy (para matrices multidimensionales) o Matplotlib (para realizar gráficas), también empleados en este proyecto.

En su uso para el auto-codificador, es necesario leer un conjunto de imágenes, y posteriormente guardar las salidas en un formato de imagen determinado. Estas acciones pueden realizarse con dicha librería o directamente con Matplotlib. Además, scikit-image se ha empleado durante este proyecto para realizar un "resize" a las imágenes ya recortadas previo a introducirlas en el auto-codificador. Esto finalmente se ha podido solucionar y no ha sido necesario en la solución final, pero sí que ha formado parte de las pruebas realizadas durante toda la ejecución.

3.3.2 PIL (Python Imaging Library)

Esta librería de Python, al igual que en el caso de Scikit-image, es una librería de código libre para la apertura, manipulación y salvado de imágenes en diferentes formatos.

En este proyecto se ha utilizado PIL principalmente por su versatilidad a la hora de guardar las imágenes, ya que permite indicar propiedades como la resolución deseada para un formato determinado, lo que otras librerías no admiten.

3.3.3 Matlab

Es el software por excelencia empleado para todo tipo de cálculo matemático. Sus últimas versiones hasta incluyen herramientas para el tratamiento de algoritmos de aprendizaje máquina.

Sin embargo, ha sido por su potencial para el tratamiento de imágenes por lo que se ha seleccionado para realizar el preprocesado de las imágenes de Rayos X, el recorte en fragmentos de 200x200 así como la posterior lectura de la imagen completa reconstruida tras pasar por el auto-codificador.

Otro de los motivos por lo que se ha elegido es por el manejo y soltura que se tiene con este sofware tras las prácticas en el grado y máster, lo que ha facilitado la realización de esta parte más de tratamiento de imágenes que se excede un poco al alcance del proyecto que está centrado en el aprendizaje profundo. La versión empleada para la realización del proyecto ha sido la R2015a.

3.3.4 Aracne

Se trata de un software para el análisis y conteo de hilos de imágenes de Rayos X de lienzos, diseñado por el Grupo de Aprendizaje, Procesado de señal y Comunicaciones dentro del Departamento de Teoría de la Señal y Comunicaciones de la Universidad de Sevilla. Sus autores son: Juan José Murillo Fuentes, Irene Fondón García, Marta Ternero Gutiérrez, Lucía Córdoba-Saborido y Pablo Aguilera Bonet.

Entre los prerrequisitos para ejecutar Aracne se necesita tener previamente instalado *MATLAB® Compiler Runtime MCR V9.4*. En el siguiente enlace puede encontrarse un manual de uso de Aracne con toda la información acerca de la instalación y ejecución del mismo: http://grupo.us.es/gapsc/aracne/manual/.

Esta herramienta será utilizada para comparar los resultados de las imágenes a la salida del auto-codificador, comparándolo con la imagen original y con el procesado con el algoritmo de PCA. El software realiza un conteo de los hilos verticales y horizontales y muestra una serie de gráficos como mapas de colores e histogramas, así como estadísticas útiles para el análisis de las soluciones.

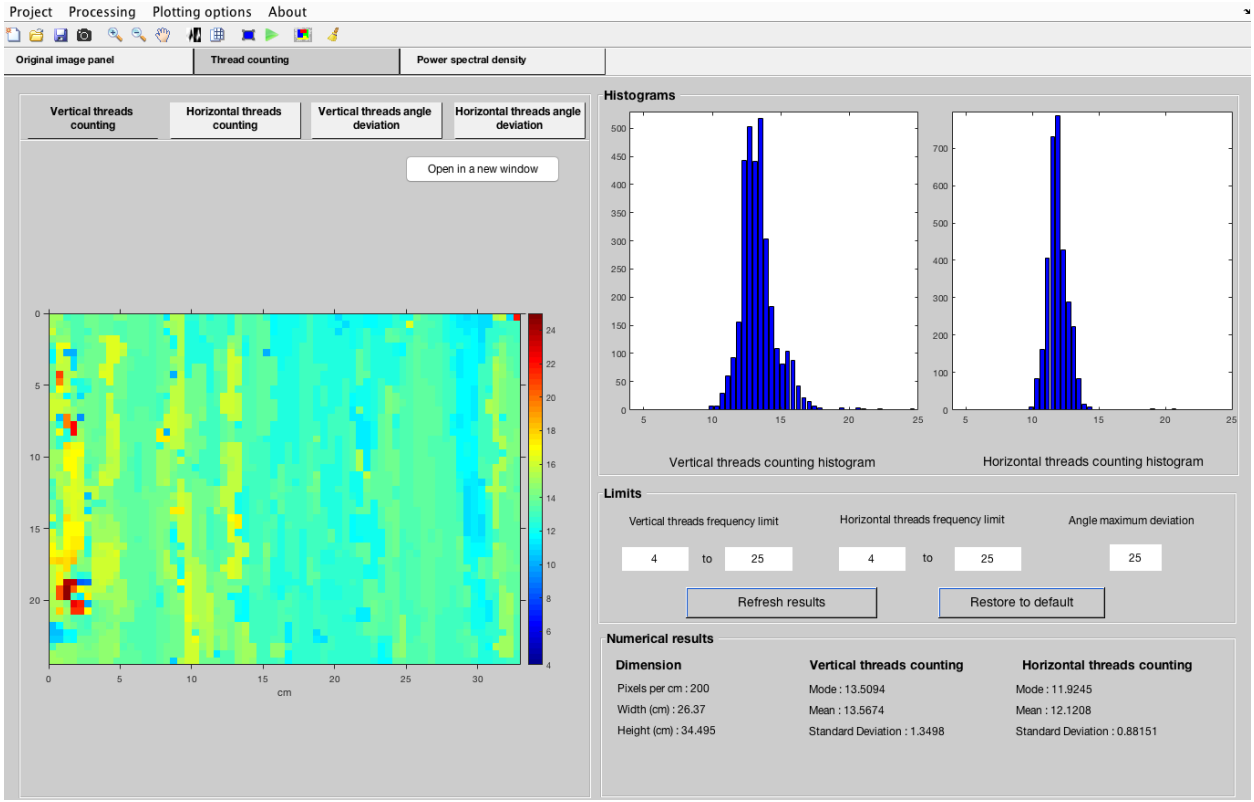

Figura 3-9. Ejemplo de resultados del software Aracne (obtenido de su web [6])

4 CONFIGURACIÓN Y PRUEBAS INICIALES

Intenta aprender algo sobre todo y todo sobre algo.

- Thomas Huxley -

A partir de este capítulo y los siguientes, se va a exponer el diseño y la implementación de un auto-codificador con el objetivo puesto en la aplicación de éste sobre las imágenes de Rayos X de cuadros, empleando las herramientas vistas anteriormente.

En concreto, en este capítulo se va a exponer la configuración básica del equipo para poder ejecutar el código desarrollado posteriormente, y se comenzará con un diseño básico de un autoencoder aplicado a una base de datos de imágenes de dígitos manuscritos.

4.1 Configuración del sistema

En esta sección se va a exponer la configuración del equipo que se ha necesitado para la realización del trabajo. Las características del sistema donde se ha desarrollado el proyecto son las siguientes:

Tabla 4–1. Características de sistema donde se ha desarrollado el proyecto

Elemento	Valor
Sistema Operativo	Microsoft Windows 10 Pro
Tipo de sistema	x64
CPU	Intel Core i7 hasta 3.6 GHz
RAM	8 GB
GPU	NVIDIA GeForce 930M

En primer lugar, para el lenguaje Python es necesario un entorno de desarrollo. Para el caso de Windows, se ha empleado la distribución Anaconda, una plataforma para Python que incluye numerosos módulos y proporciona una forma sencilla de administrar la instalación de paquetes o librerías ("*conda*")[6].

Anaconda ya incluye muchos paquetes básicos de Python como *NumPy*, *SciPy*, *Matplotlib* o *scikit-learn* entre otros. Sin embargo, será necesario instalar los paquetes de Keras y Tensorflow. Para ello se abrirá una consola

[6] Puede descargarse el software Anaconda directamente desde este enlace: https://www.anaconda.com/download/

de comandos del propio entorno (*Anaconda Prompt*) y se ejecutará lo siguiente:

```
#conda install -c conda-forge keras
#conda install tensorflow
```

Sin embargo, el entrenamiento de una red neuronal con cierta complejidad ejecutando Tensorflow por defecto realizando los cálculos con la CPU puede dar lugar a ejecuciones con una lentitud extrema, incluso llegando a bloquear el equipo.

Para evitar esto, se decide cambiar la configuración para que se ejecute en la GPU (unidad de procesamiento de gráficos), de forma que se acelere el funcionamiento de aplicaciones dedicadas al deep learning, análisis y otros algoritmos de ingeniería. La idea es que se asigne a la GPU las partes donde la computación es más intensa, mientras que el resto del código se desarrolla en la CPU.

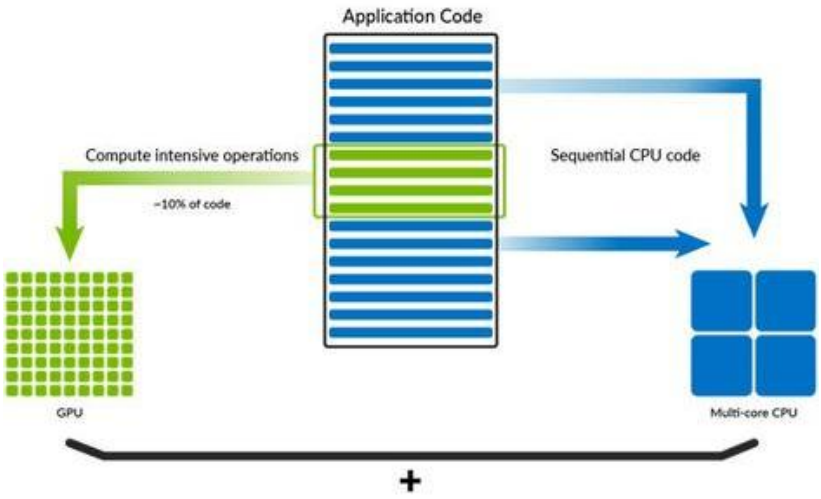

Figura 4-1. Funcionamiento de aceleración por GPU (Fuente: zdnet.com)

Para ejecutar Tensorflow mediante GPU (en este caso de NVIDIA), será necesario instalar en el equipo una serie de drivers (si no los trae ya instalados por defecto):

- CUDA Toolkit: Entorno de desarrollo para la creación de aplicaciones aceleradas por GPU.
- cuDNN: Librería para redes neuronales aceleradas por GPU.

Tras configurar el equipo para poder ejecutar la aceleración por GPU, será necesario reinstalar Tensorflow, en este caso la versión dedicada para GPU. Si no se desea eliminar la versión anterior, se puede crear un nuevo entorno en Anaconda, donde se indique la versión de python con la que se desea trabajar:

```
#conda create -n tensorflow-gpu python=3.5.2
#activate tensorflow-gpu
```

Una vez nos encontramos en el nuevo entorno, se instalará la versión de Tensorflow dedicada a GPU (se debe recordar que al estar en un nuevo entorno será necesario instalar en este también la librería de Keras):

```
#conda install -c conda-forge tensorflow-gpu
```

Se puede comprobar que se está usando la GPU en Tensorflow tras la instalación de la siguiente forma:

```
#python
#import tensorflow as tf
#sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
```

Con esta configuración ya es posible la ejecución de los algoritmos de *deep learning* empleados a lo largo de este proyecto. Si se desea además utilizar la herramienta TensorBoard, será necesario realizar la siguiente instalación (de nuevo desde la consola de comandos de Anaconda):

```
#conda install -c conda-forge tensorboard
```

Es posible que tras la instalación se obtenga un error de este estilo al intentar ejecutar Tensorboard

```
  serialized_pb=_b('\n+tensorboard/plugins/audio/plugin_data.proto\x12\x0btens
orboard\"}\n\x0f\x41udioPluginData\x12\x0f\n\x07version\x18\x01
\x01(\x05\x12\x37\n\x08\x65ncoding\x18\x02
\x01(\x0e\x32%.tensorboard.AudioPluginData.Encoding\"
\n\x08\x45ncoding\x12\x0b\n\x07UNKNOWN\x10\x00\x12\x07\n\x03WAV\x10\x0b\x62\x
06proto3')
  TypeError: __init__() got an unexpected keyword argument
'serialized_options'
```

El problema que se da con este error es que la versión de TensorBoard es diferente a la instalada previamente de Tensorflow. Para solucionar el problema, pasa por actualizar el que tenga una versión inferior. En este caso, la versión más antigua era la de Tensorflow, por lo que para actualizarlo:

```
#conda upgrade -c conda-forge tensorflow
```

Nótese que esta actualización es para la versión normal de Tensorflow, para la versión que emplea la GPU habrá que actualizar el paquete correspondiente. Tras solucionar este problema, se podrá ejecutar Tensorboard que como ya se comentó anteriormente funciona como un servidor web local.

```
#tensorboard --logdir=/tmp/tb
TensorBoard 1.9.0 at http://DESKTOP-M1SPAAK:6006 (Press CTRL+C to quit)
```

4.2 Pruebas iniciales con MNIST

Las primeras implementaciones realizadas parten de un auto-codificador muy sencillo, que se va a ir haciendo cada vez más complejo hasta llegar al objetivo deseado. Para estas primeras pruebas, se va a utilizar la base de datos (*dataset*) MNIST, que ofrece imágenes de dígitos manuscritos para emplearlos en algoritmos de *Deep Learning*, con un total de 60000 imágenes de entrenamiento y 10000 para testeo.

Antes que nada, será necesario importar las librerías que vamos a emplear: en principio las capas básicas para construir el auto-codificador con Keras, así como el modelo para su posterior entrenamiento.

```
from keras.layers import Input, Dense
from keras.models import Model
import numpy as np
```

La carga de dicho *dataset* en Keras es muy sencilla, se puede tener disponible para su uso con tan solo un par de líneas de código:

```
from keras.datasets import mnist
(x_train, _), (x_test, _) = mnist.load_data()
```

MNIST incluye las imágenes y sus respectivas etiquetas para poder emplearlo en algoritmos de clasificación. Como ya se ha comentado, un auto-codificador es un modelo de aprendizaje no supervisado, por lo que no requiere de las etiquetas para su funcionamiento. Es por ello que durante la carga de los datos se descartan dichas "labels". A continuación se van a ir detallando algunas de las pruebas realizadas con esta base de datos.

4.2.1 Autoencoder no convolucional

El primer auto-codificador desarrollado sigue la estructura de un "undercomplete autoencoder", siendo la capa de entrada el conjunto de la imagen del dígito manuscrito, y la salida este mismo conjunto tras la decodificación. Se tiene una capa oculta que es la encargada de "codificar".

En este caso, al ser una red neuronal básica (no convolucional), los datos de entrada no pueden tener varias dimensiones. Esto significa que se hace necesario "aplanar" las imágenes de 28x28 a vectores de tamaño 784.

```
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))
```

4.2.1.1 Autoencoder básico

A continuación se van a resumir algunas de las líneas de código principales para este auto-codificador, de forma que se sienten las bases para el resto. Básicamente, se definen las tres capas, que en este caso serán densas, es decir, completamente conectadas. La capa oculta convertirá la entrada en un vector de dimensión 32. Posteriormente construiremos el modelo a partir de las capas definidas.

```
# Tamaño de la representación codificada
encoding_dim = 32

# Crea espacio para la entrada
input_img = Input(shape=(784,))
# Capa "encoded": es la representacion codificada de la entrada
encoded = Dense(encoding_dim, activation='relu')(input_img)
# Capa "decoded": es la reconstruccion de la entrada
decoded = Dense(784, activation='sigmoid')(encoded)

# Creacion del modelo, que mapea una entrada a su reconstruccion
autoencoder = Model(input_img, decoded)
```

Tras definir el modelo, el siguiente paso será compilarlo y entrenarlo. En este caso se empleará una función de pérdidas de entropía cruzada binaria, que ya se expuso en el apartado 2.2.1.1.

```
  autoencoder.summary()

  autoencoder.compile(optimizer='adadelta',loss='binary_crossentropy',
metrics=["accuracy"])

  h = autoencoder.fit(x_train, x_train,
                  epochs=20,
                  batch_size=256,
                  shuffle=True,
                  validation_data=(x_test, x_test))
```

La función *summary()* te muestra un resumen de la red que se ha diseñado. Esto viene muy bien para tener claros los parámetros y las dimensiones de la red, sobre todo cuando se vuelve más compleja.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         (None, 784)               0
_________________________________________________________________
dense_1 (Dense)              (None, 32)                25120
_________________________________________________________________
dense_2 (Dense)              (None, 784)               25872
=================================================================
Total params: 50,992
Trainable params: 50,992
Non-trainable params: 0
_________________________________________________________________
```

Una vez finalizado el entrenamiento de la red neuronal, podremos obtener diversos tipos de resultados. El más importante es la salida del autoencoder para ciertas imágenes de validación. Esto se consigue con la función *predict()* sobre el propio modelo*:*

```
  decoded_imgs = autoencoder.predict(x_test)
```

Con esto podemos posteriormente representar la salida de la red neuronal y comparar los resultados con la entrada. En este caso, la salida obtenida con este autoencoder es la siguiente:

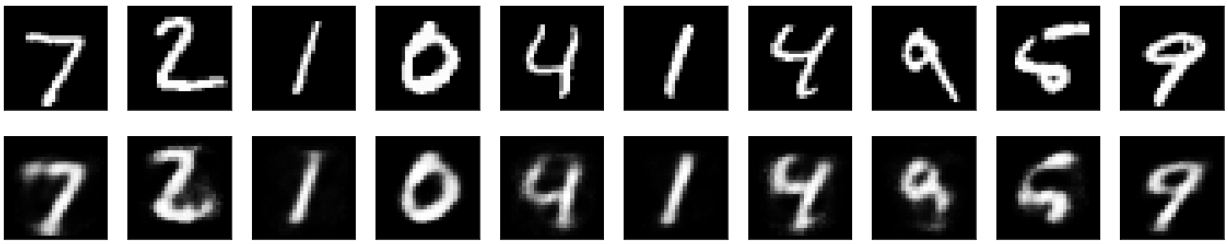

Figura 4-2. Salida de auto-codificador básico (no convolucional) con MNIST

Existen una serie de estadísticas que pueden obtenerse para ver cómo se ha desarrollado el entrenamiento del modelo. Las más interesantes son la exactitud (“accuracy”) del modelo, y la minimización de la función de

pérdidas durante el entrenamiento. Para poder obtener dicha información, debe indicarse como parámetro a la hora de compilar el auto-codificador (`metrics=["accuracy"]`).

La función *history()* actúa como un "callback" para obtener diferentes valores internos del modelo durante el entrenamiento. Entre las propiedades de esta función se encuentra la propiedad de "log" que incluye los valores de exactitud y pérdidas durante el entrenamiento y la validación[7].

```
acc = h.history["acc"]
loss = h.history["loss"]
acc_val = h.history["val_acc"]
loss_val = h.history["val_loss"]
```

El valor de "accuracy" se obtiene como la media de los diferentes valores obtenidos en los "batchs" anteriores, realizando una comparación entre la salida predicha y la salida real (en el caso del autoencoder sería el mismo valor de entrada). Keras emplea la siguiente fórmula en su código a nivel de Tensorflow:

```
K.mean(K.equal(y_true, K.round(y_pred)))
```

Para el valor de "loss" ocurre lo mismo que en el caso anterior, calculando el valor de la función de pérdidas minimizada a través de la media de los "batchs" anteriores. En nuestro caso, como ya se ha comentado previamente, se ha utilizado la función de pérdidas de entropía cruzada binaria (ecuación *(2-6)*). A nivel de código de Keras, directamente llama a la función "categorical_crossentropy" de Tensorflow:

```
K.categorical_crossentropy(y_true, y_pred)
```

Estos valores pueden ser representados en gráficas para ver su evolución, tanto para los datos de entrenamiento como para los de validación:

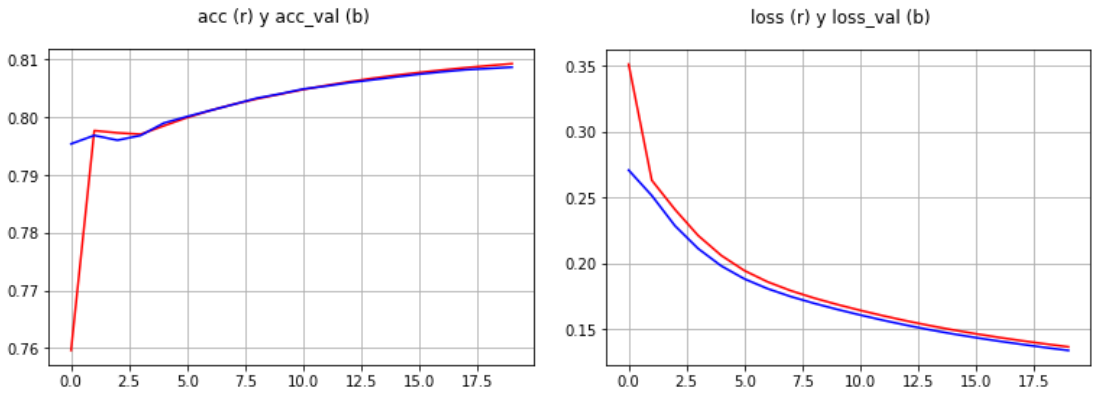

Figura 4-3. Gráficas de "accuracy" y "loss" para autoencoder básico

Como se ha visto en este caso, el uso de Keras facilita muchísimo la creación de la red neuronal, permitiendo que se obtengan resultados fácilmente sin necesidad de un código complejo. Además, es capaz de adaptarse a las necesidades y a la complejidad que se le quiera dar al modelo.

[7] Uso de callbacks en Keras: https://keras.io/callbacks/

4.2.1.2 Autoencoder disperso

La siguiente prueba realizada es ver qué pasa si a la configuración inicial le añadimos cierto valor de dispersión. Se va a aplicar una regularización de actividad sobre la capa oculta, del tipo L1 (esto se expuso en el apartado 2.2.1.2.1), en el que el valor "10e-7" se corresponde con el valor de λ a utilizar. Este valor se ha tomado como base para posteriormente encontrar cuál sería el que mejor resultado obtendría sobre la red.

```
encoded = Dense(encoding_dim, activation='relu',
                activity_regularizer=regularizers.l1(10e-7))(input_img)
```

Para encontrar el valor óptimo de este parámetro se ha realizado una validación cruzada sobre la red neuronal. Esto se ha realizado en primer lugar creando el modelo del autocodificador (al igual que antes), pero incluyéndolo en una función que posteriormente será llamada pasándole el valor correspondiente del "activity regularizer".

```
def crear_modelo( reg_value ):
      ...
      return autoencoder
```

La librería *scikit-learn* incluye la función *StratifiedKFold* que permite realizar una validación cruzada de forma automática. No obstante, no se ha podido emplear ya que está definida para un conjunto de datos planos, es decir, no puede tener dos dimensiones como es el caso de las imágenes.

Se podría haber probado realizando un "reshape" sobre los valores de las imágenes, pero finalmente se ha optado por realizar la validación cruzada de forma manual, empleando el 70% de los datos de entrenamiento para "*train*" y el 30% restante para los datos de validación cruzada (*cv*). Los pasos seguidos para su implementación son los siguientes:

- En un bucle en el que se harán las diferentes iteraciones con divisiones de los datos, se realiza esto de forma aleatoria (empleando función de "numpy"):

```
indices = np.random.permutation(m)
ind_tr=indices[:n_tr]
ind_cv=indices[n_tr:]

Xtr=x_train[ind_tr,:];
Xcv=x_train[ind_cv,:];
```

- Dentro de otro bucle for que recorre el vector de valores para el "activity regularizer" se crea en cada iteración el modelo con el valor en ese momento y se entrena el modelo.

```
model = crear_modelo(reg_value)
```

- A la salida del entrenamiento se obtiene el valor de "accuracy" de los datos de validación, que se almacena en una matriz.
- Al finalizar todas las iteraciones se nos queda una matriz en este caso de las siguientes dimensiones:
  - 4x5 → 4 iteraciones permutación de datos, 5 valores diferentes de "activity regularizer"
  - Por ejemplo, en una de las pruebas se obtiene lo siguiente:

```
array([
[ 0.80421117,  0.80056449,  0.80625   ,  0.8089273 ,  0.8089601 ],
[ 0.80411671,  0.80074355,  0.80569536,  0.80852438,  0.80848455],
[ 0.80505152,  0.79424667,  0.80685452,  0.80947853,  0.80937939],
[ 0.80417297,  0.80417092,  0.80670621,  0.80847591,  0.80850106]])
```

- Tras ello se calculan las medias de las iteraciones para cada uno de los valores del parámetro, obteniendo a la salida lo siguiente:

```
Valor de regularizer: 0.0001     Media accuracy: 80.4388092359%
Valor de regularizer: 1e-05      Media accuracy: 79.9931405597%
Valor de regularizer: 1e-06      Media accuracy: 80.6376522475%
Valor de regularizer: 1e-07      Media accuracy: 80.8851526909%
Valor de regularizer: 1e-08      Media accuracy: 80.8831278086%
```

Para el ejemplo expuesto (que es el mismo modelo que el autoencoder básico), se observa que tras la validación cruzada el que obtiene mejor "accuracy" es el 1e-07, aunque por muy poca diferencia con el resto.

Con esta prueba se ha querido tener claro el concepto de dispersión aplicado sobre una red neuronal tradicional, y ver cómo se puede realizar una validación cruzada de los datos para encontrar el mejor valor. En apartados posteriores en los que se diseñe un autoencoder convolucional se verá como la dispersión se regulará directamente con una capa adicional de *Dropout*, otro tipo de regularación de dispersión y cuyo hiperparámetro ya se expuso en el apartado 3.1.2.4.

4.2.2 Autoencoder convolucional

Ahora se realiza la prueba de introducir los datos en un auto-codificador convolucional. Para este caso, los datos de entrada tienen un tratamiento diferente, los valores no se aplanan como en el caso no convolucional, sino que se mantienen en 2D. Además, como la base de datos MNIST es en blanco y negro, tan solo se tendrá un canal en la capa de entrada (si la imagen por ejemplo es en color se podrá cambiar para tres canales).

La estructura de capas de un autoencoder convolucional ya se vio en la subsección 2.2.2, que aplicado para las funciones en Keras se resume en:

- Codificación: capas de *Conv2D* y *MaxPooling2D*
- Decodificación: capas de *Conv2D* y *UpSampling2D*

El ejemplo que se ha implementado en este caso es el representado en la figura:

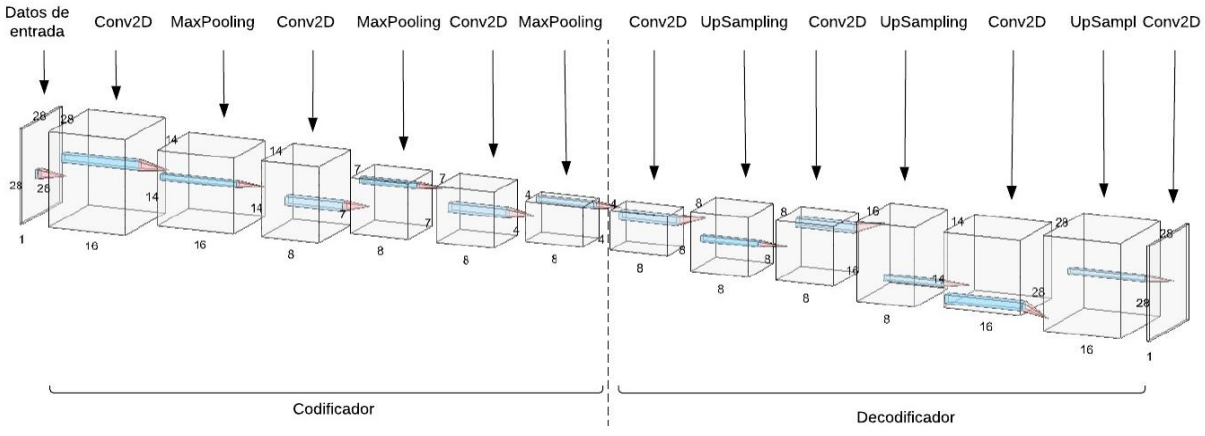

Figura 4-4. Arquitectura de capas autoencoder convolucional (versión 1)

Como puede observarse es una red neuronal convolucional profunda con numerosas de las capas indicadas. Este modelo será el que se vaya a ir modificando en función de las necesidades y los resultados.

El modelo de este autoencoder convolucional básico escrito en Keras es el siguiente:

```
# Capa de entrada
input_img = Input(shape=(28, 28, 1))

# Codificador
x = Conv2D(16, (3, 3), activation='relu', padding='same')(input_img)
capa1 = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(capa1)
capa2 = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(capa2)
encoded = MaxPooling2D((2, 2), padding='same')(x)

# Decodificador
x = Conv2D(8, (3, 3), activation='relu', padding='same')(encoded)
capa3 = UpSampling2D((2, 2))(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(capa3)
capa4 = UpSampling2D((2, 2))(x)
x = Conv2D(16, (3, 3), activation='relu')(capa4)
capa5 = UpSampling2D((2, 2))(x)

# Capa de salida
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(capa5)
```

El resto de parámetros a la hora de entrenar el modelo se han mantenido con respecto al caso no convolucional. También se han tomado todos los datos de MNIST, y el entrenamiento se ha efectuado con 20 “epochs”.

Como puede verse en el resumen de capas, a pesar de ser una estructura mucho mayor, el número de parámetros es muy inferior al necesario para el caso no convolucional, lo que hace que se mejore la velocidad de entrenamiento.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_5 (InputLayer)         (None, 28, 28, 1)         0
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 28, 28, 16)        160
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 14, 14, 16)        0
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 14, 14, 8)         1160
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 7, 7, 8)           0
_________________________________________________________________
conv2d_15 (Conv2D)           (None, 7, 7, 8)           584
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 4, 4, 8)           0
_________________________________________________________________
conv2d_16 (Conv2D)           (None, 4, 4, 8)           584
_________________________________________________________________
up_sampling2d_7 (UpSampling2 (None, 8, 8, 8)           0
_________________________________________________________________
conv2d_17 (Conv2D)           (None, 8, 8, 8)           584
_________________________________________________________________
up_sampling2d_8 (UpSampling2 (None, 16, 16, 8)         0
_________________________________________________________________
conv2d_18 (Conv2D)           (None, 14, 14, 16)        1168
_________________________________________________________________
up_sampling2d_9 (UpSampling2 (None, 28, 28, 16)        0
```

```
_________________________________________________________________
conv2d_19 (Conv2D)           (None, 28, 28, 1)         145
=================================================================
Total params: 4,385
Trainable params: 4,385
Non-trainable params: 0
_________________________________________________________________
```

4.2.2.1 Análisis de capas internas de auto-codificador convolucional

Para estudiar cómo se comporta el autoencoder a través de las capas ocultas se va a representar la salida de alguna de ellas. Cada una de las capas que se ha señalado de color rojo o azul se ha representado unos dígitos para ver su transformación. En el caso de las marcadas de color azul, son las salidas de la "codificación" y "decodificación"

Se ha tenido que estudiar cómo son las dimensiones de cada capa para poder representar dicha salida. Las capas intermedias seleccionadas se han tomado (a excepción de la "codificada" y "decodificada") tras la salida de realizar la convolución y el muestreo.

También se muestra en el siguiente análisis la representación del filtro de cada capa convolucional (pesos), para ver cómo evolucionan a la vez que los datos conforme se avanza en la red neuronal.

En las figuras se representan en la parte superior los pesos y en la parte inferior las salidas de cada una de las "capas".

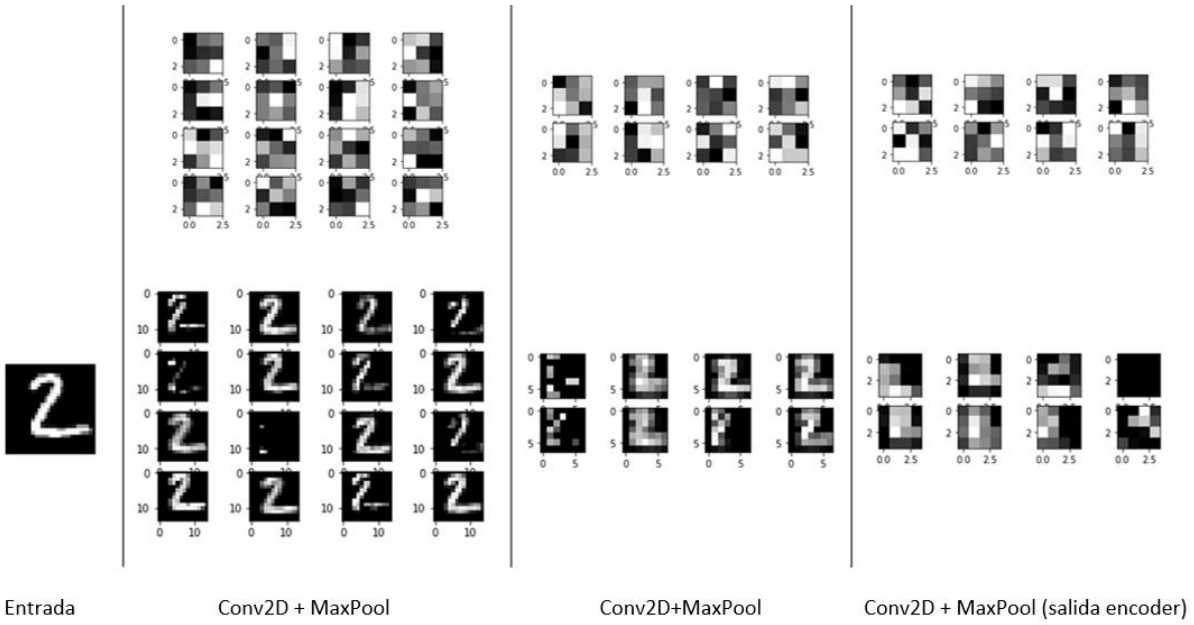

Figura 4-5. Análisis de capas de codificación en autoencoder convolucional

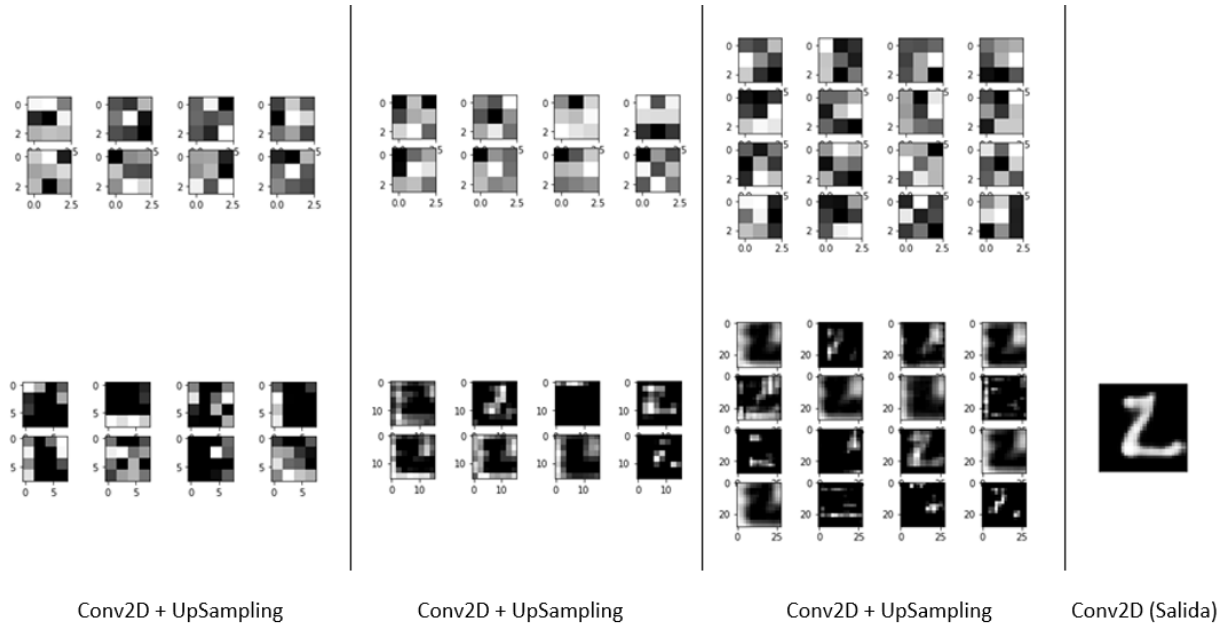

Figura 4-6. Análisis de capas de decodificación en autoencoder convolucional

Los resultados a la salida de este autoencoder, en principio parecen ser peores que en el caso no convolucional, pero se verá con las diferentes modificaciones realizadas que finalmente se mejorarán.

Los valores obtenidos de "accuracy" y pérdidas para este ejemplo son:

- Accuracy: 81.5683671951%
- Loss: 14.1179821253%

4.2.2.2 Añadiendo ruido al auto-codificador convolucional

Ahora nos preguntamos qué pasa si a los datos de entrada se le añade ruido. Esto es la idea del Denoising autoencoder (DAE), que pretendía evitar que aprendiera la matriz identidad a partir de añadir ruido.

Nuestro objetivo de hacer esta prueba es ver si realmente el modelo es robusto frente al ruido, ya que cuando se aplique a las imágenes de Rayos X tendrán bastante ruido que será necesario que se elimine.

El modelo del autoencoder se ha modificado con respecto al anterior, disminuyendo el número de capas. Es por ello que se hará también la misma prueba sin añadirle ruido a este modelo de autoencoder.

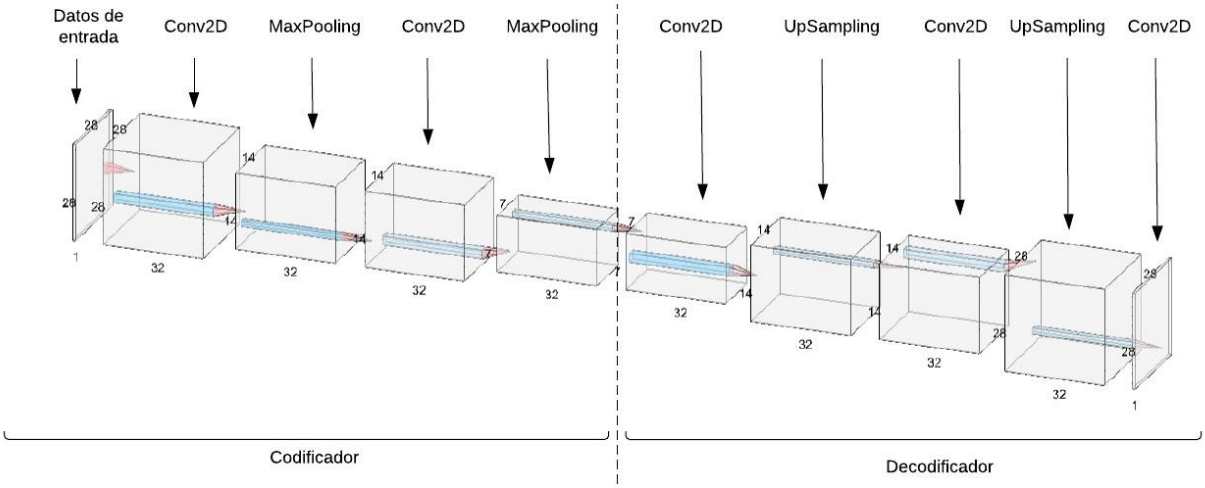

Figura 4-7. Arquitectura de capas autoencoder convolucional (versión 2)

Si se entrena el modelo con los datos de entrada sin introducir aún el ruido, se obtiene a la salida una "accuracy" sobre los datos de validación de un 82.40%.

```
Epoch 18/20
6000/6000 [==============================] - 3s 496us/step - loss: 0.0837 - acc: 0.8131 - val_loss: 0.0822 -
val_acc: 0.8239
Epoch 19/20
6000/6000 [==============================] - 3s 496us/step - loss: 0.0838 - acc: 0.8131 - val_loss: 0.0778 -
val_acc: 0.8248
Epoch 20/20
6000/6000 [==============================] - 3s 496us/step - loss: 0.0829 - acc: 0.8132 - val_loss: 0.0811 -
val_acc: 0.8240
```

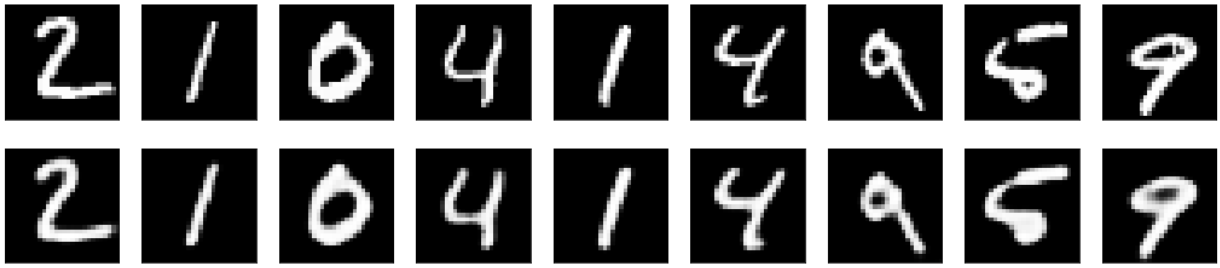

Figura 4-8. Resultados autoencoder convolucional (versión 2) sin ruido

Ahora se introduce a los datos de entrada un ruido gaussiano con un factor de un 0.7:

Figura 4-9. Datos de entrada modificados con un ruido gaussiano

```
  noise_factor = 0.7
  x_train_noisy = x_train + noise_factor * np.random.normal(loc=0.0,
scale=1.0, size=x_train.shape)
  x_test_noisy = x_test + noise_factor * np.random.normal(loc=0.0, scale=1.0,
size=x_test.shape)
```

```
x_train_noisy = np.clip(x_train_noisy, 0., 1.)
x_test_noisy = np.clip(x_test_noisy, 0., 1.)
```

Si se vuelve a entrenar el modelo con estos datos de entrada modificados, se obtiene una "accuracy" sobre los datos de validación de un 81.30%.

```
Epoch 18/20
6000/6000 [==============================] - 3s 500us/step - loss: 0.1437 - acc: 0.8045 - val_loss: 0.1426 -
val_acc: 0.8116
Epoch 19/20
6000/6000 [==============================] - 3s 499us/step - loss: 0.1433 - acc: 0.8046 - val_loss: 0.1394 -
val_acc: 0.8181
Epoch 20/20
6000/6000 [==============================] - 3s 499us/step - loss: 0.1421 - acc: 0.8049 - val_loss: 0.1398 -
val_acc: 0.8130
```

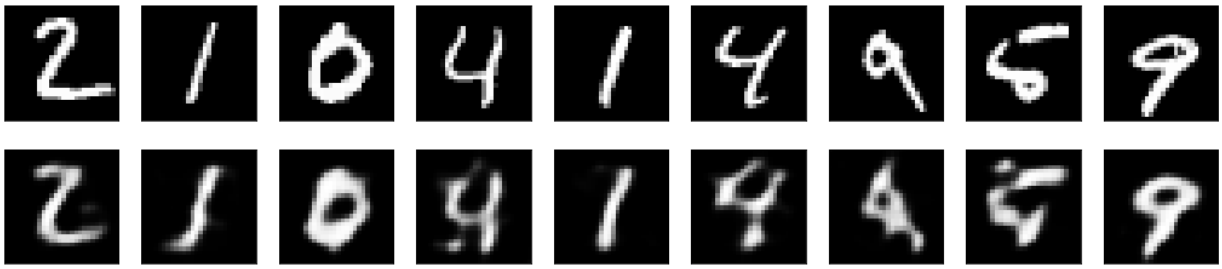

Figura 4-10. Resultados autoencoder convolucional (versión 2) con ruido

Se observa que cuando se introduce ruido la salida tiene un valor de exactitud inferior al caso de si no se incluye el ruido. Esto se puede observar también en la representación de los dígitos efectuada, que la reconstrucción en el caso de tener ruido es más borrosa.

Entonces uno se puede preguntar por qué se le añade ruido si realmente los resultados no mejoran. Es una forma de comprobar que el autoencoder funciona de forma muy correcta para eliminar el ruido (robustez frente a ruido).

Para el caso SIN ruido, al tener un mayor número de neuronas que las dimensiones de la entrada, realmente el autoencoder hace poco: solo se encarga de copiar, no se queda con la información relevante. El autoencoder se comporta como una matriz identidad.

El hecho de tener ruido le obliga a extraer la información importante. Realmente es un método de regularización al igual que la dispersión para evitar este problema de copia directa.

4.2.2.3 Decisiones sobre el modelo convolucional

Como ya se comentó en la sección 3.1 existen una serie de parámetros que nos permiten tomar decisiones acerca del diseño del autoencoder. Con las pruebas realizadas ya se han tenido en cuenta algunos de ellos, como por ejemplo las funciones de activación a emplear en la red (ReLU y sigmoide) así como el *padding* para cuadrar las salidas de las capas y el *stride* (que está puesto por defecto a 1).

Este apartado se va a centrar en las pruebas sobre otros dos conceptos: Batch Normalization y Dropout. Para ello se va a emplear el autoencoder convolucional versión 1 que es la primera prueba de red neuronal convolucional realizada.

Con este modelo, si volvemos a entrenar en este caso con 50 "epochs", los resultados son los siguientes (incluyendo valores de "accuracy" y "loss"):

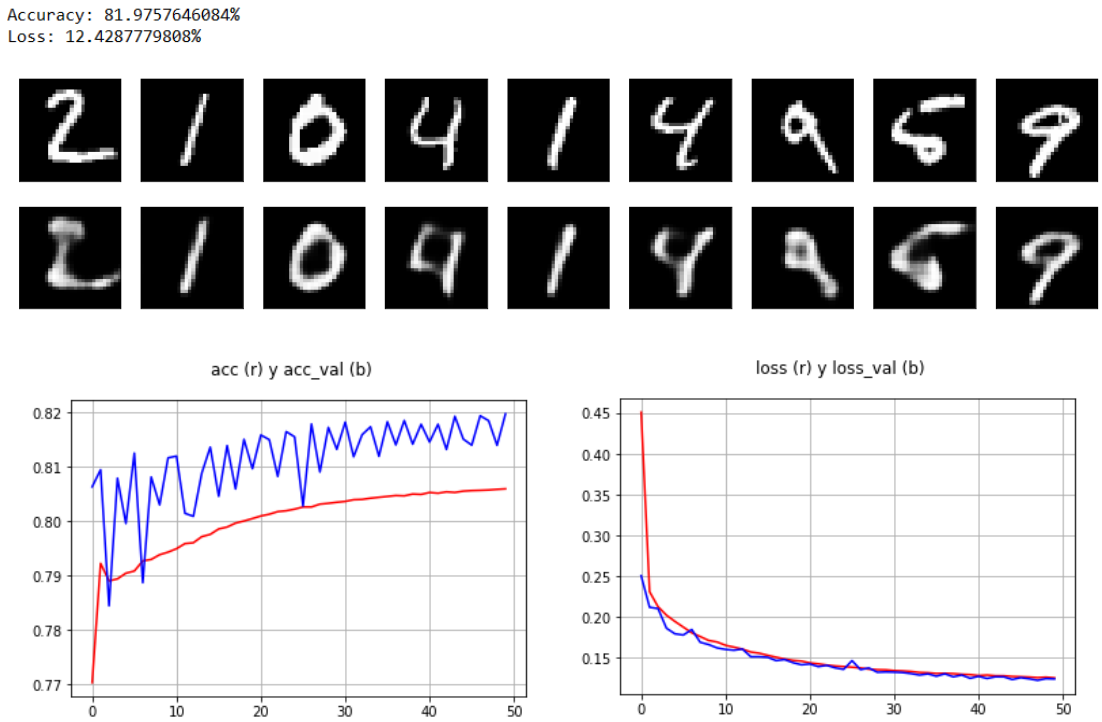

Figura 4-11. Resultados autoencoder convolucional (versión 1)

Para el caso de Batch Normalization, lo que se hace es desglosar la capa de convolución poniendo aparte la función de activación. Entre estas dos, se introduce la nueva capa de batch normalization. Esto se va a realizar para cada capa de convolución interna.

Un ejemplo de cómo quedaría en la estructura de capas es el siguiente (para simplificar en la memoria no se vuelve a copiar toda la estructura completa):

```
Layer (type)                 Output Shape              Param #
=================================================================
input_9 (InputLayer)         (None, 28, 28, 1)         0
_________________________________________________________________
conv2d_51 (Conv2D)           (None, 28, 28, 16)        160
_________________________________________________________________
batch_normalization_15 (Batc (None, 28, 28, 16)        64
_________________________________________________________________
activation_15 (Activation)   (None, 28, 28, 16)        0
_________________________________________________________________
max pooling2d 22 (MaxPooling (None, 14, 14, 16)        0
...
```

La salida obtenida tras entrenar de nuevo el modelo con las capas de Batch Normalization es la que se muestra a continuación, en la que se observa que el resultado del "accuracy" para los datos de validación es mucho más regular que en el caso no normalizado.

Además se observa a simple vista en los dígitos representados que la salida se visualiza mejor que en el caso anterior.

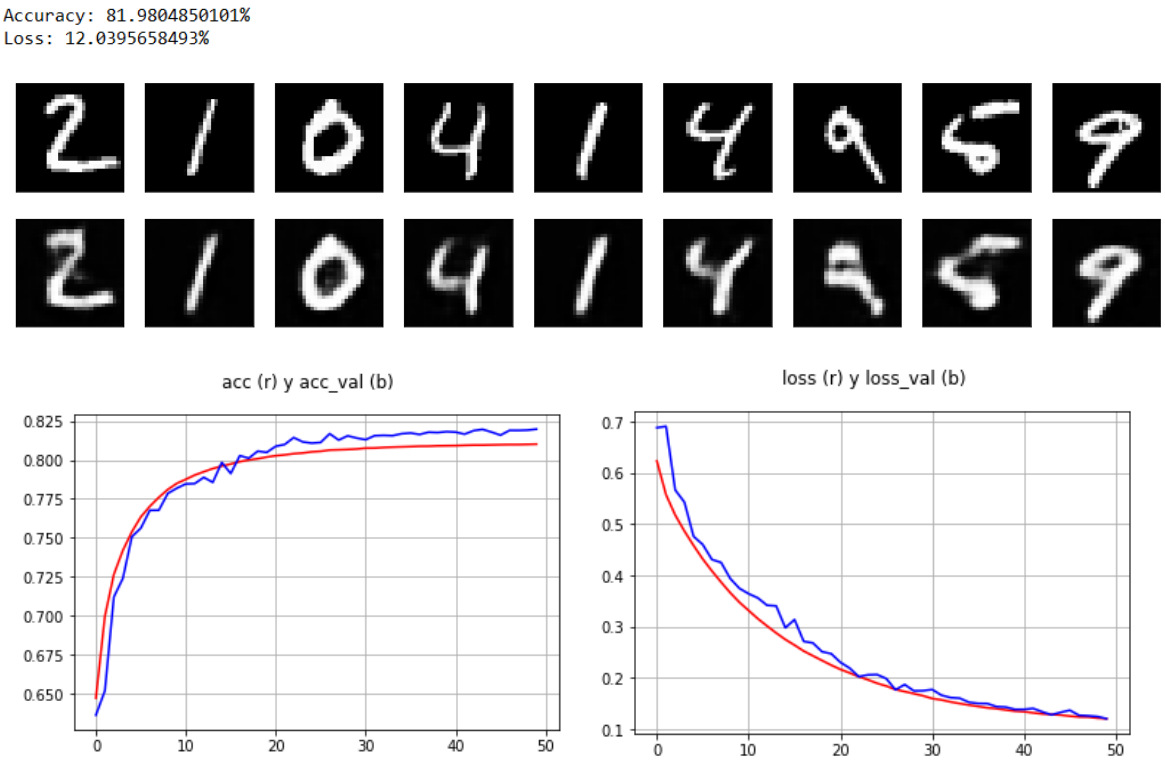

Figura 4-12. Resultados autoencoder convolucional (versión 1) con Batch Normalization

Se introducen ahora las capas de Dropout de forma independiente (el mismo modelo sin tener en cuenta Batch Normalization). Este Dropout puede realizarse de dos formas, agregándolo solo en tras la capa de entrada o incluyéndolo también sobre las capas internas al igual que se ha realizado con el Batch Normalization.

En el ejemplo se va a añadir un dropout del 20%, es decir, 1 de cada 5 neuronas se desactivará aleatoriamente en cada actualización.

La primera prueba realizada consiste en introducir esta regularización directamente sobre la entrada:

```
input_img = Input(shape=(28, 28, 1))
x = Dropout(0.2)(input_img)
x = Conv2D(16, (3, 3), activation='relu', padding='same')(x)
```

La segunda prueba consiste en incluir el Dropout (DP) también en las capas internas al igual que se hacía con Batch Normalization (en este caso no es necesario separar la función de activación).

```
conv2d_72 (Conv2D)           (None, 28, 28, 16)        160
_________________________________________________________________
dropout_3 (Dropout)          (None, 28, 28, 16)        0
_________________________________________________________________
max_pooling2d_31 (MaxPooling (None, 14, 14, 16)        0
...
```

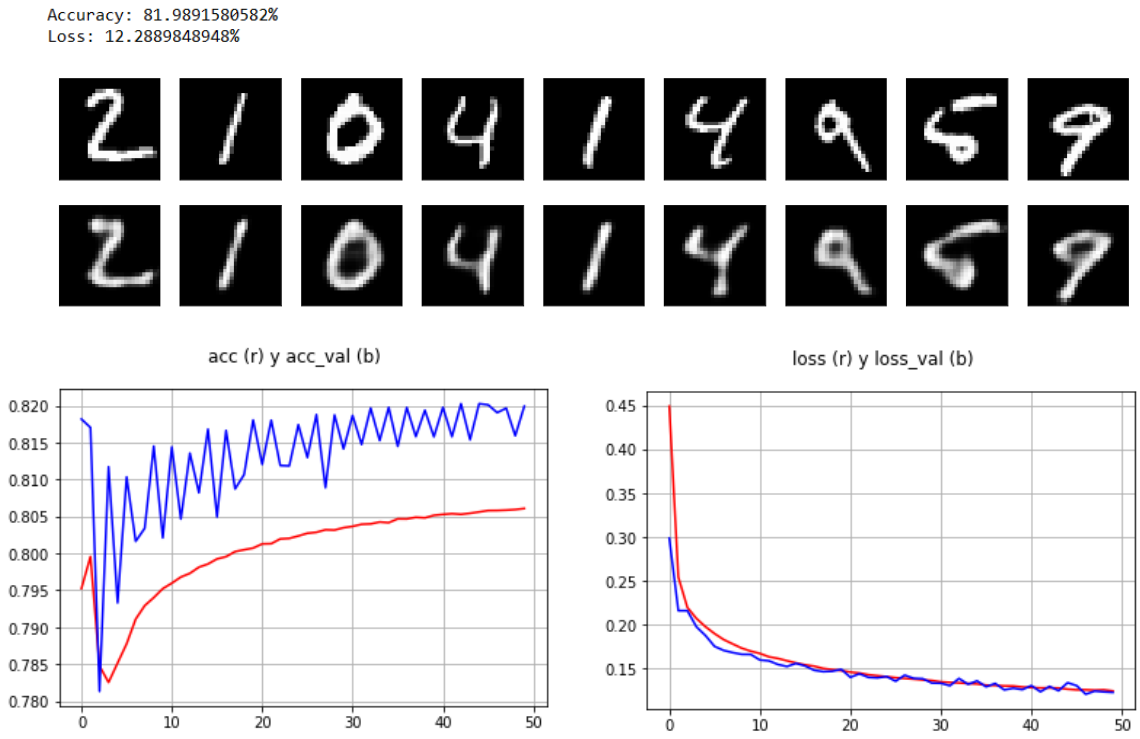

Figura 4-13. Prueba AE convolucional (versión 1) con DP=0.2 solo en la entrada

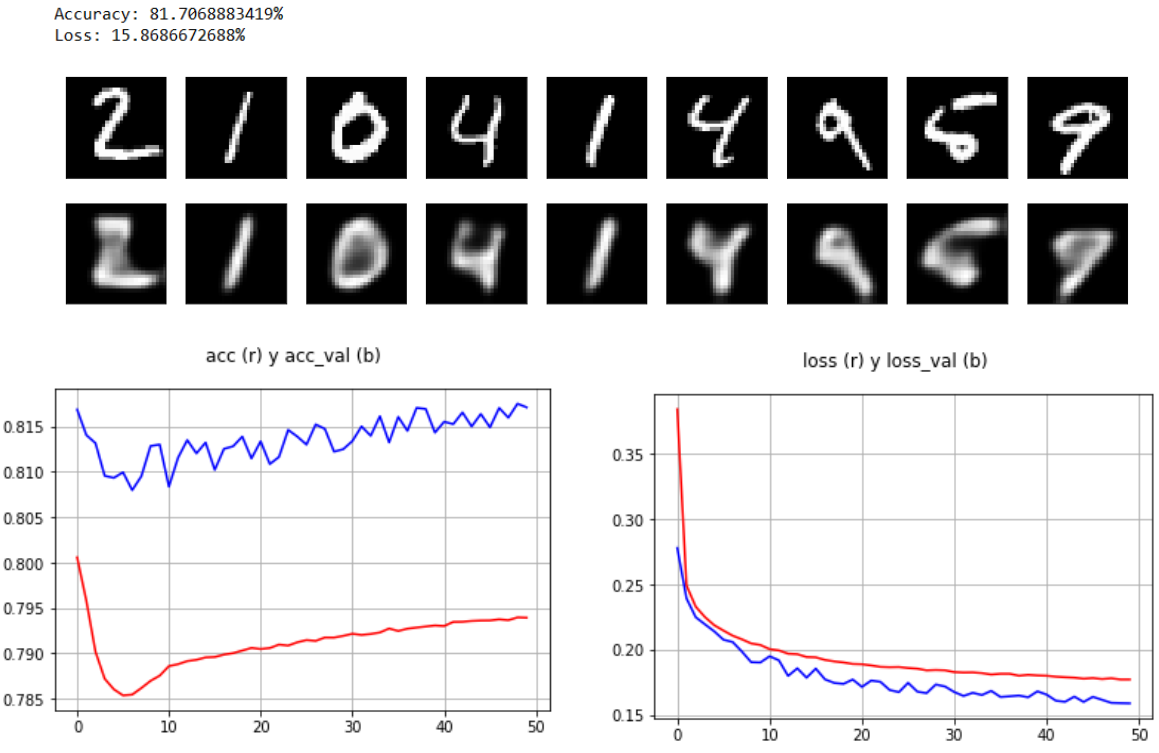

Figura 4-14. Prueba AE convolucional (versión 1) con DP=0.2 en la entrada y en capas internas

Con Dropout solo en la capa de entrada mejora mínimamente con respecto al caso básico de autoencoder convolucional. Sin embargo, cuando se añade DP en las capas internas se observan peores resultados. Esto no es raro, estamos desactivando neuronas aleatoriamente pero no estamos haciendo gran cantidad de iteraciones ni tenemos una gran red neuronal, por lo que en principio parece lógico pensar que el modelo pueda empeorar.

Por último, quedaría la prueba en la que se incluyen tanto la regularización de Dropout como el Batch Normalization (BN). Igualmente, lo realizamos en dos ejecuciones distintas: una cuando el DP se aplica solo a la entrada y otra en la que se aplica también en las capas internas.

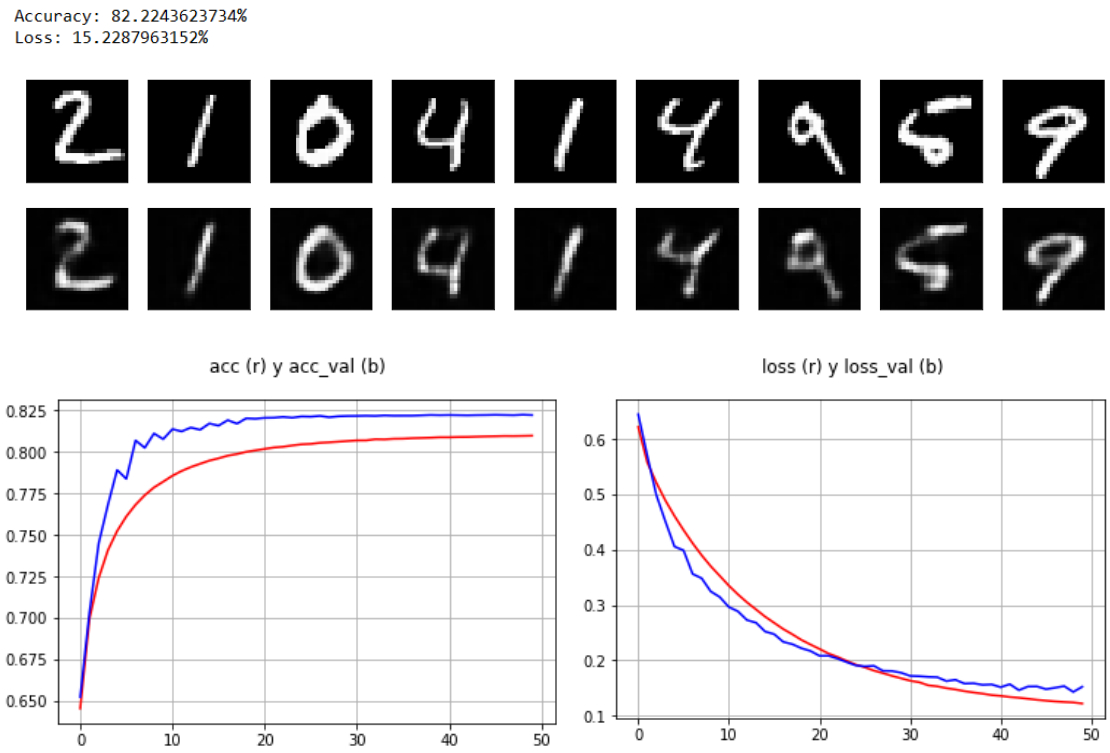

Figura 4-15. Prueba AE convolucional (versión 1) con BN y DP=0.2 en la entrada

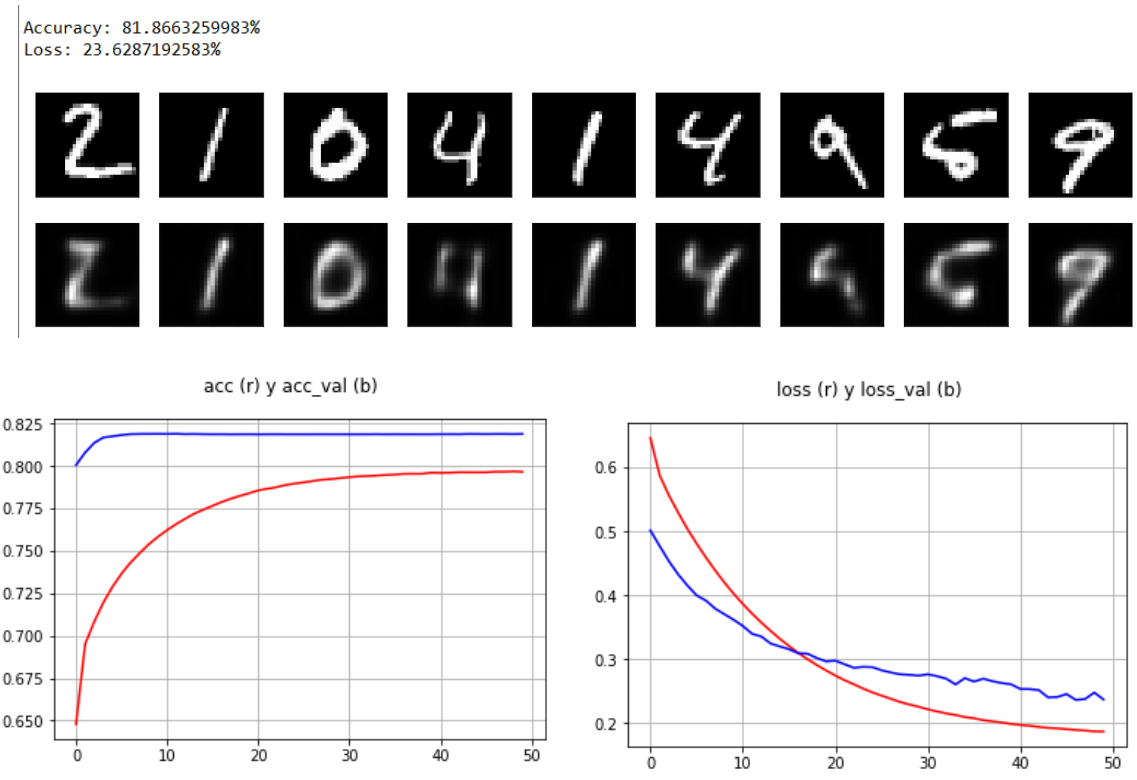

Figura 4-16. Prueba AE convolucional (versión 1) con BN y DP=0.2 en la entrada y en capas internas

Seguimos teniendo grandes valores de pérdidas en la validación uniendo el batch normalization junto con el dropout. Al igual que se comentaba en el caso de usar solo el dropout, en este caso no tenemos muchas iteraciones ni una gran cantidad de neuronas.

En algunos casos de los mostrados se ha podido observar un efecto curioso en las gráficas de pérdidas: los valores de las pérdidas de validación en las primeras etapas son menores que las de entrenamiento. Esto a pesar de parecer ilógico que se obtenga mejor resultado al comienzo con las muestras que se van a predecir, puede deberse a que el número de muestras de entrenamiento es tres veces mayor que las de validación (75-25). Dado que el error se calcula sobre todas las muestras, se podría esperar que la medida de las pérdidas al comienzo sea también algo menor en los datos de validación. Esto se corrige posteriormente en las siguientes etapas en las que ambas gráficas se aproximan.

Con esto ya hemos realizado un primer análisis de las posibilidades que nos ofrece el modelo y cómo se comporta ante determinados cambios introducidos. Esto será muy diferente en función de los datos de entrada, y habrá que rediseñar la red para que se llegue a la solución deseada.

En el próximo capítulo se extrapolará esto a la aplicación de las imágenes de Rayos X de cuadros, intentando llegar a una solución óptima a la salida del auto-codificador.

5 APLICACIÓN SOBRE IMÁGENES DE RAYOS X DE CUADROS

Aprendí que lo difícil no es llegar a la cima, sino jamás dejar de subir.

- Walt Disney -

El ámbito del estudio de la Historia del Arte se está viendo en auge en tanto que mediante el uso de las técnicas actuales que permite la tecnología se pueden obtener nuevas propiedades de las obras que no se conocían anteriormente.

Se va a aplicar lo estudiado anteriormente sobre los auto-codificadores sobre unas imágenes de Rayos X de obras de arte para ver si se pueden obtener ciertas características interesantes. La salida del autoencoder debería haber eliminado en cierto modo el ruido que contiene la radiografía: grietas, cortes, etc. La idea principal es que los hilos que forman el tejido sean observables fácilmente. Con esto, y como ya se comentó durante la introducción, puede servir de tarea previa importante para realizar luego análisis de conteo o análisis de los patrones de entrelazado de los tejidos.

En la siguiente ilustración se ha tomado de ejemplo la radiografía de la obra "*Retrato de un hombre viejo con barba"* de Vincent Van Gogh (una de las imágenes utilizadas para la realización del proyecto cedidas por el *RKD*) para mostrar de forma clara cuál es el objetivo final de esta aplicación.

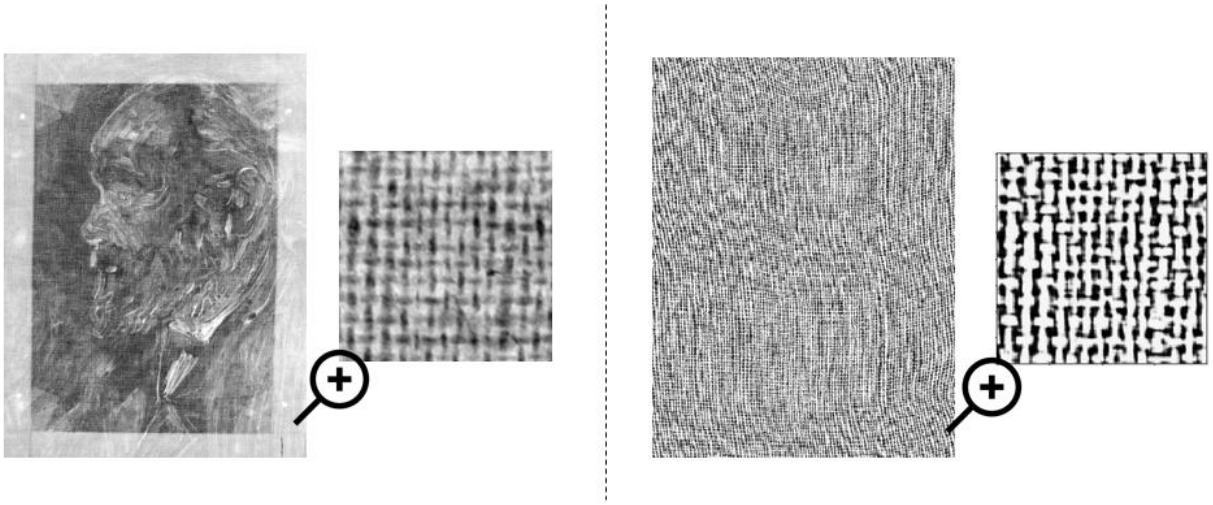

Figura 5-1. Ejemplo de tratamiento de imagen de Rayos X tras pasar por un autoencoder

A la izquierda se puede observar la radiografía con un zoom de un recorte de la imagen realizado a 200x200, mientras que a la derecha se observa la imagen reconstruida tras haber pasado por el auto-codificador, y de nuevo un zoom de un recorte a 200x200 de esta imagen de salida.

La diferencia entre ambas es notable, mientras que en el original resulta bastante complicado distinguir los hilos, a la salida se ha procesado la imagen y reducido el ruido, lo que hace que los hilos se muestren de manera mucho más visible al ojo humano.

Para lograr esto, se ha planteado una arquitectura sobre Matlab y Python con varias partes bien diferenciadas para lograr el tratamiento de la imagen que se desea realizar. En la siguiente figura se puede observar la idea principal de dicha arquitectura desarrollada.

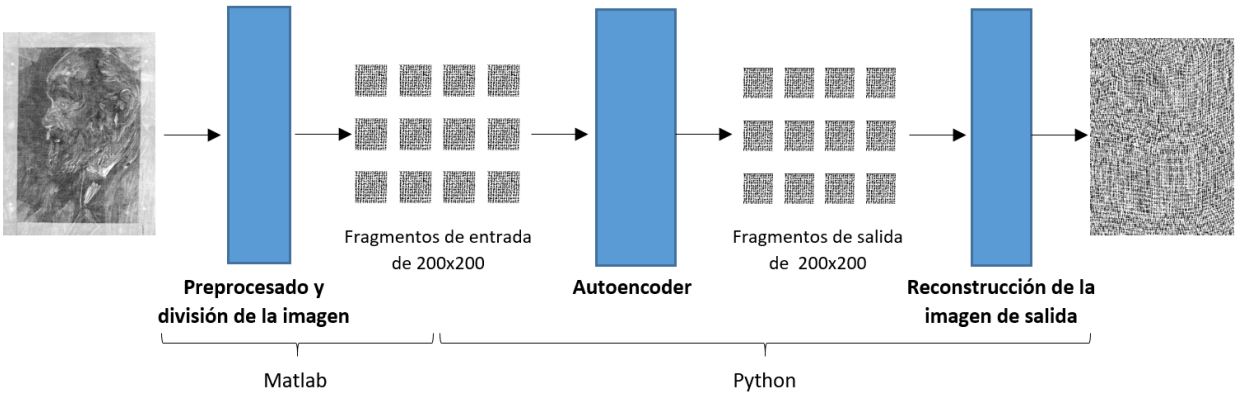

Figura 5-2. Arquitectura principal de la aplicación sobre Rayos X de cuadros

Esta idea se va a ver modificada en cuanto a que diferenciaremos por un lado el entrenamiento del auto-codificador, en el que se le pasará como entrada diversos fragmentos de diferentes imágenes para lograr un algoritmo generalista, y por otro lado la parte en la que una vez entrenado el autoencoder, se le pasa como entradas todos los fragmentos de un lienzo y su salida posteriormente es reconstruida.

Todo esto se va a detallar en las siguientes secciones, que se han ordenado según las 3 fases de la arquitectura presentada.

A continuación se muestra un listado con el conjunto de imágenes de Rayos X que se ha trabajado, y que irán apareciendo a lo largo de esta memoria.

Tabla 5–1. Listado de imágenes de Rayos X de cuadros

Nombre imagen	Cuadro	Pintor	Cedida por
F00205p01	Retrato de un hombre viejo con barba	Vincent van Gogh	RKD
F00651p01	Jardín del Hospital de Saint-Paul (La caída de las hojas)	Vincent van Gogh	RKD
MNP07905a00xf	Dos racimos de uvas con una mosca	Miguel de Pret	Museo Nacional del Prado
MNP07906a00xf	Dos racimos de uvas	Miguel de Pret	Museo Nacional del Prado
p01175p02xf2017	Mercurio y Argos	Diego Velázquez	Museo Nacional del Prado

5.1 Preprocesado y división de la imagen

En la siguiente ilustración se ha detallado un diagrama de flujos con los diferentes pasos que se siguen en el código generado para esta primera fase, realizado en Matlab (*preprocesado_imagenRayosX.m*).

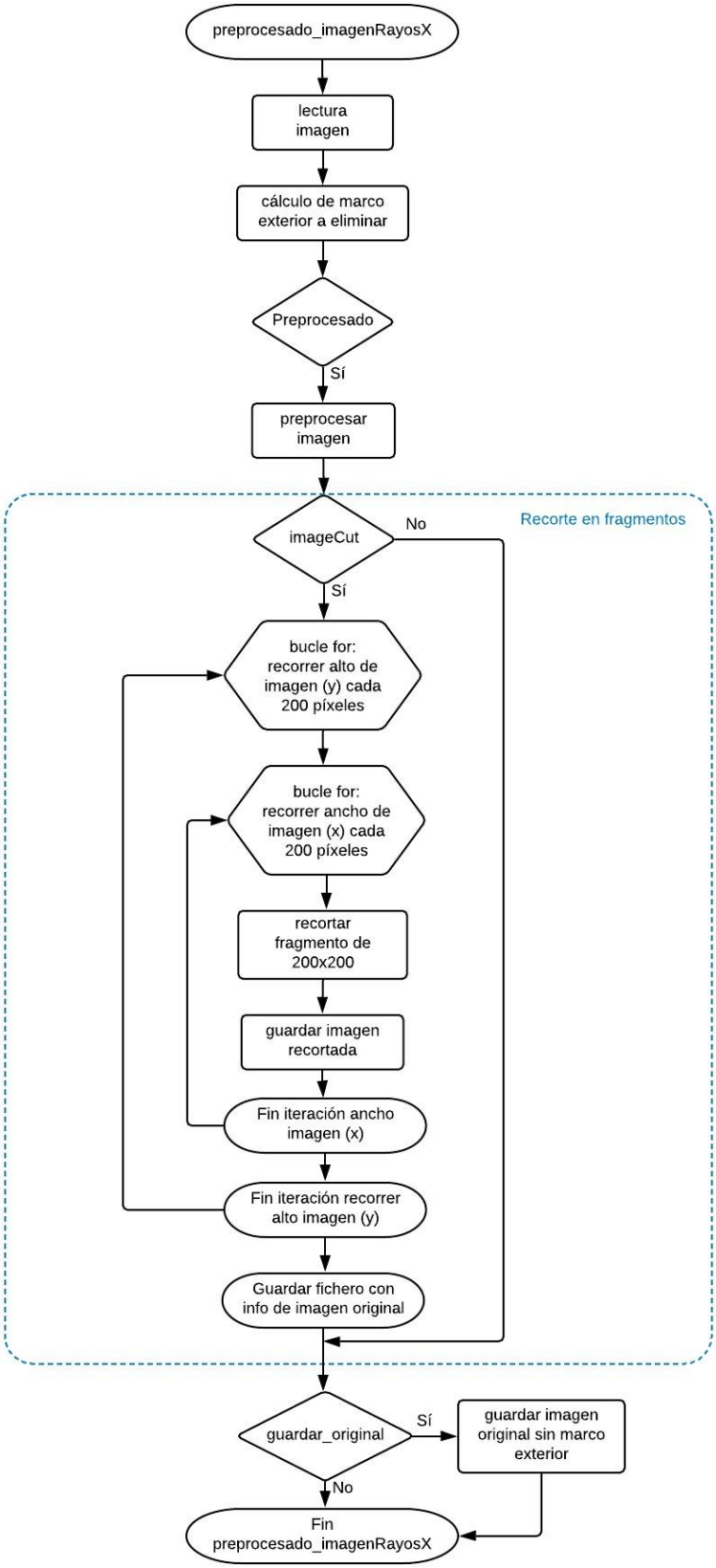

Figura 5-3. Diagrama de código Matlab para *preprocesado_imagenRayosX.m*

Debido a limitaciones computacionales, no es posible pasarle una imagen completa de gran tamaño como entrada al auto-codificador. Es por ello que la primera fase previa al entrenamiento pasa por dividir la imagen en fragmentos más pequeños que sí puedan ser procesados por la red neuronal.

Esta división se ha realizado en fragmentos de 200x200 píxeles. Se ha decidido tomar este tamaño porque en la conversión a centímetros supone aproximadamente que sea de 1cm x 1cm, lo cual es un tamaño estándar que suele ser empleado en el problema en cuestión.

Además de este particionado, se le realiza un preprocesado a la imagen de Rayos X. Esto se hace para normalizar los fragmentos, de forma que todos tengan media 1 y varianza 0. También se le hace el complemento a la imagen (inversa) de manera que los ceros se convierten en unos y viceversa. Esto simplemente se realiza porque posteriormente se podrán visualizar mejor los hilos, al invertirse los blancos y los negros.

Los fragmentos no se obtienen de la imagen completa: se realiza un recorte de un 10% de los márgenes externos de ésta. Esto se debe a que la mayoría de las radiografías con las que se ha trabajado incluyen un borde donde se ha grabado información de la imagen (nombre, identificación, etc.). Como esto no se desea para el entrenamiento del autoencoder, se elimina.

Dentro de la parte en la que se realizan los recortes, se recorren bucles con el tamaño del ancho y alto de la imagen, saltando en bloques de 200 píxeles, y generando dicho recorte de 200x200, el cual será guardado en el directorio indicado como parámetro.

Tras recorrer estos bucles, se guarda un pequeño fichero en el mismo directorio con la información acerca de la imagen original que posteriormente será leída desde el código Python para la reconstrucción de los recortes. Dicho archivo en formato .txt contendrá los siguientes datos: nombre de la imagen, tamaño del recorte, tamaño x (ancho) de la imagen, tamaño y (alto) de la imagen, y la resolución.

5.2 Autoencoder

Una vez se han generado los recortes, ya se pueden emplear como entrada para el auto-codificador. Como en cualquier problema de aprendizaje, para el autoencoder tendremos dos fases: la de entrenamiento, donde se aprenderá a partir de unas muestras y se "creará" el modelo; y la de predicción, donde se introducirán en la red neuronal los fragmentos de una imagen completa y devolverá unos fragmentos de salida tras pasar por el modelo anteriormente creado.

5.2.1 Entrenamiento

Se han tomado aleatoriamente cierto número de recortes generados de varias imágenes de Rayos X para que formen parte de la entrada del auto-codificador durante esta fase de entrenamiento. Esto se hace así para encontrar un modelo general, que no solo aprenda de una imagen concreta, y sea capaz de tomar decisiones para otras imágenes nuevas que se le presenten.

En total, las muestras introducidas vienen dadas por la siguiente tabla, que refleja el número total de fragmentos que se han producido para cada imagen en la fase anterior, y el número de éstos que han sido seleccionados para el entrenamiento.

Tabla 5–2. Fragmentos empleados para la fase de entrenamiento del AE

Imagen	Total fragmentos generados	Nº fragmentos de entrenamiento
F00205p01	1271	620
F00651p01	3933	456
MNP07905a00xf	744	217
MNP07906a00xf	744	124
p01175p02xf2017	3164	701
Total	9856	2118

En el siguiente diagrama se han detallado los pasos que se siguen en el código (en este caso se trabaja en Python) para el entrenamiento del autoencoder (*entrenamientoAE_rayosx.py*). Este conjunto de acciones va a ser general para todos los modelos empleados, lo que cambiará será la arquitectura del auto-codificador diseñada, que se incluirá dentro de dicho apartado.

Figura 5-4. Diagrama de código Python para *entrenamientoAE_rayosx.py*

Para la lectura de las imágenes en Python, se recorre el directorio donde se han almacenado y se van leyendo una a una y añadiendo a una lista, que posteriormente será la entrada del auto-codificador. A continuación se detalla el fragmento de código que realiza esto (se excluye el fichero con la información de la imagen original que también se almacena en dicho directorio):

```
for archivo in listdir(carpeta_entrenamiento):
    if archivo!="info_img_orig.txt":
        nombre_img=carpeta_entrenamiento + archivo;
        img = io.imread(nombre_img, as_grey=True);
        img = img/255;
        imagenes.append(img);
        cuenta=cuenta+1;
```

La separación de las muestras para entrenamiento y para validación se ha realizado con una división del 70% y del 30% del total de muestras de entrada, respectivamente.

La forma de realizar la arquitectura del autoencoder, así como su posterior compilado, entrenamiento, predicción y representaciones se realiza de la misma manera que se mostró para el caso de las pruebas iniciales con MNIST.

A lo largo del desarrollo del proyecto se ha realizado una gran cantidad de pruebas con diversos modelos de autoencoders convolucionales para intentar encontrar las salidas óptimas que reflejen lo que se busca (por recordar, se quiere eliminar el ruido de las imágenes e intentar que la visualización de los hilos sea la mejor posible). Será en el siguiente capítulo donde se detallarán las soluciones finales a las que se ha llegado tras las diversas pruebas y entrenamientos del AE.

Un tema importante es que tras entrenar el modelo, se quiere guardar para posteriormente poder realizar las predicciones sin tener cada vez que realizar el entrenamiento (con el tiempo y coste que ello conlleva). Este guardado del modelo se puede realizar de forma muy sencilla con Keras, con una única línea de código, siendo la variable '*autoencoder'* el modelo creado previamente:

```
autoencoder.save("modelo_autoencoder.h5")
```

De cara a facilitar el uso de esta aplicación se ha creado una función "main" que se encarga de llamar al resto de los códigos y disponer de los parámetros de forma más intuitiva. Es por ello, que para el código de entrenamiento se ha adecuado en una función, cuyos parámetros se detallan a continuación:

Tabla 5–3. Parámetros de la función *entrenamientoAE_rayosx*

Nombre parámetro	Descripción
carpeta	Directorio donde se encuentran los recortes de entrada (p.ej.: muestras varias)
ruta_modelo	Ruta del fichero .h5 donde se guardará el modelo del AE (pesos, etc.)
num_epochs	Número de epochs empleado para entrenar el modelo
batch_s	Valor del 'batch_size' a utilizar para el entrenamiento
optimizer_func	Función de optimización para el entrenamiento del modelo
loss	Función de pérdidas para el entrenamiento del modelo

Se muestra ahora un ejemplo de llamada a esta función de forma independiente (sin necesidad de utilizar el "main") con unos parámetros de prueba:

```
entrenamientoAE_rayosx("C:/Users/FJMC/Desktop/TFM/RayosX/muestras_varias/",
                  "C:/Users/FJMC/Desktop/TFM/RayosX/modelo_autoencoder.h5",
                  50,
                  50,
                  'adadelta',
                  'binary_crossentropy')
```

5.2.2 Predicción

Tras entrenar el modelo, podemos utilizarlo para realizar lo que se mostraba en la Figura 5-2, en la que se le pasa como entrada del auto-codificador el conjunto de fragmentos de una imagen completa, de forma que se obtiene una salida formada por nuevos fragmentos, con el objeto de que posteriormente dichos fragmentos sean utilizados para reconstruir de nuevo la imagen.

Para ello, será necesario cargar el modelo que previamente se había guardado. De nuevo en Keras con una simple línea de código es posible volver a tener el modelo:

```
autoencoder = load_model("modelo_autoencoder.h5")
```

El resto de acciones realizadas durante la predicción siguen siendo las mismas que las empleadas para el entrenamiento (carga de imágenes, predicción, etc.), excepto que en este caso no es necesario ni compilar ni entrenar la red neuronal.

El principal motivo por el que se ha diferenciado el código para entrenamiento y predicción de imágenes completas es que las entradas del autoencoder son diferentes para cada una: en el primer caso se toman una serie de muestras aleatorias de diferentes imágenes, mientras que para el segundo los fragmentos pertenecen a toda una imagen completa de forma ordenada.

Otro motivo importante es que, como ya se ha comentado, si se tiene que entrenar el mismo modelo cada vez que queremos pasar una imagen por el autoencoder es necesario un tiempo y coste de computación excesivamente alto.

El código donde se desarrolla la predicción de las imágenes completas se ha incluido en una función a la que

se ha añadido la reconstrucción de la imagen a partir de los fragmentos de salida (esta reconstrucción se detallará en la siguiente sección). A continuación se muestra el diagrama de bloques de la función diseñada:

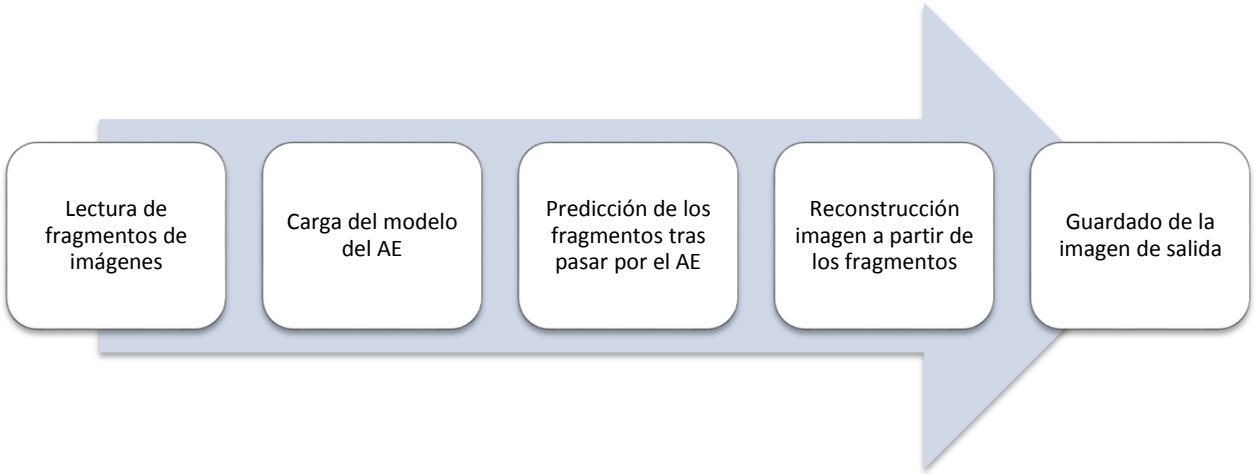

Figura 5-5. Diagrama de código Python para *ejecucionAE_rayosx*

Esta función se ha llamado con el nombre de '*ejecucionAE_rayosx*' y necesita que se le pasen como parámetros los que se detallan en la siguiente tabla:

Tabla 5–4. Parámetros de la función *ejecucionAE_rayosx*

Nombre parámetro	Descripción
carpeta	Directorio donde se encuentran los recortes de entrada de la imagen completa a reconstruir
tam_x	Tamaño x (ancho) de imagen original para reconstruccion
tam_y	Tamaño y (alto) de imagen original para reconstruccion
modelo_AE	Ubicación del modelo del AE a utilizar (previamente entrenado)
ruta_salida	Ubicación donde se almacenará la imagen de salida
resolucion_salida	Resolucion que tendrá la imagen reconstruida (coincidirá con la de la imagen original)

Un ejemplo de llamada a esta función con unos parámetros como prueba es:

```
ejecucionAE_rayosx("C:/Users/FJMC/Desktop/TFM/RayosX/F00205p01_EnSim/",
           10420,
           7923,
           "C:/Users/FJMC/Desktop/TFM/RayosX/modelo_autoencoder.h5",
           "C:/Users/FJMC/Desktop/TFM/RayosX/recon/recon_AE_F00205p01.tif",
           600)
```

El hecho de realizar esta parte como una función es que se ha pensado también para una futura integración en otro código o incluso en otro lenguaje (como puede ser Matlab), de forma que desde ahí sea posible llamar a la función de Python y el resultado que devuelve (la imagen reconstruida) pueda emplearse en otro código.

5.3 Reconstrucción de la imagen de salida

Esta parte de la aplicación se ha integrado dentro de la función '*ejecucionAE_rayosx*' comentada anteriormente. Su objetivo es simplemente el de obtener una imagen completa tal y como la original con los fragmentos obtenidos a la salida del auto-codificador.

A pesar de intentar realizar una reconstrucción más compleja que eliminara el efecto de los bordes, se ha optado por la solución más sencilla, debido a que el estudio de otro tipo de reconstrucciones se excede a los objetivos de este proyecto y se ha decidido dejar como mejora futura.

La forma en la que se hace la reconstrucción es mediante la unión de nuevo de los fragmentos en el mismo orden en el que fueron obtenidos. El orden es el elemento más importante, y esto se traslada a la primera parte de la aplicación en la que se realizaba la división de la imagen original.

Durante dicha división es necesario que los nombres de los fragmentos generados se crearan de manera ordenada. Como la división se realizaba recorriendo fila a fila la imagen en saltos de 200 píxeles, el nombre de los fragmentos se ha generado con forma de matriz identificando cada fragmento con la fila y columna a la que pertenecen. Con esto es posible mantener el mismo orden a la hora de leer las imágenes desde la función y poder realizar la unión de los fragmentos sin problemas.

Otro asunto importante a la hora de reconstruir la imagen es la forma de obtener el tamaño que debe tener. Es necesario recordar que la imagen original fue recortada eliminando un 10% de sus bordes, así como que al coger bloques de 200x200 es posible que algunos píxeles quedaran descolgados.

A continuación se muestra un ejemplo sobre la forma de obtener los tamaños de la imagen a reconstruir:

Ejemplo 5–1. *Obtener tamaño de la reconstrucción a partir de una imagen de 10420×7923 y una división en fragmentos de 200×200 píxeles.*

Durante la eliminación de los bordes se suprime un 10% de cada lado de la imagen.

```
borde_x=round((n*0.1));
borde_y=round((m*0.1));
```

Con esto, para una imagen de 10420×7923 se quedaría en 8336×6339.

Para el cálculo de la reconstrucción se tiene lo siguiente:

$$filas = \frac{8336}{200} = 41.68 \approx 41\ filas\ (bloques\ de\ 200\ en\ y) \rightarrow\ m = 41 \times 200 = 8200 \tag{5–1}$$

$$columnas = \frac{6339}{200} = 31.695 \approx 31\ columnas\ (bloques\ de\ 200\ en\ x) \rightarrow\ n = 31 \times 200 = 6200 \tag{5–2}$$

La imagen reconstruida será por tanto de tamaño 8200×6200.

Esto dista de la imagen original sin bordes, que tiene 8336×6339. Esto se debe a las transformaciones realizadas al coger en bloques de 200, lo que hace que se pierdan algunos píxeles de los bordes derecho e inferior de la imagen.

Para unir los fragmentos, se recorre la imagen de la misma forma que se realizó para la división, recorriendo los fragmentos organizados en una matriz de bloques de 200×200. En cada iteración, se introducirá el fragmento correspondiente en una nueva matriz '*recon*' del tamaño obtenido previamente. El tamaño se ha definido con un vector de dos valores (m, n) denominado '*out_shape*'.

Se muestra a continuación la parte del código Python donde se realiza dicha reconstrucción, que como se puede observar no requiere de muchas líneas:

```
recon = np.empty(out_shape);
for i in range(0,out_shape[0],200):
    for j in range(0,out_shape[1],200):
        recon[i:i+200,j:j+200] = decoded[p];
        p=p+1;
```

El problema de esta solución para reconstruir, como ya se ha comentado, es que no elimina el efecto de los bordes. Esto es un efecto no deseado ya que puede dar problemas a la hora de realizar el conteo de los hilos del tejido.

Si hacemos un zoom de una de las imágenes reconstruidas se puede observar de forma leve dicho efecto. En este caso se ha tomado un zoom que engloba 4 bloques, y los bordes de unión de estos se pueden diferenciar a simple vista.

Figura 5-6. Zoom de reconstrucción de imagen a la salida para *F00205p01*

6 SOLUCIONES

Nos encanta tener razón, pero aprendemos más cuando cometemos errores.

- Dylan Wiliam -

En este capítulo se presentan las salidas que mejor solución proporcionan para la eliminación de ruido sobre las imágenes de Rayos X. Previamente, se expondrán de forma breve algunos de los problemas hallados para llegar hasta ellas.

6.1 Problemas encontrados

Durante la realización de este proyecto se han efectuado una gran cantidad de pruebas con diversas arquitecturas y/o cambio de parámetros sobre el autoencoder de forma que se llegara a una solución que visualmente pudiera ofrecer una buena calidad a la hora del conteo de hilos sobre el tejido de los lienzos.

En esta sección se presentan de forma breve algunos de los contratiempos que se han presentado a lo largo de la realización de dichas pruebas y las soluciones aplicadas que han llevado a las que serán las soluciones finales que posteriormente se exponen.

6.1.1 Errores de recursos

Las primeras pruebas realizadas pasándole como entrada al auto-codificador un conjunto de imágenes recortadas de tamaño 400x400 píxeles en lugar de los 200x200 que finalmente se han empleado provocaron un excesivo consumo de recursos en el equipo, provocando el error que se muestra a continuación:

```
 ResourceExhaustedError (see above for traceback): OOM when allocating tensor
with shape[128,32,400,400] and type float on
/job:localhost/replica:0/task:0/device:GPU:0 by allocator GPU_0_bfc
```

Este tipo de errores no solo han aparecido por usar fragmentos de mayor tamaño, en casos en el que la arquitectura del auto-codificador ha sido de un número considerable de capas o en el que por ejemplo el valor del batch-size se ha aumentado, también ha aparecido un error similar al mostrado.

6.1.2 Redimensionado de fragmentos

Sumado al problema de recursos, otro de los inconvenientes encontrados ha sido la forma en la que se codifican los fragmentos de la imagen a la hora de ser leídos desde Python.

La mayoría de las librerías empleadas en Python para la lectura de imágenes realizan una representación de los píxeles sobre una matriz de intensidad para la escala de grises en enteros de 16 bits ($2^{16} = 65536$ valores en el rango [0, 65535]). Sin embargo, a la hora de pasar esto como entrada para el autocodificador (recordamos que se emplea Keras que usa el motor de TensorFlow) la representación de los píxeles se realiza en enteros de 8 bits ($2^8 = 256$ valores en el rango [0, 255]).

Como primera solución para ambos problemas se propuso utilizar una función que redimensionara la imagen de forma que por un lado al tener una imagen de menor tamaño no hubiera problema de recursos sobre la máquina, y por otro lado que eliminara el problema de la codificación en bits.

Para realizar esto, se empleó la función "resize" de *scikit-image*, la cual comprimía la imagen al tamaño que le fuera indicado. Con esto, las salidas del auto-codificador comenzaron a mostrar resultados más coherentes y visualmente aceptables.

El mayor inconveniente de emplear este "resize" es que la calidad de los fragmentos disminuía considerablemente, lo cual afectaba a la salida del auto-codificador. El uso de esta función resultaba un hecho contradictorio con la idea de un auto-codificador, ya que su función principal es la de "codificar" la imagen para volver a "decodificarla" posteriormente. Si se usaba este redimensionado previo a la entrada del auto-codificador se estaba realizado ya dicha codificación con anterioridad a la red neuronal, por lo que los resultados se veían "manipulados".

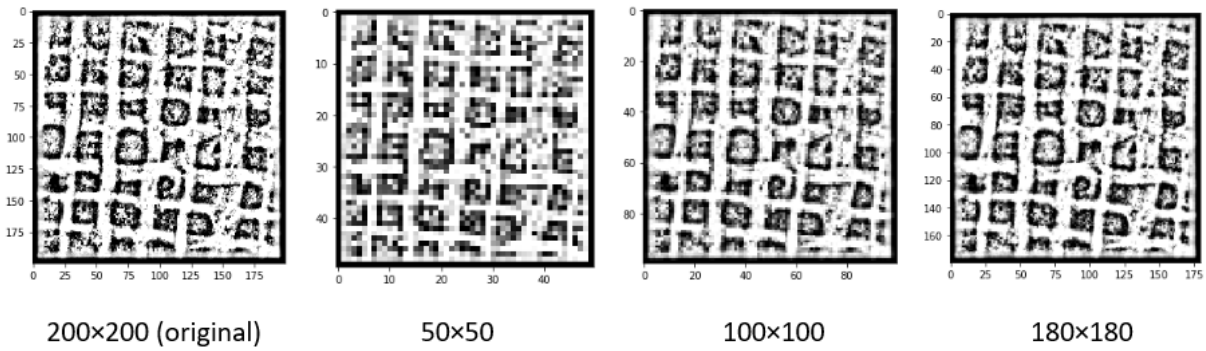

Figura 6-1. Comparativa de imágenes con "resize" para fragmentos de la obra *Ixión* de José de Ribera

Para evitar este inconveniente con respecto a la calidad de las imágenes, finalmente se ha optado por no realizar este redimensionado. Las alternativas que se han empleado para solucionar los problemas de recursos y codificado en la lectura de imágenes han sido las siguientes:

- Cambios de arquitecturas del auto-codificador, reduciendo el número de capas
- Reducción del valor del batch-size
- Cambio de representación de píxeles de 16 bits a 8 bits (dividiendo la matriz de píxeles que compone la imagen entre 255) para evitar así el problema de compatibilidad en dicha representación de píxeles a la entrada del auto-codificador

6.2 Decisiones y pruebas

A partir de las conclusiones obtenidas con las pruebas del auto-codificador con la base de datos MNIST en el apartado 4.2.2.3, se determina no emplear Batch Normalization, ya que se obtuvieron grandes valores de pérdidas a la salida. A esto se le suma que ya se le realiza a las imágenes una normalización de media y

varianza previa a recortarlas (en Matlab), lo cual equivale a realizar dicho Batch Normalization.

Sin embargo, sí que se han realizado diversas pruebas incluyendo las capas de Dropout, tanto a la entrada como en las capas intermedias; así como pruebas modificando el valor del stride a la hora de realizar la convolución.

6.2.1 Dropout

Se han probado diversas estructuras cambiando el número de capas, etc. Todas las pruebas se realizaron con un entrenamiento de la red neuronal de 50 *epochs*.

A partir de aquí, se va a emplear como ejemplo la salida del autoencoder para el cuadro *Mercurio y Argos* de Diego Velázquez (*p01175p02xf2017*). Para una mejor visualización, se ha tomado uno de los fragmentos a la entrada y se ha comparado con su salida del auto-codificador.

En primer lugar, se realizó una prueba con una estructura de red neuronal con capas de 32 y 16 neuronas, un *batch-size*=50 y un Dropout de 0.2 a la entrada:

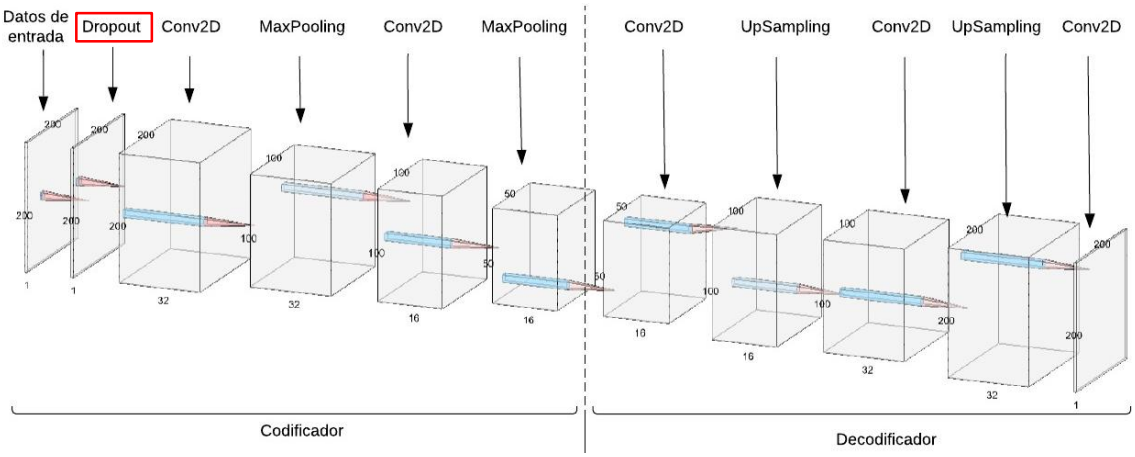

Figura 6-2. Arquitectura de capas auto-codificador con Dropout a la entrada

La salida obtenida tras pasar un fragmento por este auto-codificador es la que se refleja en la siguiente captura, donde se puede observar que aunque se ha eliminado un poco el ruido, las fronteras son difusas lo que hace que sea difícil de distinguir de forma correcta los hilos del tejido.

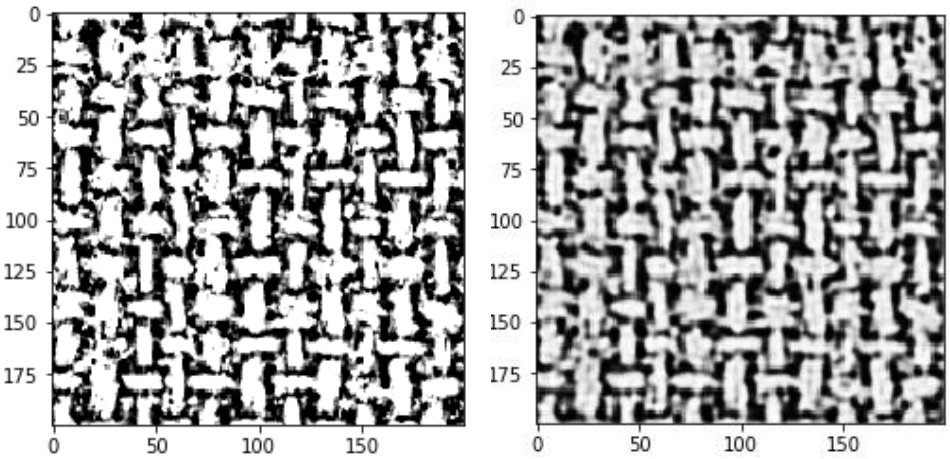

Figura 6-3. Resultados para AE con Dropout (versión 1). Imagen izquierda entrada, imagen derecha salida

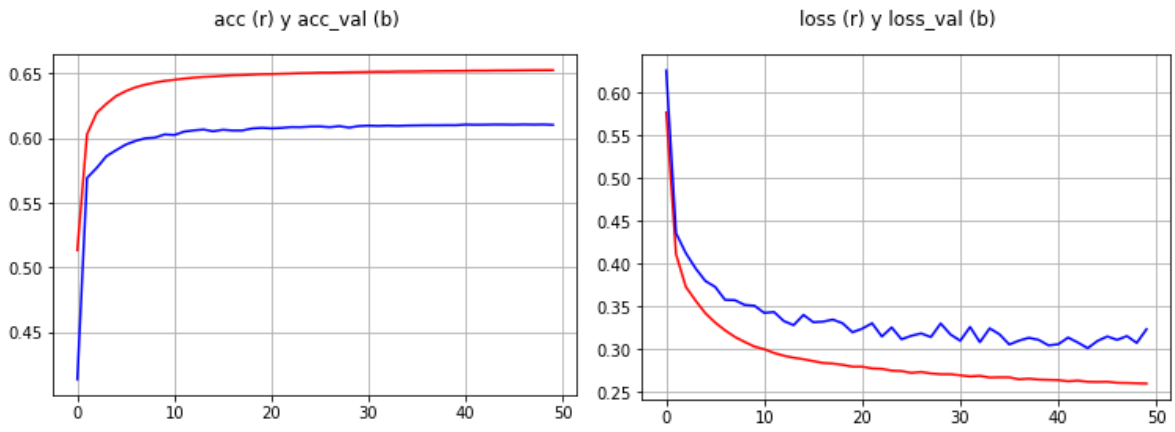

Figura 6-4. Resultados "accuracy" y "loss" durante entrenamiento de AE con Dropout (versión 1)

Al intentar sobre dicha arquitectura incluir un Dropout en las capas intermedias, se obtiene el error de problema de recursos en el equipo. Es por ello, que se decide reducir el número de neuronas en las capas, pasando éste a 16 y 8 neuronas respectivamente, sobre la misma arquitectura anterior, añadiendo para cada capa intermedia otra adicional de Dropout.

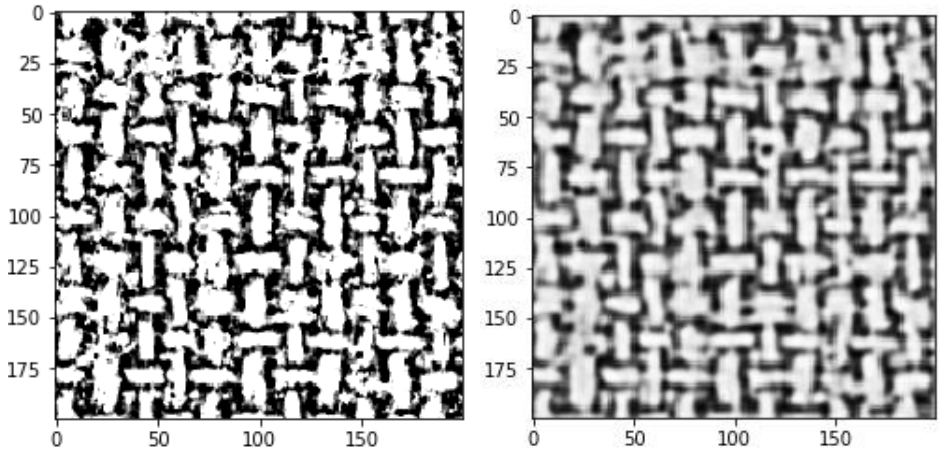

Figura 6-5. Resultados para AE con Dropout (versión 2). Imagen izquierda entrada, imagen derecha salida

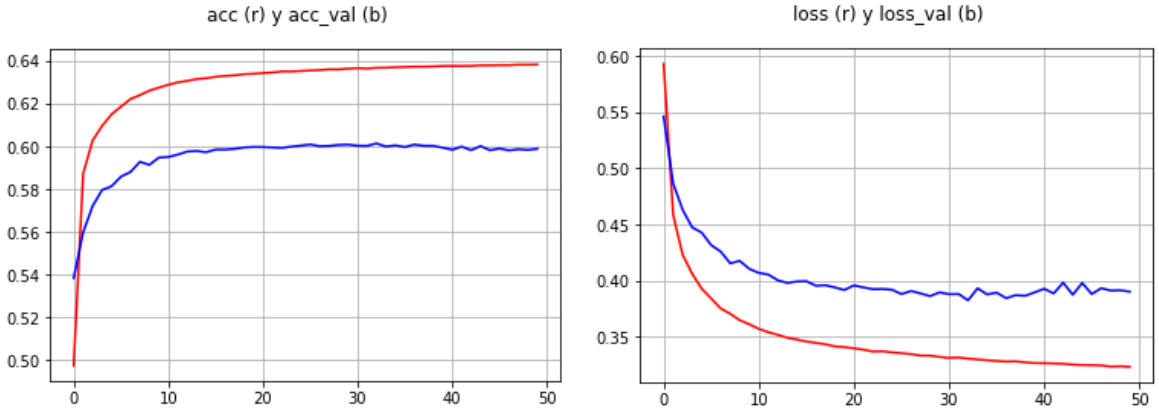

Figura 6-6. Resultados "accuracy" y "loss" durante entrenamiento de AE con Dropout (versión 2)

Como puede observarse en las figuras anteriores, la salida del autoencoder cuando se le incluye Dropout para el caso de tener como entrada los fragmentos de las imágenes de Rayos X no es muy buena, siendo algo más

nítida para el caso de incluir DP solo a la entrada que si se introduce en capas intermedias (teniendo en cuenta que además en este caso se ha reducido el número de neuronas). Por este motivo, se verá que en las soluciones elegidas finalmente no se emplea las capas de Dropout sobre la arquitectura del auto-codificador.

6.2.2 Stride

El cambiar el tamaño de paso o "stride" a la hora de realizar la convolución en el auto-codificador va a obligar a reestructurar de forma completa la arquitectura del auto-codificador, así como generará salidas diferentes a las ya vistas con el valor de "stride" por defecto, que es igual a 1.

En este caso, se ha tomado un *stride=2* durante la codificación y se ha reformulado completamente la red adaptando los tamaños obtenidos a las salidas de las capas de convolución. Aquí ya no se emplean capas de Dropout, y se puede observar que la disposición de las capas no coincide exactamente con el modelo "general" empleado en ejemplos anteriores.

El número de capas dedicado a la codificación será menor que el dedicado a la decodificación. Esto se debe a que durante la codificación se emplea una convolución con un *stride=2,* mientras que para la decodificación si se mantiene dicho valor el tamaño de la imagen continuaría disminuyendo. Es por ello que se vuelve al valor por defecto.

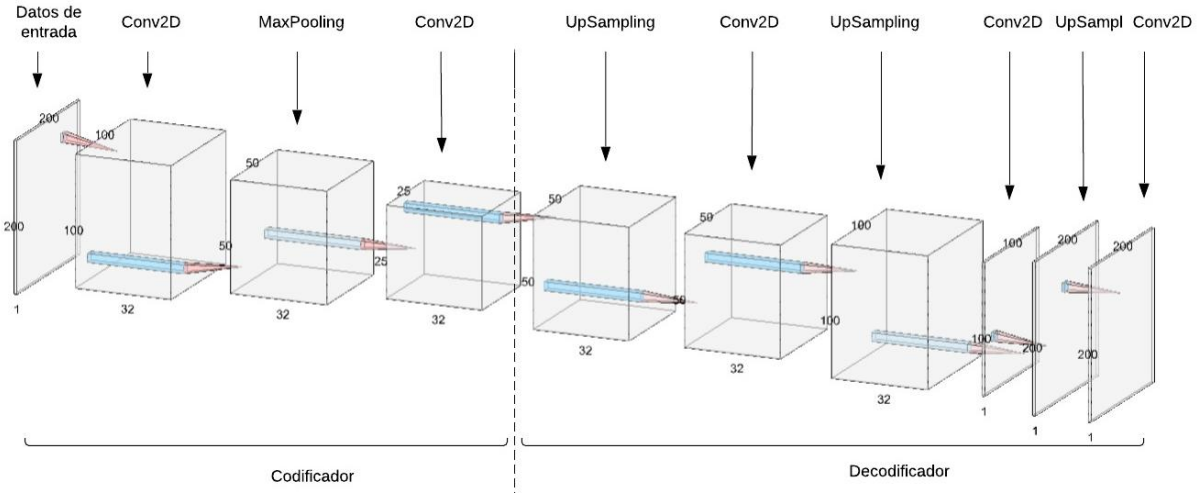

Figura 6-7. Arquitectura de capas solución auto-codificador con *stride=2*

A pesar de que la arquitectura es diferente y no "simétrica" con respecto a las capas entre el codificador y el decodificador, los resultados obtenidos son buenos visualmente comparados con las pruebas anteriores. Esta prueba se ha realizado con un entrenamiento de 50 "epochs".

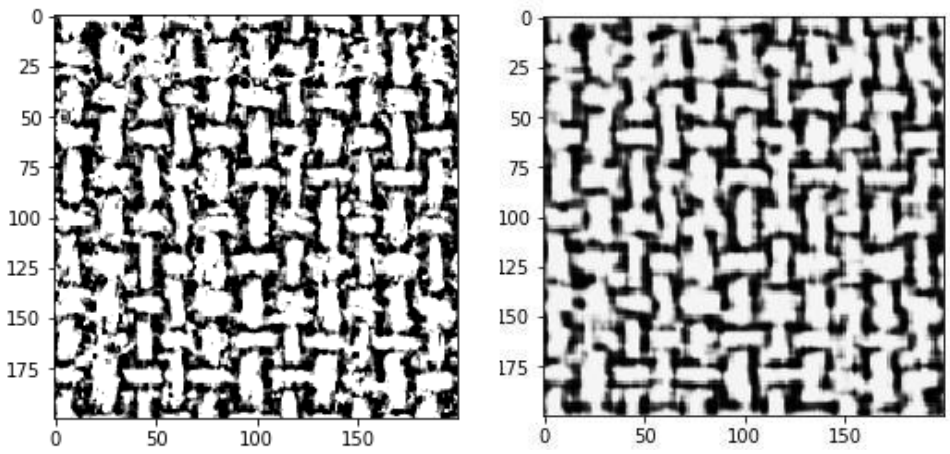

Figura 6-8. Resultados para solución AE con *stride=2*. Imagen izquierda entrada, imagen derecha salida

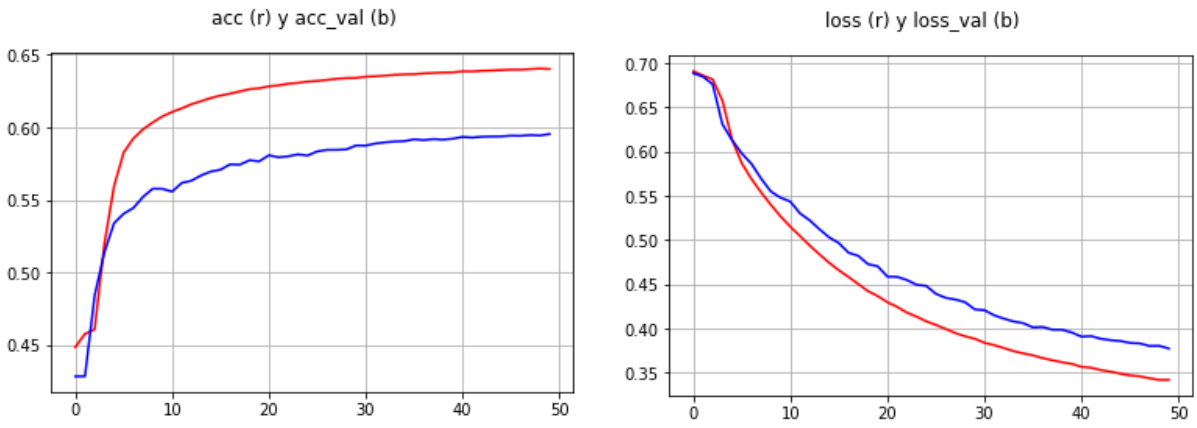

Figura 6-9. Resultados "accuracy" y "loss" durante entrenamiento de solución AE con *stride=2*

Como se ha podido observar, los resultados de este modelo son buenos, aunque si se reconstruye la imagen completa los hilos horizontales se vuelven difusos y complicados de distinguir. En el siguiente apartado se analizará la solución final adoptada en la que sin modificar el valor del "stride" se han obtenido resultados incluso mejores que los obtenidos con este caso.

6.3 Solución con auto-codificador

Tras numerosas pruebas realizadas, finalmente se ha decidido adoptar la solución que se describe a continuación, siendo una solución viable desde el punto de vista computacional y que aporta una buena calidad de imagen, donde se pueden distinguir mejor, visualmente, los hilos de la tela. Se podrá observar también que esta solución una disminución del ruido de las imágenes de Rayos X que se introducen en la entrada.

6.3.1 Arquitectura del modelo

Esta solución presenta una arquitectura en la que se ha vuelto a poner el valor de *stride* por defecto (*stride=1*), y se han mantenido capas de 32 y 16 neuronas. La diferencia principal con los modelos anteriores es que por un lado no se incluye en este caso Dropout (ya se ha visto que los resultados con esta capa no mejoran la salida) y se ha incrementado el número de capas, es decir, se han incluido dos capas más (una en el codificador y otra en el decodificador), que cada una de ellas engloban capas de convolución y muestreo.

En la siguiente tabla se resumen los parámetros empleados para el modelo y entrenamiento de este auto-codificador:

Tabla 6–1. Parámetros de diseño para solución de auto-codificador

Parámetro	Valor
Número de neuronas por capa	32 y 16
Número total capas codificador	3 (Conv2D + MaxPooling)
Número total capas decodificador	3 (Conv2D + UpSampling)
Número capas Dropout	0
Batch-size	50
Número de "epochs"	50
F. activación capas intermedias	ReLU
F. activación capa de salida	Sigmoide
Función de pérdidas	Entropía cruzada binaria

Esta arquitectura de auto-codificador convolucional se basa en la idea del modelo "undercomplete" que se presentó en los primeros capítulos, ya que el número de neuronas va disminuyendo conforme se llega a la imagen "codificada" (teniendo una capa oculta de 25x25x16). No obstante, el hecho de mantener un modelo convolucional y la búsqueda de una mejor visualización a la salida hace que dicha "imagen" codificada presente una profundidad no contemplada en los modelos del tipo "undercomplete".

La función de pérdidas utilizada para el entrenamiento sigue siendo la misma que la utilizada en las pruebas anteriores, la función de entropía cruzada binaria, detallada en el apartado 2.2.1.1.

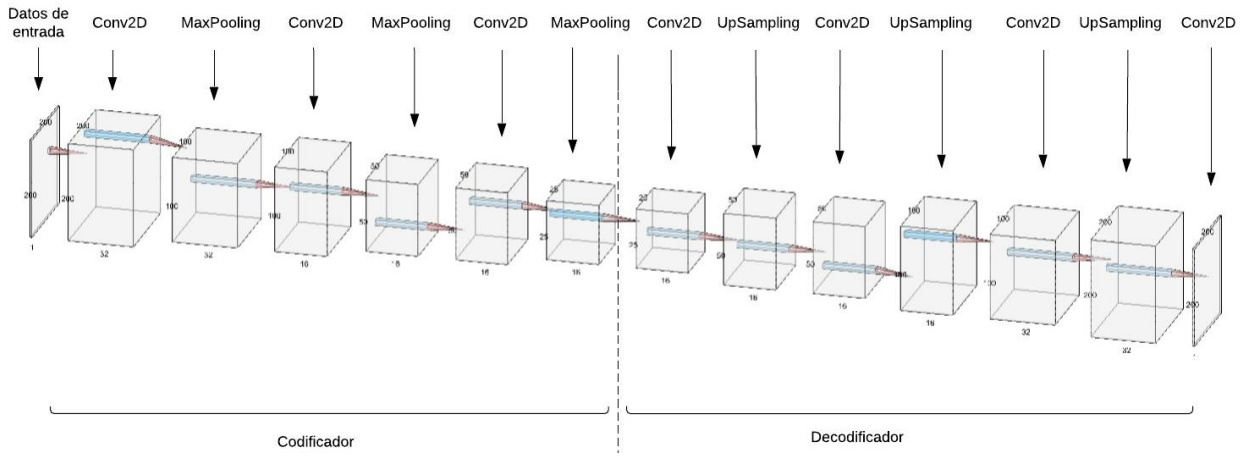

Figura 6-10. Arquitectura AE solución final problema imágenes Rayos X de cuadros

En la captura anterior se muestra un esquema de la arquitectura a nivel de bloques, en la que se puede observar de forma clara el aumento del número de capas de ésta. En el siguiente sub-apartado se detallan los resultados que han motivado a dicho aumento de capas convolucionales.

A continuación se muestra el detalle de dicha arquitectura, con cada una de las capas y el tamaño de ellas. Se han marcado en color azul las capas correspondientes a la imagen codificada y a la salida (imagen decodificada):

```
_________________________________________________________________
Layer (type) Output Shape Param #
=================================================================
input_1 (InputLayer) (None, 200, 200, 1) 0
_________________________________________________________________
conv2d_1 (Conv2D) (None, 200, 200, 32) 320
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 100, 100, 32) 0
_________________________________________________________________
conv2d_2 (Conv2D) (None, 100, 100, 16) 4624
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 50, 50, 16) 0
_________________________________________________________________
conv2d_3 (Conv2D) (None, 50, 50, 16) 2320
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 25, 25, 16) 0
_________________________________________________________________
conv2d_4 (Conv2D) (None, 25, 25, 16) 2320
_________________________________________________________________
up_sampling2d_1 (UpSampling2 (None, 50, 50, 16) 0
_________________________________________________________________
conv2d_5 (Conv2D) (None, 50, 50, 16) 2320
_________________________________________________________________
up_sampling2d_2 (UpSampling2 (None, 100, 100, 16) 0
_________________________________________________________________
conv2d_6 (Conv2D) (None, 100, 100, 32) 4640
_________________________________________________________________
```

```
up_sampling2d_3 (UpSampling2 (None, 200, 200, 32) 0
_________________________________________________________________
conv2d_7 (Conv2D) (None, 200, 200, 1) 289
=================================================================
Total params: 16,833
Trainable params: 16,833
Non-trainable params: 0
_________________________________________________________________
```

6.3.2 Entrenamiento y resultados

Los valores de "*accuracy*" y "*loss*" obtenidos durante el entrenamiento, como se puede observar en las siguientes gráficas, no difieren demasiado de las pruebas anteriores, de hecho incluso es un poco más inestable en las diferentes fases del entrenamiento, el cual se ha realizado para 50 *epochs*. Además, se ha mantenido un *batch-size* de valor 50.

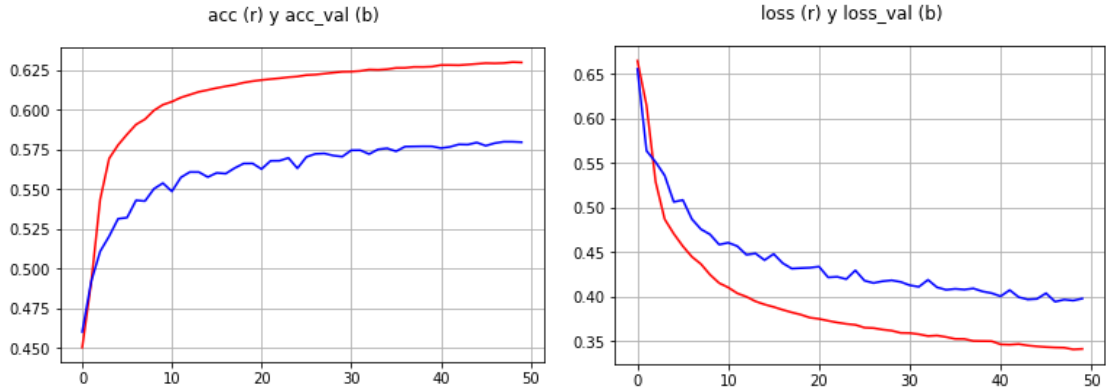

Figura 6-11. Resultados "accuracy" y "loss" durante entrenamiento de solución final AE

Siguiendo con el fragmento del cuadro *Mercurio y Argos* de Diego Velázquez (*p01175p02xf2017*), se muestra a continuación la comparativa entre la imagen a la entrada del auto-codificador y su salida, reflejando lo que se comentaba al comienzo de este apartado de la mejor visualización de los hilos en el fragmento de salida.

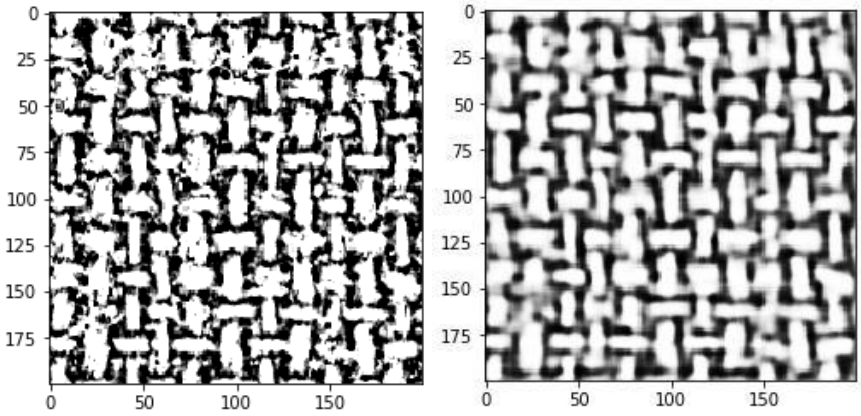

Figura 6-12. Resultados para solución final AE. Imagen izquierda entrada, imagen derecha salida

La salida no es totalmente nítida, y por eso la oscilación que se veía en los valores de "accuracy" y de pérdidas, pero sí que nos da una idea bastante buena si lo que se quiere es realizar un conteo de los hilos del tejido. Además, como ya se ha comentado, la entrada del AE está compuesta por fragmentos de varias imágenes, y esta solución es la que mejor salida obtiene para las diversas imágenes probadas. En el próximo capítulo se analizará de forma más detallada los resultados obtenidos tras reconstruir las imágenes.

6.3.3 Reduciendo la capa "codificada"

El modelo anterior presenta dos inconvenientes. El primero de ellos es que dicho modelo requiere una gran cantidad de parámetros de entrenamiento (16833 concretamente). Esto hace que el entrenamiento se convierta en un proceso largo y lento.

El segundo inconveniente es que con esta solución realmente no se está comprimiendo la imagen al ser codificada, ya que ésta tiene de tamaño 25x25x16. En este caso, aunque se cumple el objetivo de la aplicación de eliminar el ruido de las imágenes de Rayos X, no se cumple el principio de un auto-codificador cuya seña de identidad es precisamente esa "codificación", es decir, tener una imagen de menor tamaño que la entrada que recoja los datos más importantes.

Debido a esto, se ha probado a reducir esa capa capa codificada, de forma que la imagen obtenida ahí sea de 25x25x1. El modelo sigue siendo exactamente igual, solo que se ha cambiado la capa de codificación, en la que en lugar de tener 16 "filtros" se tiene solo uno, al igual que las entradas y salidas del auto-codificador.

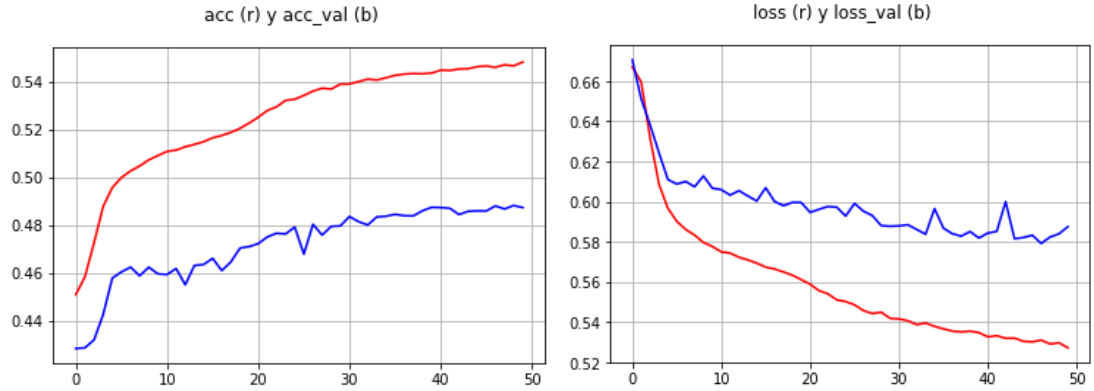

Figura 6-13. Resultados "accuracy" y "loss" de solución con imagen codificada de tamaño inferior

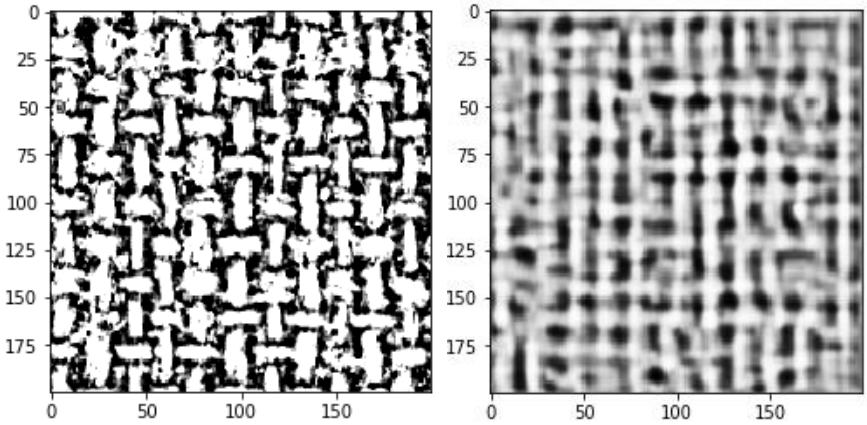

Figura 6-14. Resultados para solución con reduciendo capa codificada. Izquierda entrada, derecha salida

El problema que presenta es que la salida tras la reconstrucción no es demasiado buena, observándose zonas demasiado borrosas, que no permiten distinguir nada en claro de la imagen, y mucho menos permitir realizar un conteo de hilos.

Se han probado diversas arquitecturas intentando encontrar un modelo de este tipo que obtenga buenos resultados, pero sin éxito. Es por ello que se ha decidido descartar esta "compresión" para la aplicación en concreto de los Rayos X en estudio, y continuar con el modelo de auto-codificador presentado en la sección anterior.

6.4 Solución con PCA

Como se ha comentado en los primeros capítulos de esta memoria, una de las propiedades de un autoencoder es la reducción de la dimensionalidad del espacio de características. Se ha comprobado que aplicado a las imágenes de Rayos X de los cuadros si se fuerza esta propiedad no llega a tomar toda la información que se requiere y se obtienen malos resultados.

Ahora se va a comparar eso con una de las soluciones más empleadas cuando se emplean términos de reducción de dimensionalidad, que es PCA. Para ello, se ha utilizado en Python la librería *scikit-learn*, que incluye funciones para realizar dicha "codificación" de manera muy sencilla.

Para ello, se ha realizado dos bloques diferentes a nivel de código: por un lado se ha tomado una imagen y se ha entrenado el modelo en varias ocasiones sobre la misma variando el número de componentes empleado en el algoritmo; por otro lado se ha simulado el mismo entorno que para el auto-codificador, entrenando el modelo de PCA para una serie de recortes diversos y luego aplicar dicha solución sobre una imagen completa.

6.4.1 Análisis de algoritmo aplicado sobre una imagen

La idea del primer bloque diseñado para PCA es ver cómo se comporta el algoritmo sobre un solo recorte y qué resultados se obtienen variando el número de componentes. Este código se encuentra en el fichero "*prueba_1imagen_pca.py*".

Simplificando, los comandos principales para aplicar este algoritmo son los que se muestran a continuación, en los que se crea el modelo de PCA, se entrena y se aplica a la imagen (*img_pca*). Con esto, tendremos una imagen comprimida con PCA y para verla de nuevo restauramos de nuevo haciendo la transformada inversa (*img_restored*):

```
from sklearn.decomposition import PCA
...
pca = PCA(n_components = n_comp)
pca.fit(img)
img_pca = pca.fit_transform(img)
img_restored = pca.inverse_transform(img_pca)
```

Se ha tomado de nuevo un recorte de 200x200 del cuadro *Mercurio y Argos* de Diego Velázquez (*p01175p02xf2017*), y se ha aplicado el algoritmo realizando la codificación con diferentes ratios de compresión, como se muestra en la tabla, en la que también se refleja la varianza retenida obtenida tras la prueba realizada con el fragmento indicado:

Tabla 6–2. Valores y parámetros para PCA en un solo recorte

Ratio de compresión	Nº de componentes	Varianza retenida obtenida
2.5%	5	91.4868545847%
12.5%	25	96.8356891478%
25%	50	98.1391487502%
37.5%	75	98.7010269617%

En las siguientes figuras se ha representado la imagen original, la imagen tras ser codificada con PCA (*img_pca*) y la imagen reconstruida (*img_restored*), para cada uno de los ratios especificados en la tabla.

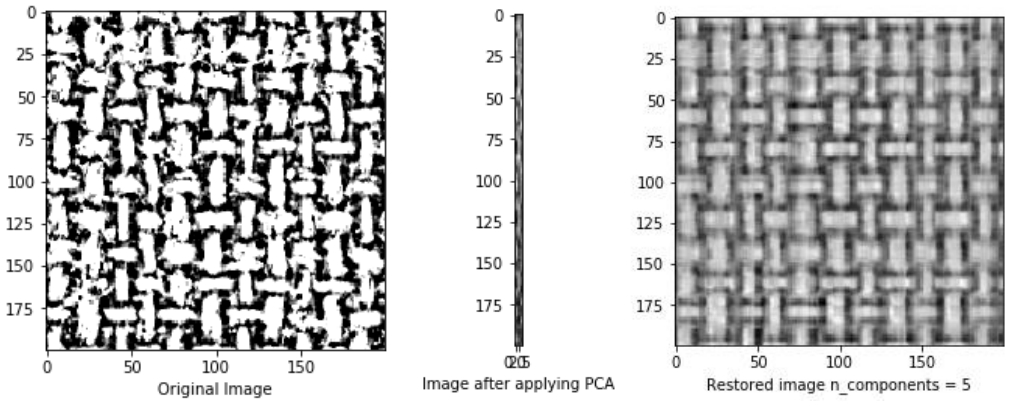

Figura 6-15. Solución con PCA para un ratio de compresión del 2.5% (5 componentes)

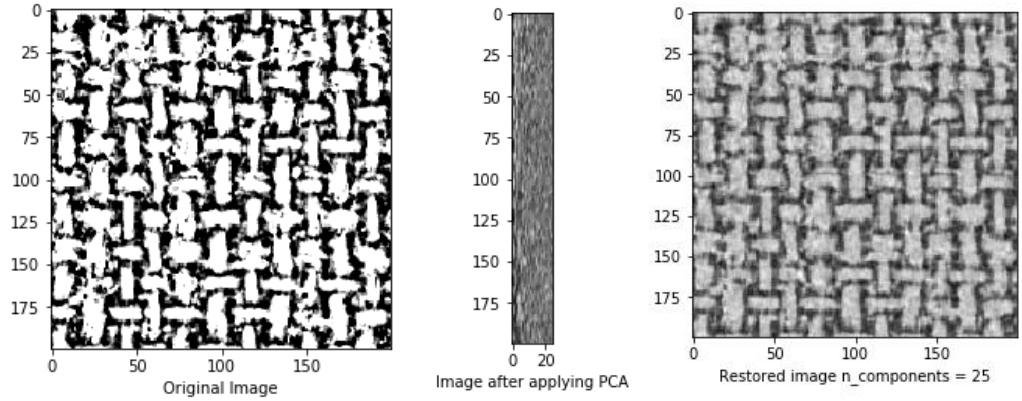

Figura 6-16. Solución con PCA para un ratio de compresión del 12.5% (25 componentes)

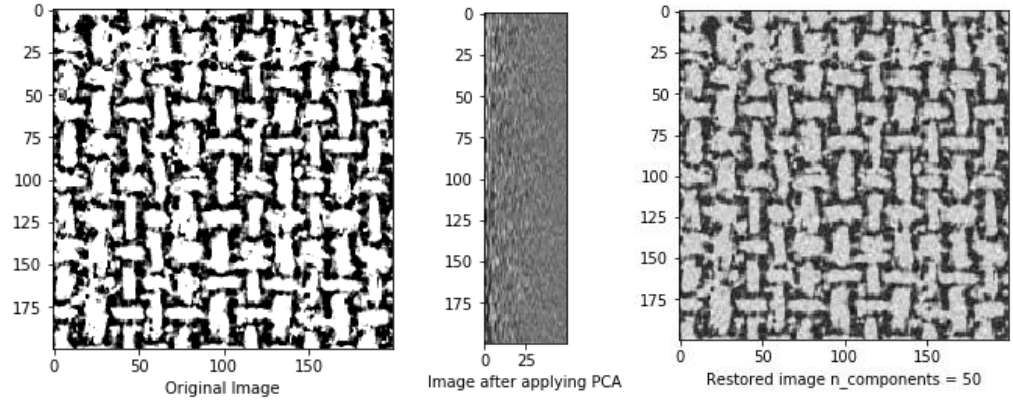

Figura 6-17. Solución con PCA para un ratio de compresión del 25% (50 componentes)

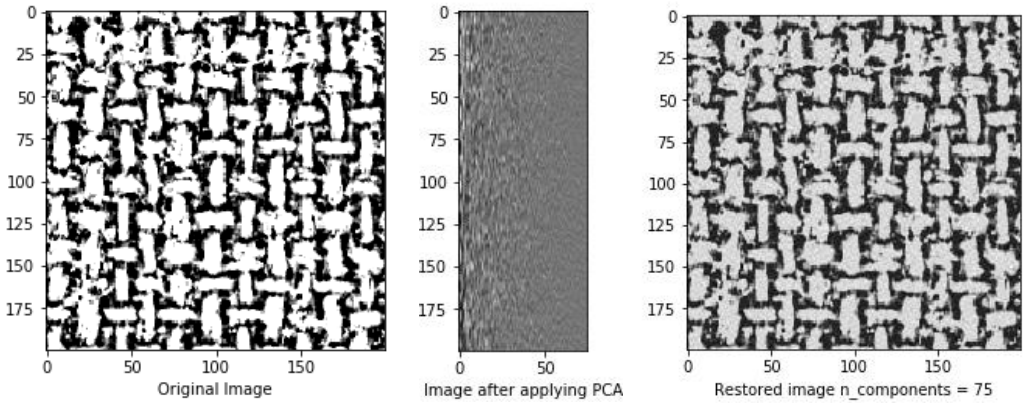

Figura 6-18. Solución con PCA para un ratio de compresión del 37.5% (75 componentes)

Con las salidas obtenidas se puede concluir que cuanto mayor sea el número de componentes empleados en PCA, mayor será la exactitud y la calidad de la imagen reconstruida con respecto a la original. Sin embargo, cuanto menor es el número de componentes, mayor es la compresión.

6.4.2 Aplicación de PCA sobre imágenes completas: entrenamiento y ejecución

De las pruebas anteriores parte el código desarrollado para esta aplicación, que sigue la misma estructura diseñada para el caso del auto-codificador, con una función para el entrenamiento (*entrenamientoPCA_rayosx.py*) y otra para la ejecución y reconstrucción de imagen a la salida del algoritmo (*ejecucionPCA_rayosx.py*).

La función de entrenamiento toma los mismos recortes empleados para entrenar el auto-codificador, y con ellas entrena el algoritmo. Tal y como se mostró en el caso del auto-codificador, las imágenes se leían del directorio y se almacenaban en una lista. Pero el algoritmo de PCA no admite matrices de tres dimensiones, por lo que se tiene que redimensionar el conjunto de imágenes de la siguiente forma para poder realizar el entrenamiento (este *reshape* será posteriormente detallado en la Figura 6-19 y su implicación sobre el número de componentes a utilizar en el algoritmo):

```
for archivo in listdir(carpeta_entrenamiento):
       if archivo!="info_img_orig.txt":
           nombre_img=carpeta_entrenamiento + archivo;
           img = io.imread(nombre_img, as_grey=True);
           img = img/255;
           imagenes.append(img);
           cuenta=cuenta+1;
   width=img.shape[0];

   imagenes=np.asarray(imagenes);
   imagenes_aux=imagenes.reshape(imagenes.shape[0],width*width)
```

Para el guardado del modelo de PCA entrenado, se utiliza la librería "*pickle*", realizando un *dump* del modelo entrenado en la ruta especificada:

```
   pca = PCA(n_components = n_comp)
   pca.fit(imagenes_aux)
   pickle.dump(pca, open(ruta_modelo, 'wb'))
```

Los parámetros para ejecutar la función de entrenamiento son los que se recogen en la siguiente tabla:

Tabla 6–3. Parámetros de la función *entrenamientoPCA_rayosx*

Nombre parámetro	Descripción
carpeta_entrenamiento	Directorio donde se encuentran los recortes de entrada (p.ej.: muestras varias)
ruta_modelo	Ruta del fichero donde se guardará el modelo tras entrenar PCA
num_componentes	Número de componentes con el que se entrenará el modelo PCA
pruebas_visualizacion	Variable binaria para indicar si se quieren ver las pruebas de visualización o pasar directamente al entrenamiento

Se muestra ahora un ejemplo de llamada a esta función de forma independiente (sin necesidad de utilizar el "main") con unos parámetros de prueba:

```
entrenamientoPCA_rayosx("C:/Users/FJMC/Desktop/TFM/RayosX/muestras_varias/",
                        "C:/Users/FJMC/Desktop/TFM/pca_model",
                        1000,
                        1)
```

El entrenamiento de un gran número de imágenes como ocurre en este caso hace que el número de componentes empleados para la solución sea diferente al empleado previamente para una única imagen de 200x200, viéndose aumentado. Esto se debe a que estamos apilando las imágenes creando una sola para que sea entrada del algoritmo.

En la siguiente figura se explica con un ejemplo de 2100 recortes de 200x200 y empleando 1000 componentes en el algoritmo de PCA cómo se comprime la imagen, y la correspondencia si sacamos una imagen de 200x200 (que escalando se reduciría el número de componentes a 5):

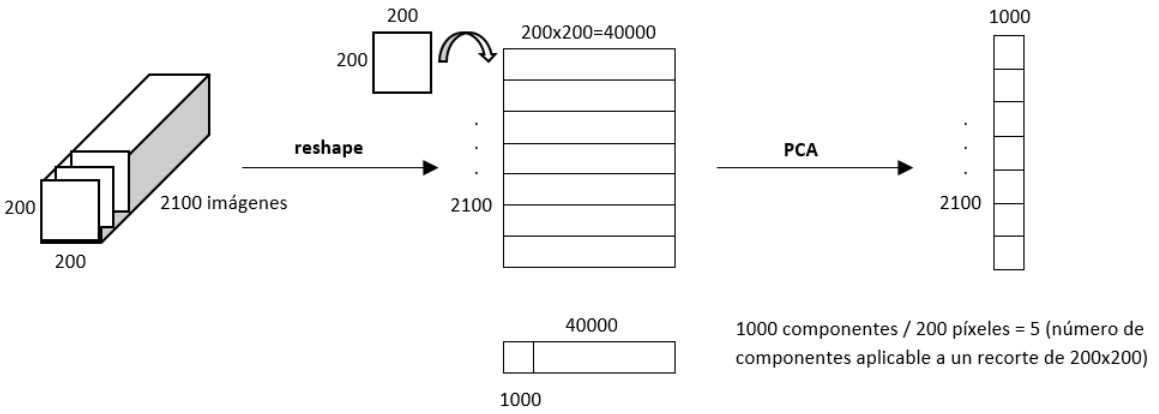

Figura 6-19. Ejemplo de aplicación de PCA sobre conjunto de imágenes de entrenamiento

A continuación se muestra una comparativa de salidas con distintos números de componentes en este entrenamiento para un recorte concreto. Estas pruebas pueden verse si se activa la variable binaria "*pruebas_visualizacion*". Como se puede observar, a partir de cierto número de componentes (3000 en este caso), el algoritmo satura, no mejorando los resultados.

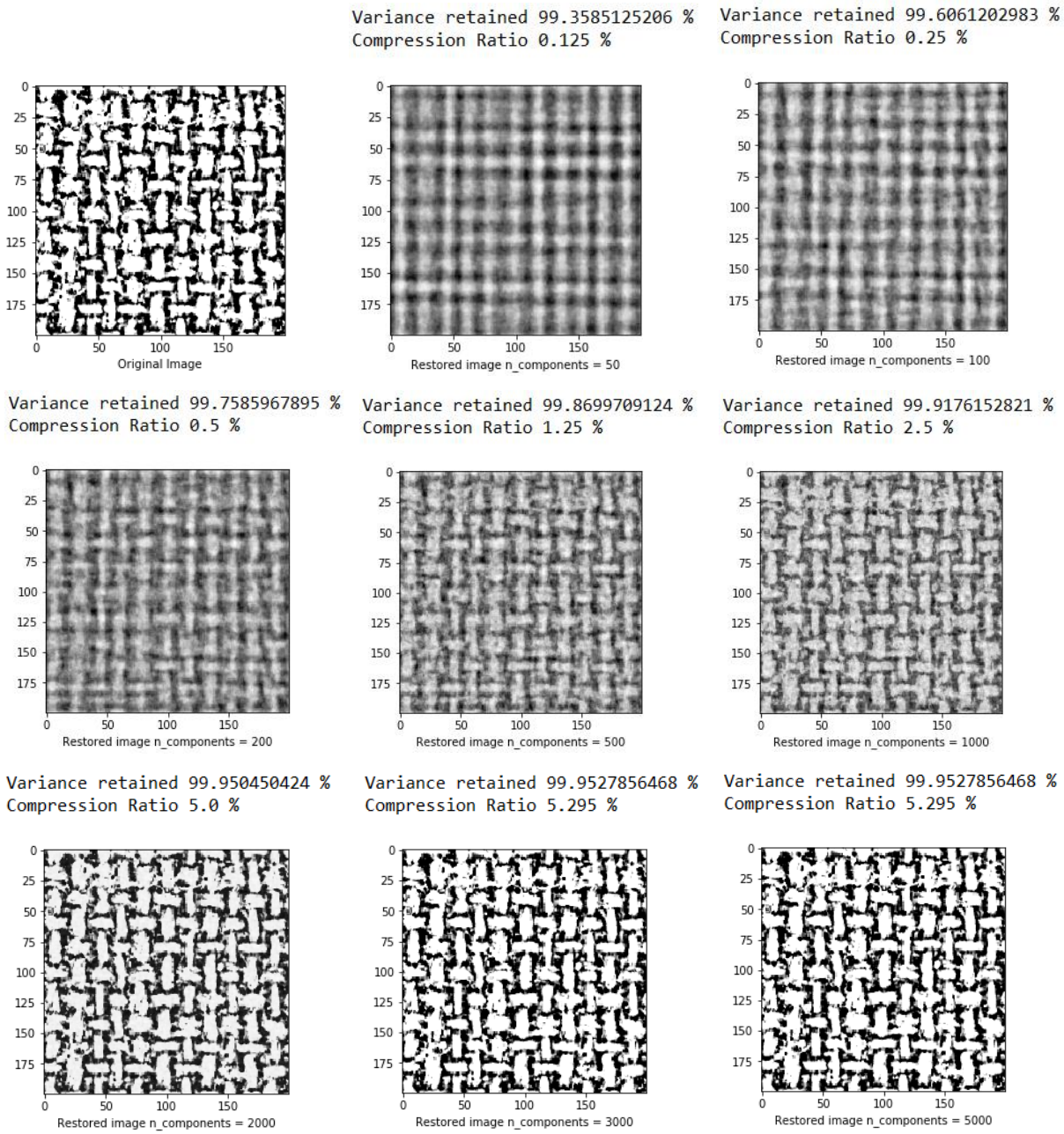

Figura 6-20. Comparativa PCA con distinto nº componentes tras entrenamiento con diferentes imágenes

Una vez entrenado el modelo, se procede a aplicar dicho modelo sobre todos los recortes que conforman una imagen completa, para la posterior reconstrucción de la imagen con los recortes de salida.

En este caso, se vuelve a emplear la librería "*pickle*" para cargar el modelo previamente guardado, y aplicarlo sobre el nuevo conjunto de imágenes:

```
# Ya el modelo está entrenado, aquí empleamos el modelo PCA generado
pca = pickle.load(open(modelo_PCA, 'rb'))
imgs_test_pca = pca.fit_transform(imagenes_test_aux)
img_test_restored = pca.inverse_transform(imgs_test_pca)
```

Esta función denominada '*ejecucionPCA_rayosx*', necesita que se le pasen como parámetros los que se detallan en la siguiente tabla (coinciden con la función correspondiente en el caso del auto-codificador):

Tabla 6–4. Parámetros de la función *ejecucionPCA_rayosx*

Nombre parámetro	Descripción
carpeta	Directorio donde se encuentran los recortes de entrada de la imagen completa a reconstruir
tam_x	Tamaño x (ancho) de imagen original para reconstrucción
tam_y	Tamaño y (alto) de imagen original para reconstrucción
modelo_PCA	Ubicación del modelo PCA a utilizar (previamente entrenado)
ruta_salida	Ubicación donde se almacenará la imagen de salida
resolucion_salida	Resolución que tendrá la imagen reconstruida (coincidirá con la de la imagen original)

Un ejemplo de llamada a esta función es:

```
ejecucionPCA_rayosx("C:/Users/FJMC/Desktop/TFM/RayosX/F00205p01_EnSim/",
           10420,
           7923,
           "C:/Users/FJMC/Desktop/TFM/pca_model",
           "C:/Users/FJMC/Desktop/TFM/RayosX/recon/recon_PCA_F00205p01.tif",
           600)
```

Como ya se ha comentado en el sub-apartado 6.3.3, no se ha podido llegar a alcanzar la misma tasa de compresión con el AE que las que se llegan a conseguir con PCA, debido a que una reducción de la capa codificada en el auto-codificador implica una disminución de la calidad (presentando la imagen zonas bastante borrosas).

Es por ello que el criterio para elegir el número de componentes empleado para la solución de PCA que se utilizará en la comparativa ha sido por ser el que visualmente ha obtenido mejores resultados en general entre las 3 imágenes de Rayos X a comparar, además de ser el resultado intermedio obtenido de las pruebas realizadas. Este valor elegido es de 500 componentes, equivalente a un ratio de compresión de 1.25% sobre el total de imágenes entrenadas.

A partir de estas soluciones, se van a comparar en el siguiente capítulo los resultados entre el auto-codificador y el algoritmo de PCA de las imágenes de Rayos X completas reconstruidas.

7 RESULTADOS

Siempre parece imposible hasta que se hace.

- Nelson Mandela -

Con los modelos anteriores, se va a realizar un análisis de los resultados obtenidos para una serie de cuadros elegidos. La comparativa se va a hacer entre el modelo de auto-codificador adoptado y un modelo de PCA. En las salidas que se van a mostrar ya se tiene en cuenta el proceso posterior a la aplicación del algoritmo correspondiente en el que la imagen es reconstruida.

En concreto, serán tres obras las elegidas como ejemplo para este estudio, cada una de un pintor diferente de forma que se puedan analizar las distinciones entre ellas:

Tabla 7–1. Imágenes de Rayos X de cuadros empleados para mostrar sus resultados definitivos

Nombre imagen	Cuadro	Pintor	Cedida por
F00205p01	Retrato de un hombre viejo con barba	Vincent van Gogh	RKD
MNP07906a00xf	Dos racimos de uvas	Miguel de Pret	Museo Nacional del Prado
p01175p02xf2017	Mercurio y Argos	Diego Velázquez	Museo Nacional del Prado

La comparativa consta de tres imágenes o fragmentos a estudiar para cada uno de los cuadros anteriormente mencionados:

- Fragmento original previo a la aplicación del algoritmo. Este fragmento ya ha pasado por la fase del preprocesado de la imagen, sin llegar a aplicarse todavía los recortes de 200x200.
- Fragmento tras la reconstrucción para el modelo del auto-codificador. La solución que se ha tomado finalmente es la que se ha presentado la arquitectura y resultados en los apartados 6.3.1 y 6.3.2.
- Fragmento tras la reconstrucción para el modelo de PCA. Se ha tomado la solución de PCA con un número de componentes de 500, es decir, con un ratio del 1.25% sobre el conjunto de imágenes entrenadas, tal y como se expone en el apartado 6.4.2.

Dichos fragmentos se han tomado de tamaño 5x5 cm (1 cm equivale a 200 píxeles aproximadamente aunque depende de la resolución de la imagen) sobre las imágenes completas: bien sobre la original preprocesada o bien sobre las imágenes reconstruidas para cada uno de los dos modelos. El tamaño se ha elegido de esta

forma simplemente para una buena visualización en la memoria de los hilos sobre las imágenes, ya que si se presentan las imágenes completas resulta muy difícil de distinguir.

El objetivo es poder comparar la solución elegida del auto-codificador con la solución aplicando el algoritmo PCA y con la imagen original preprocesada. Es por ello, que además de la visualización directa sobre los fragmentos de 5x5 cm que se representan en los siguientes sub-apartados, se ha realizado un análisis de los resultados con el programa Aracne que se encarga de hacer un conteo de los hilos de los lienzos y de representar una serie de gráficas para su estudio.

Como entrada para el software Aracne, se han utilizado otros fragmentos de las mismas imágenes, esta vez de mayor tamaño para que el conteo pudiera realizarse en una superficie mayor. Es por ello que se han introducido fragmentos de 20x20cm. Esta diferencia con los fragmentos mostrados aquí en los siguientes apartados de la memoria (fragmentos de 5x5 cm) es simplemente una cuestión de que se puedan visualizar bien en este documento.

Del mismo modo, los resultados de Aracne se comparán sobre las mismas imágenes con los tres métodos anteriormente comentados: imagen original preprocesada (previa a aplicar alguno de los algoritmos), imagen reconstruida tras aplicar el auto-codificador e imagen reconstruida tras aplicar PCA.

Para obtener las estadísticas y resultados con Aracne, el programa toma la imagen de entrada y la divide en recortes de 2 cm a los que realiza la DFT, obteniendo el número de hilos como el máximo en la horizontal y el máximo en la vertical.

Para el conteo de hilos verticales y horizontales de cada imagen, se representa un mapa de colores que ofrece una visión local del número de hilos calculado en la imagen y representando dicho valor con un color representado en la escala de la derecha. Si la tela no cambia notablemente en la imagen, el número de hilos verticales por cm no debería variar a lo largo de la vertical (igualmente para la horizontal). Sin embargo, por motivos diversos, la tela no se observa con calidad suficiente en la imagen y la DFT da resultados con alta variabilidad. Se busca por tanto métodos lo más robustos posible que no den variaciones bruscas a lo largo de la imagen. El objetivo así es que, para el conteo de hilos verticales, u horizontales, el mapa sea lo más uniforme posible, es decir, que el color predominante sea el mismo y se eviten discontinuidades con otros colores (ya que eso significa que el número de hilos calculado en ese bloque varía notablemente con respecto al resto).

Por otro lado, se representa un histograma asociado a cada conteo de hilos verticales u horizontales, que ofrece una visión global calculando un conteo total de bloques con el mismo número de hilos. El objetivo es que el histograma esté centrado en un valor y sea lo más estrecho posible, evitando que se expanda hacia otros valores que serían dichas discontinuidades (por ejemplo debido a ruido o grietas en la imagen) encontradas en el conteo como se ha comentado anteriormente.

Por último se muestra una tabla con los valores medios y desviación estándar obtenidos durante el conteo de hilos para cada imagen. En este caso sobre todo interesa que los valores de desviación estándar sean lo más pequeño posible, ya que un valor grande indica que el conteo de hilos en los distintos bloques es muy diferente.

7.1 Análisis del primer cuadro: “Retrato de un hombre viejo con barba” de Vincent Van Gogh

Comparado con la imagen de entrada previa a aplicar cualquiera de los algoritmos, la salida del auto-codificador es capaz de eliminar de forma bastante correcta el ruido inicial, permitiendo una buena visualización de los hilos tanto verticales como horizontales.

Sin embargo, para PCA parece que el ruido de la imagen se mantiene, incluso que puede haberse visto un poco incrementado a la hora de realizar la compresión.

En el fragmento se puede observar abajo a la izquierda una imperfección, que ambos algoritmos intentan evitar de alguna forma, pero que no llegan a conseguir eliminarlo. Es en el modelo del auto-codificador donde mejor se observa este intento, intentando quitar el ruido que contiene en su interior, consiguiendo dejarlo más claro que en la imagen de entrada.

Como ya se comentó en capítulos anteriores, en la salida del auto-codificador se pueden apreciar los bordes de la unión de los bloques de 200x200. Esto no va a formar parte del análisis ya que no es objetivo de este proyecto, ya que dicha reconstrucción se ha realizado para una correcta visualización sobre la aplicación.

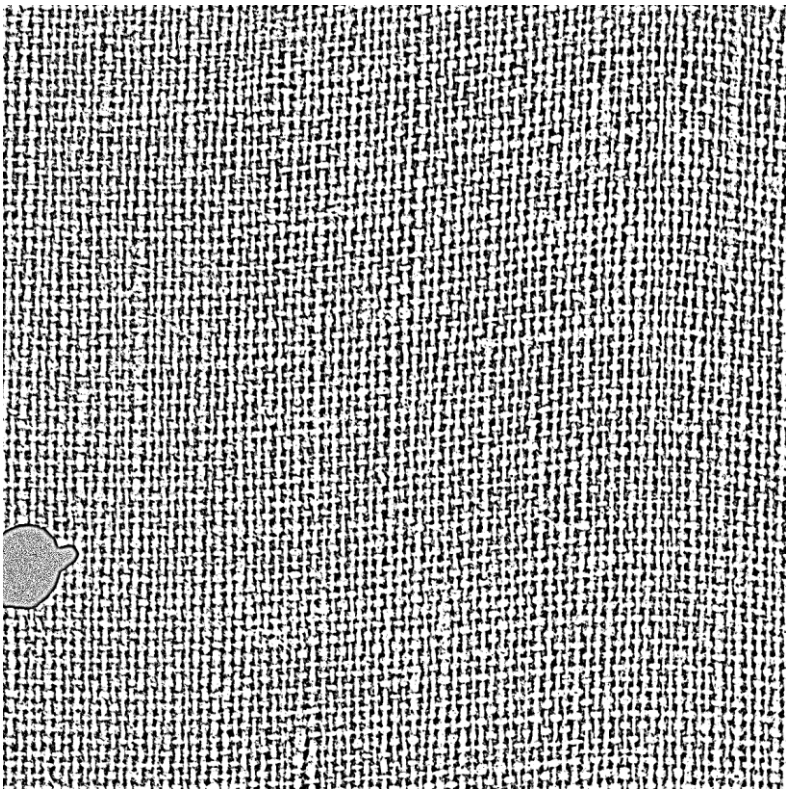

Figura 7-1. Fragmento de original preprocesada para cuadro 1 (*F00205p01*)

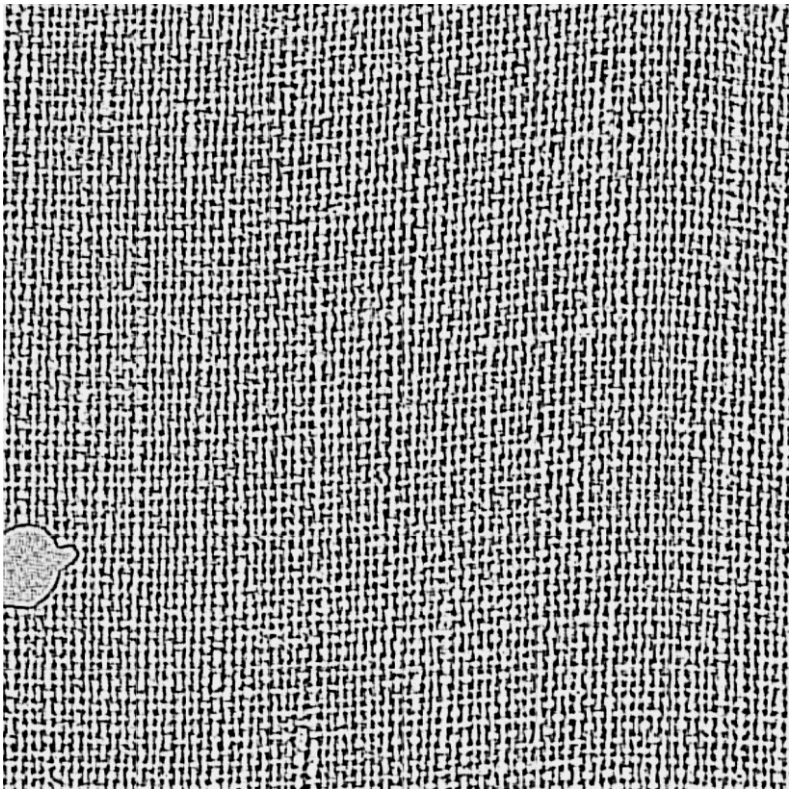

Figura 7-2. Fragmento de imagen reconstruida con auto-codificador para cuadro 1 (*F00205p01*)

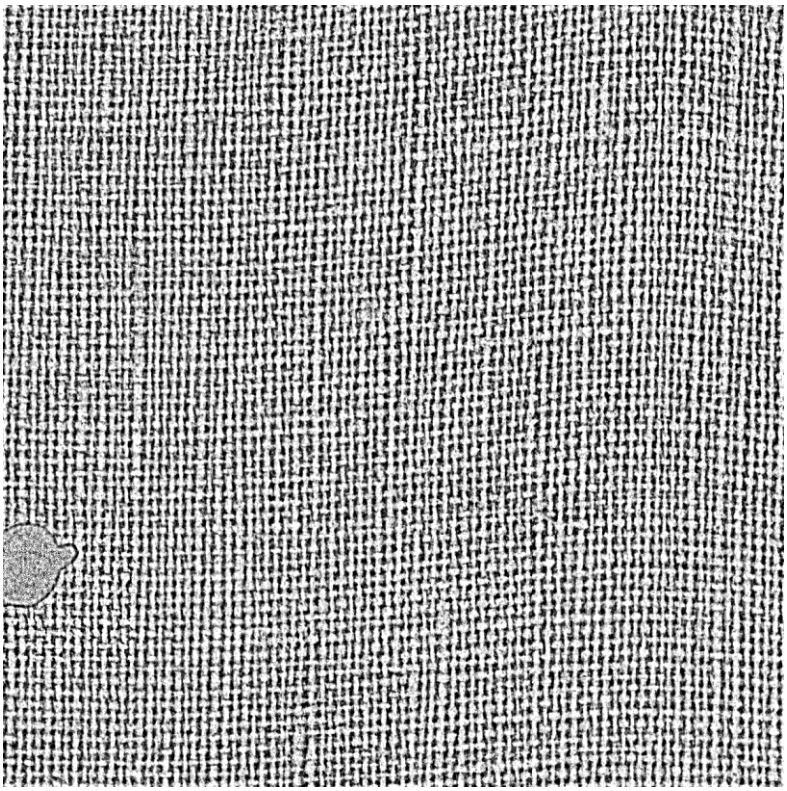

Figura 7-3. Fragmento de imagen reconstruida con PCA para cuadro 1 (*F00205p01*)

A continuación se detallan los resultados obtenidos tras analizar los fragmentos de este cuadro con Aracne:

- Conteo de hilos verticales:

En esta imagen de Van Gogh el programa de conteo Aracne aporta un resultado muy bueno y robusto, de forma que la diferencia entre los resultados de AE y PCA son parecidos entre sí y parecidos al resultado del programa, siendo mínimamente mejor los resultados con el auto-codificador, pero no distando demasiado de las demás (original y solución con PCA). En el mapa de colores no hay grandes diferencias de tonos ni discontinuidades, y en el histograma la mayoría de los valores giran en torno al número de bloques de 16 aproximadamente, existiendo algunos valores que se salen de ahí pero con valores muy pequeños.

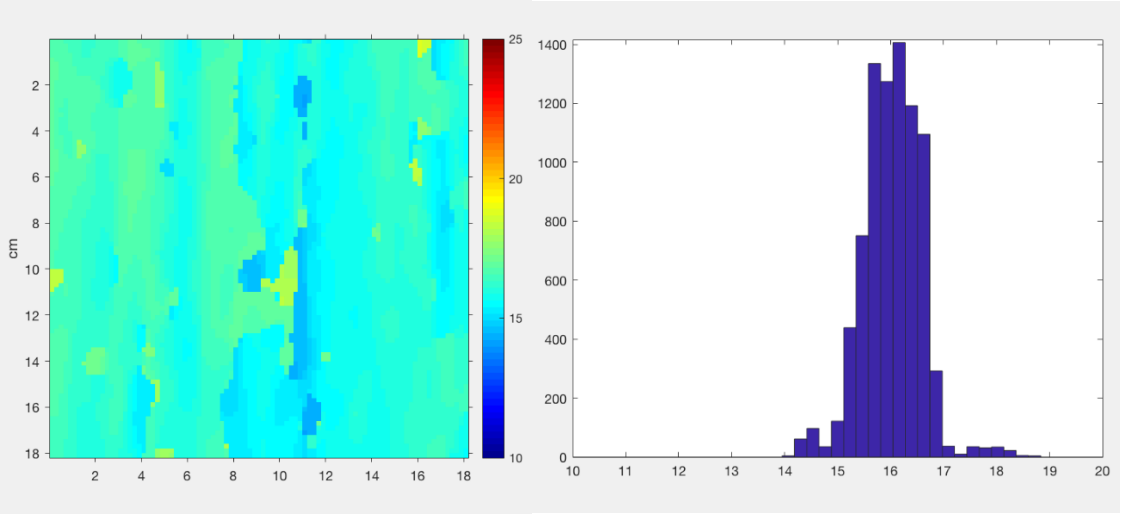

Figura 7-4. Mapa de colores e histograma de hilos verticales sobre imagen original de *F00205p01*

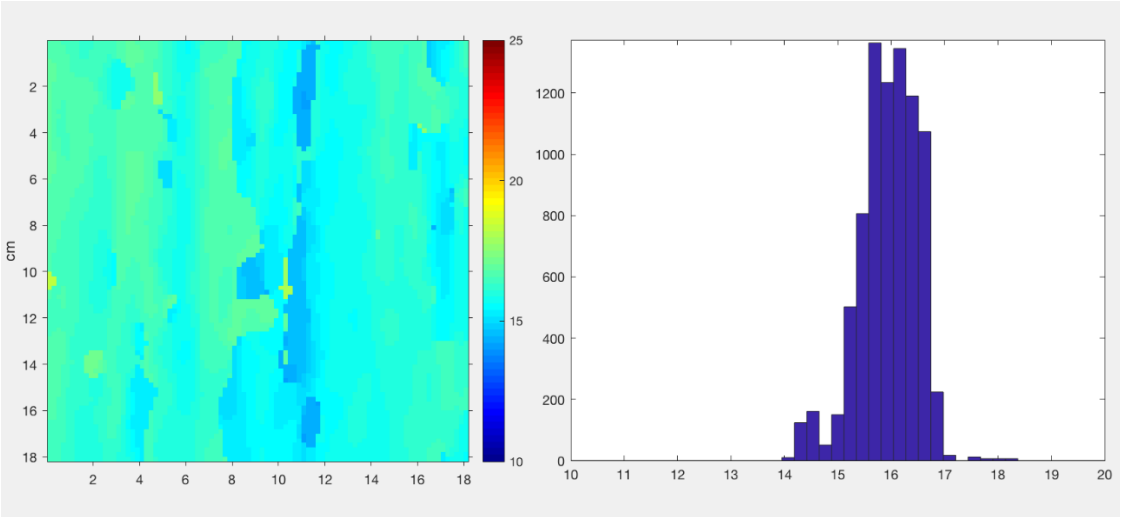

Figura 7-5. Mapa de colores e histograma de hilos verticales sobre salida con AE de *F00205p01*

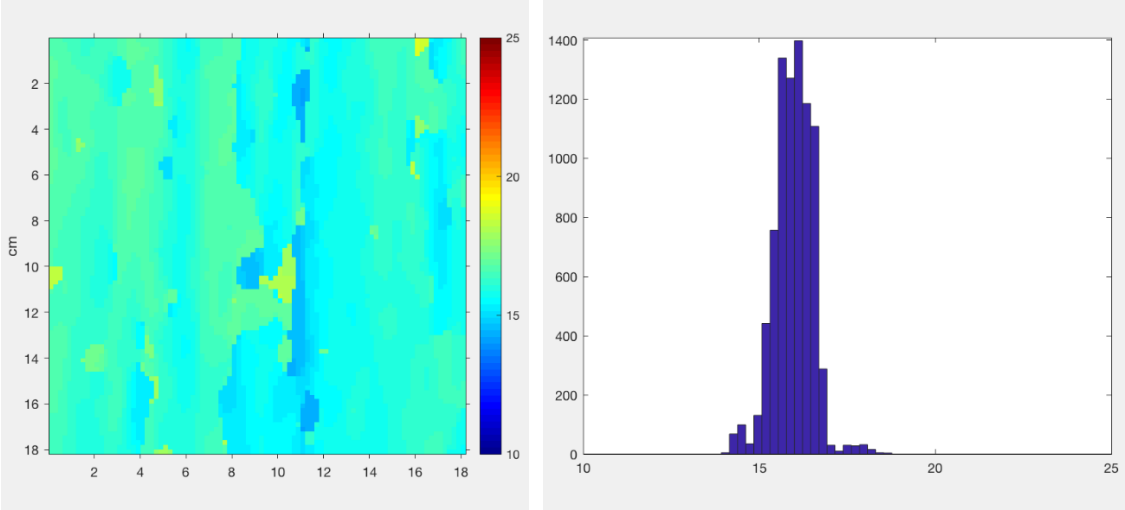

Figura 7-6. Mapa de colores e histograma de hilos verticales sobre salida con PCA de *F00205p01*

- Conteo de hilos horizontales:

En los resultados para los hilos horizontales, al igual que ocurre con los verticales los resultados son muy similares en ambas soluciones, no existiendo cambios notables ni en los mapas de colores ni en los histogramas.

Los tonos en el mapa de color son uniformes, predominando el azul que se corresponde con unos valores entre 13 y 14 hilos. En el histograma esto se ve que la gráfica de barras es bastante estrecha, lo cual es un buen resultado teniendo en cuenta que para la mayoría de los bloques el conteo calculado tiene valores semejantes.

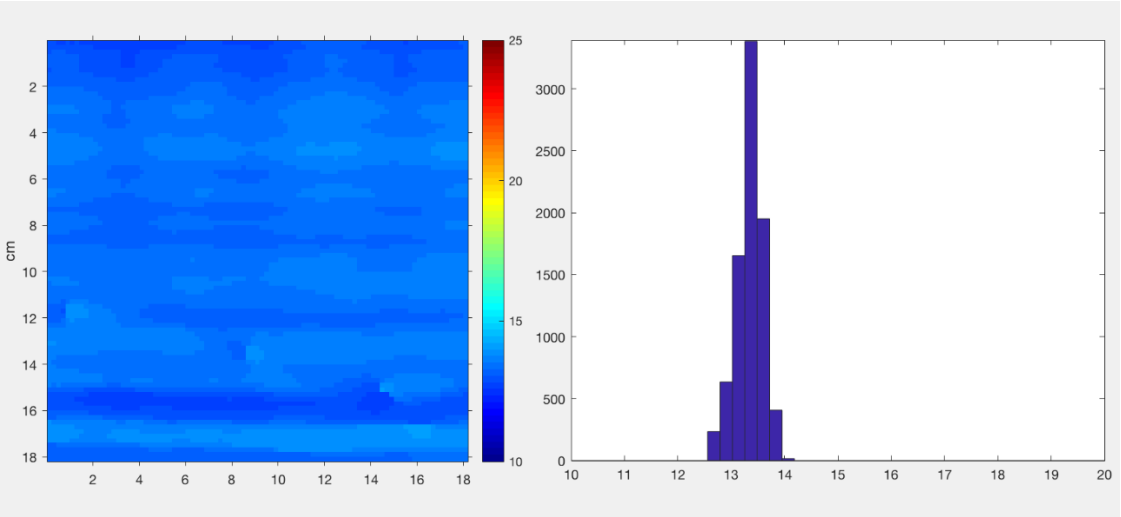

Figura 7-7. Mapa de colores e histograma de hilos horizontales sobre imagen original de *F00205p01*

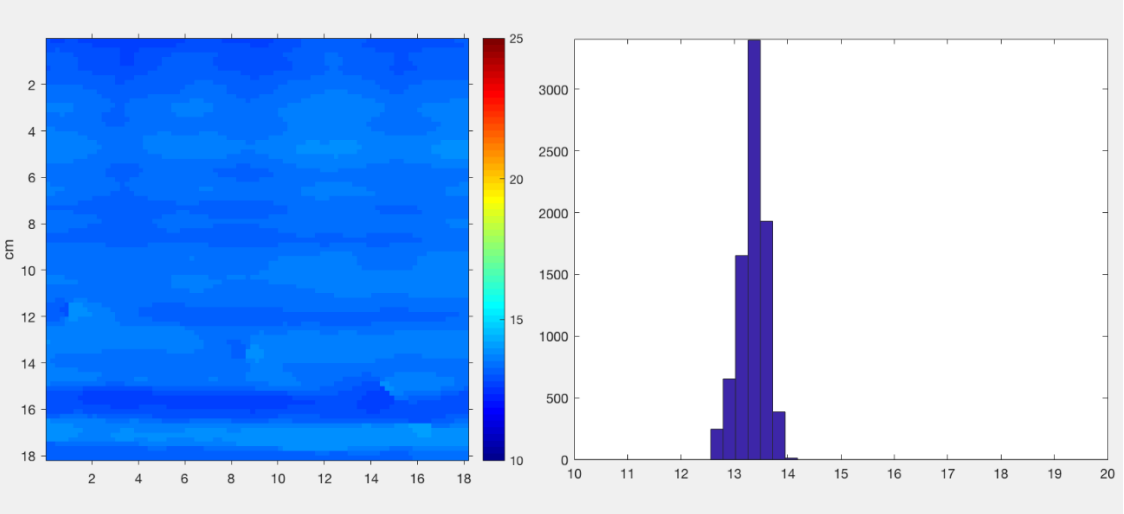

Figura 7-8. Mapa de colores e histograma de hilos horizontales sobre salida con AE de *F00205p01*

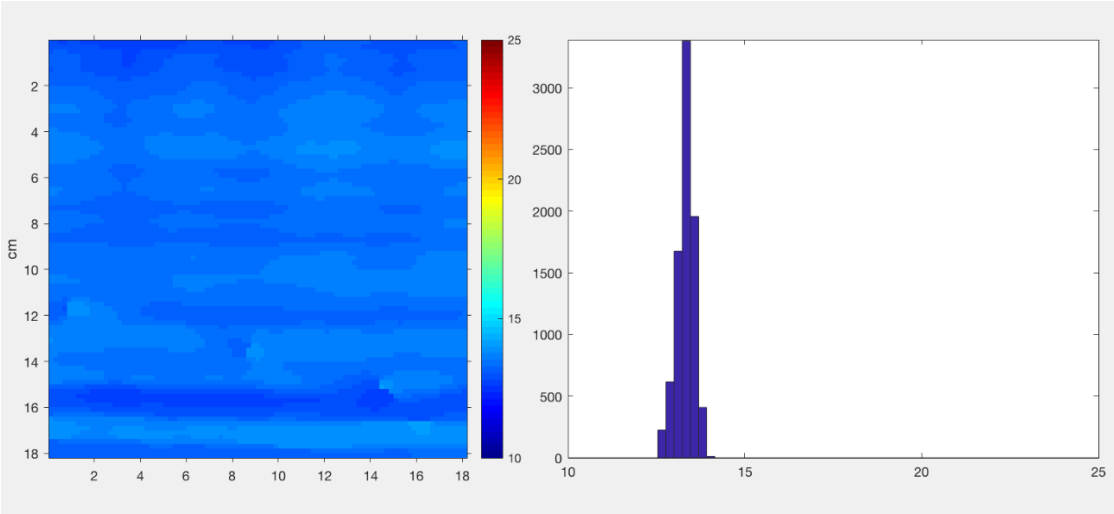

Figura 7-9. Mapa de colores e histograma de hilos horizontales sobre salida con PCA de *F00205p01*

- Comparación media y desviación estándar:

Como se puede observar, los valores en este caso varían muy poco o incluso nada como es el caso de la desviación estándar para los hilos horizontales. Las soluciones de auto-codificador y PCA poco mejoran la imagen original, que realmente no presenta tampoco grandes discontinuidades ni ruidos sobre el lienzo.

Tabla 7–2. Comparativa media y desviación estándar para imagen F00205p01

Conteo de hilos	Media			Desviación estándar		
	Img. Original	Salida AE	Salida PCA	Img. Original	Salida AE	Salida PCA
Verticales	16.03	15.95	16.02	0.56	0.55	0.55
Horizontales	13.37	13.36	13.37	0.25	0.25	0.25

7.2 Análisis del segundo cuadro: "Dos racimos de uvas" de Miguel de Pret

En este caso, la imagen presenta numerosas grietas que entorpece el conteo visual. A la salida del auto-codificador, se puede observar de nuevo la reducción de ruido, y sobre todo que se ha mitigado de manera muy correcta el problema de las grietas. Continúa siendo difícil el conteo de hilos ya que se presentan algunas zonas difusas a lo largo del fragmento (propiciadas en su mayoría por el intento de eliminar las grietas), sin embargo, para otras partes de la imagen en mejor estado seguramente con esta salida se pueda realizar un conteo o análisis del lienzo sin problemas.

Para la solución con PCA, se han mantenido la mayoría de imperfecciones que contiene el fragmento, siendo la salida muy similar a la entrada, es decir, sin grandes diferencias en términos de reducción del ruido.

Comparado con el cuadro anterior, el resultado esta vez es peor debido a que la imagen de entrada estaba en mal estado. Aun así, el AE intenta realizar un trabajo adecuado reduciendo el ruido en la medida de lo posible.

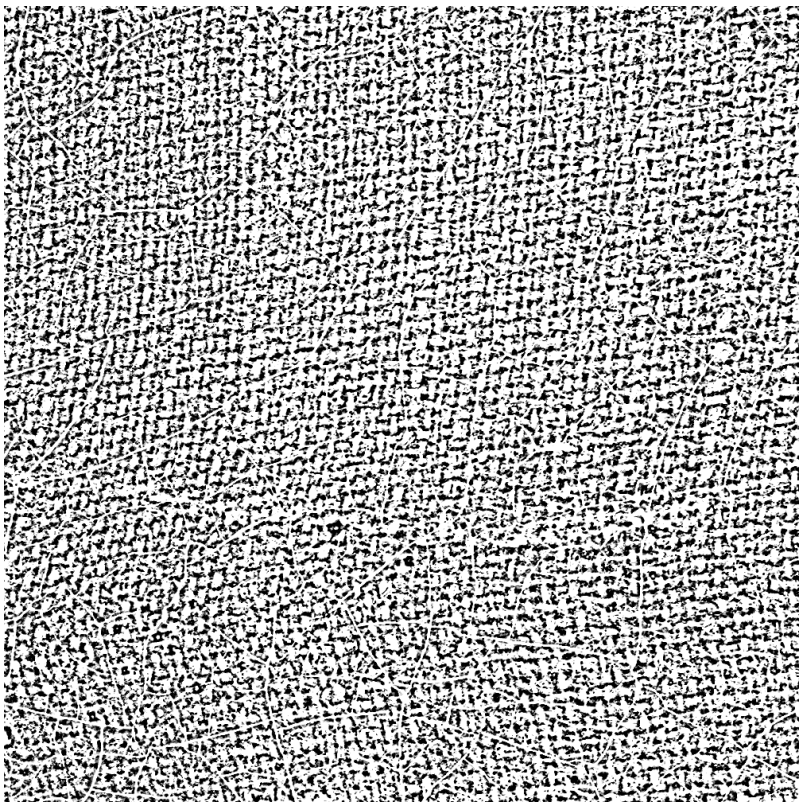

Figura 7-10. Fragmento de original preprocesada para cuadro 2 (*MNP07906a00xf*)

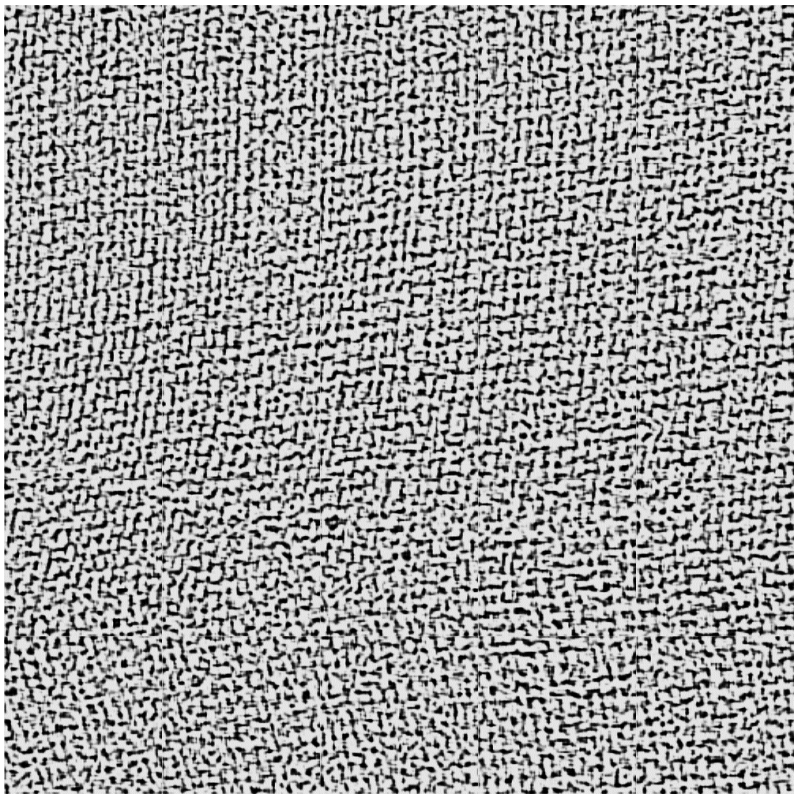

Figura 7-11. Fragmento de imagen reconstruida con auto-codificador para cuadro 2 (*MNP07906a00xf*)

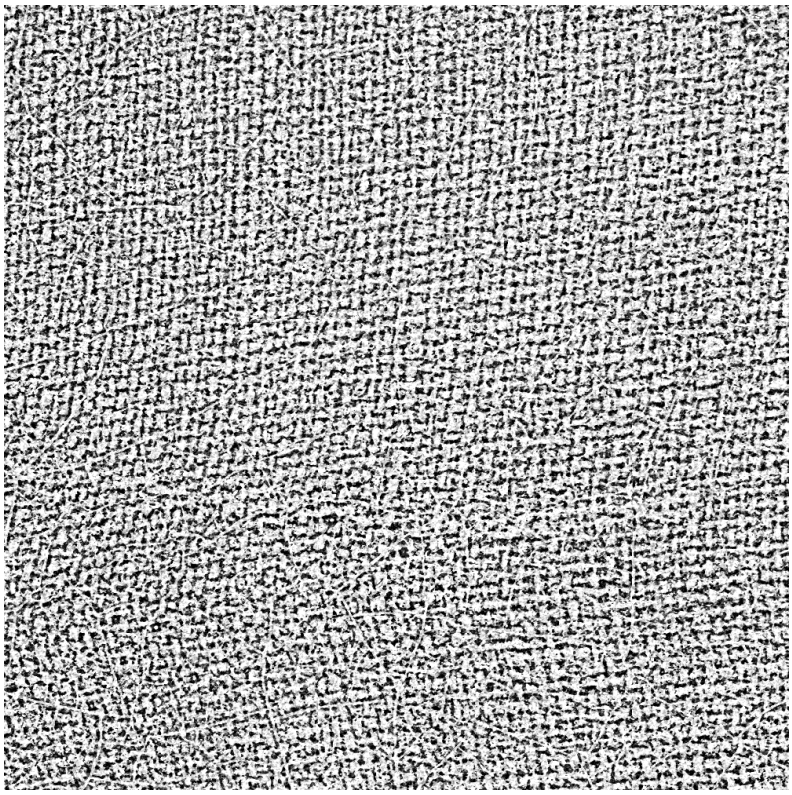

Figura 7-12. Fragmento de imagen reconstruida con PCA para cuadro 2 (*MNP07906a00xf*)

Se analizan ahora los resultados obtenidos tras pasar los fragmentos de este cuadro con Aracne:

- Conteo de hilos verticales:

En el fragmento elegido de esta imagen, se observan en el conteo de hilos verticales diferencias de colores sobre todo en el margen izquierdo, mostrando colores con tonos rojos y naranjas, lo que significa que tiene valores diferentes al resto de la imagen.

Para el caso del auto-codificador estas discontinuidades disminuyen notablemente, aunque no se refleje demasiado en el histograma se puede observar dicha mejoría en el mapa de colores. No ocurre lo mismo para PCA, que mantiene resultados similares a la imagen original.

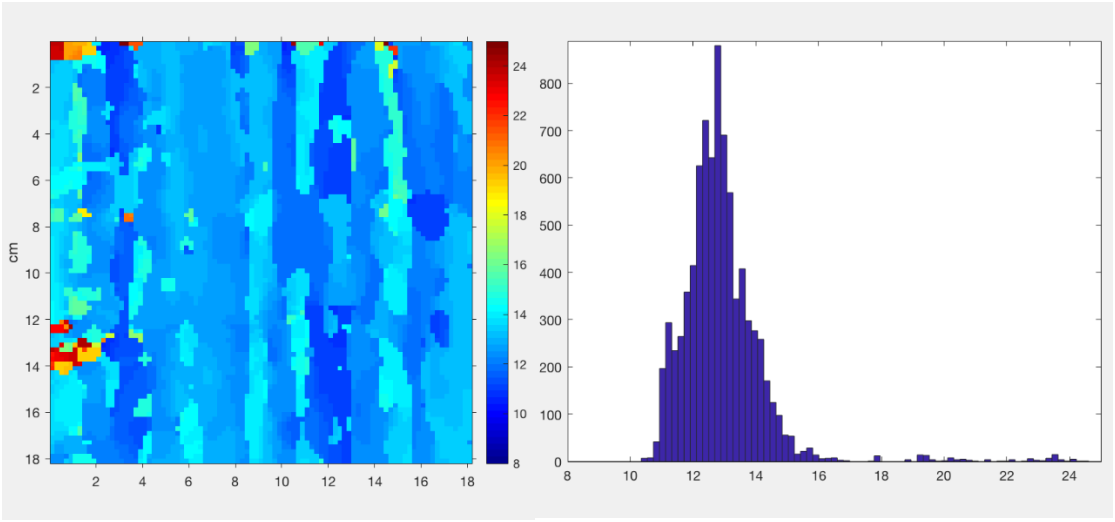

Figura 7-13. Mapa de colores e histograma de hilos verticales sobre imagen original de *MNP07906a00xf*

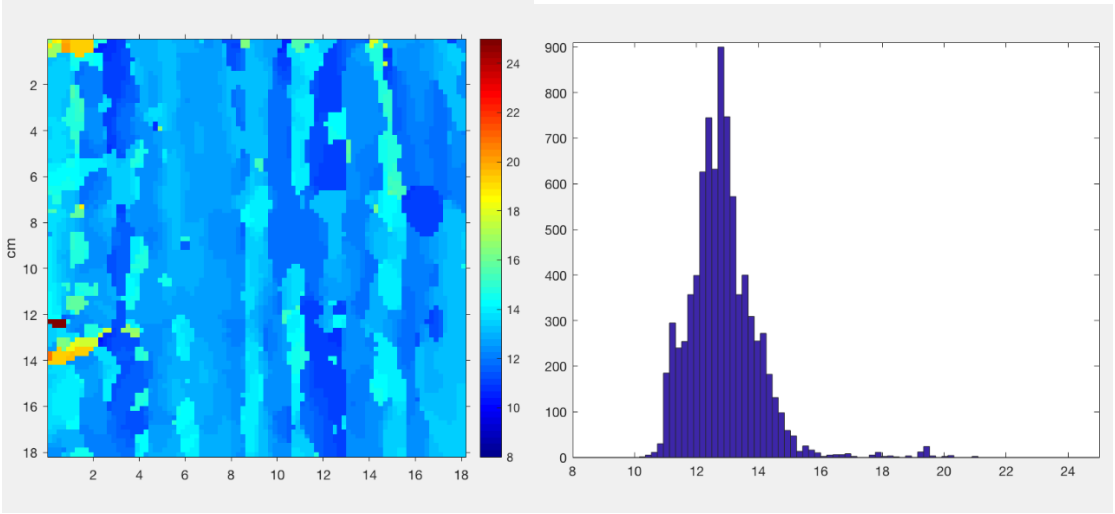

Figura 7-14. Mapa de colores e histograma de hilos verticales sobre salida con AE de *MNP07906a00xf*

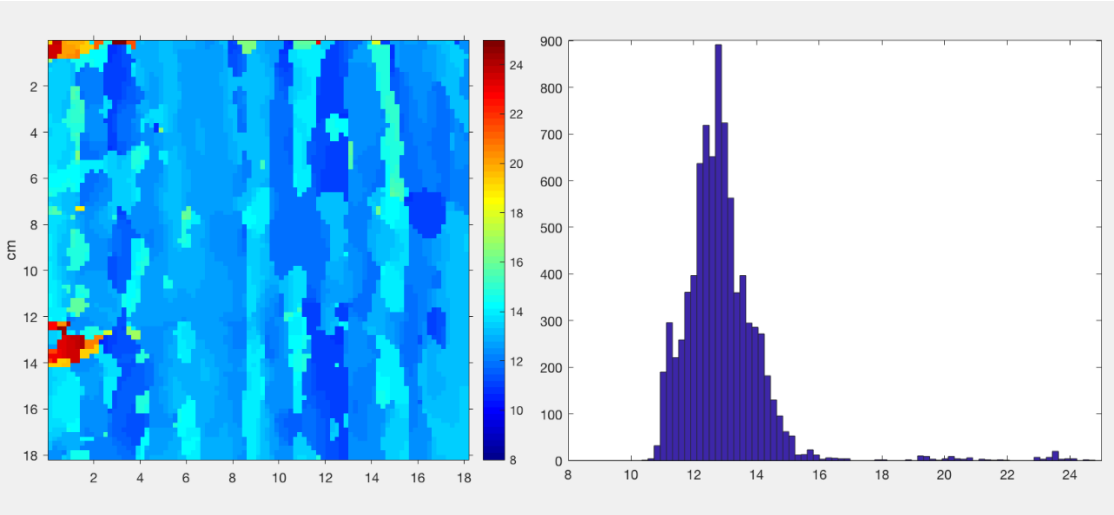

Figura 7-15. Mapa de colores e histograma de hilos verticales sobre salida con PCA de *MNP07906a00xf*

- Conteo de hilos horizontales:

Para los hilos horizontales en este caso se dispara el número de valores diferentes sobre el mapa de colores original, que se refleja también en el histograma con valores altos a la derecha de la gráfica que difieren del punto central donde deberían de concentrarse.

En el auto-codificador, estos valores se reducen, dando lugar a un mapa de colores mucho más uniforme (aunque sigue manteniendo grandes diferencias, no llega a ser una mejora muy grande). En el histograma también se puede observar que disminuyen los valores de la derecha, aunque no llegan a desaparecer del todo.

En PCA, al igual que para el caso de los hilos verticales, los resultados no distan demasiado de la imagen original.

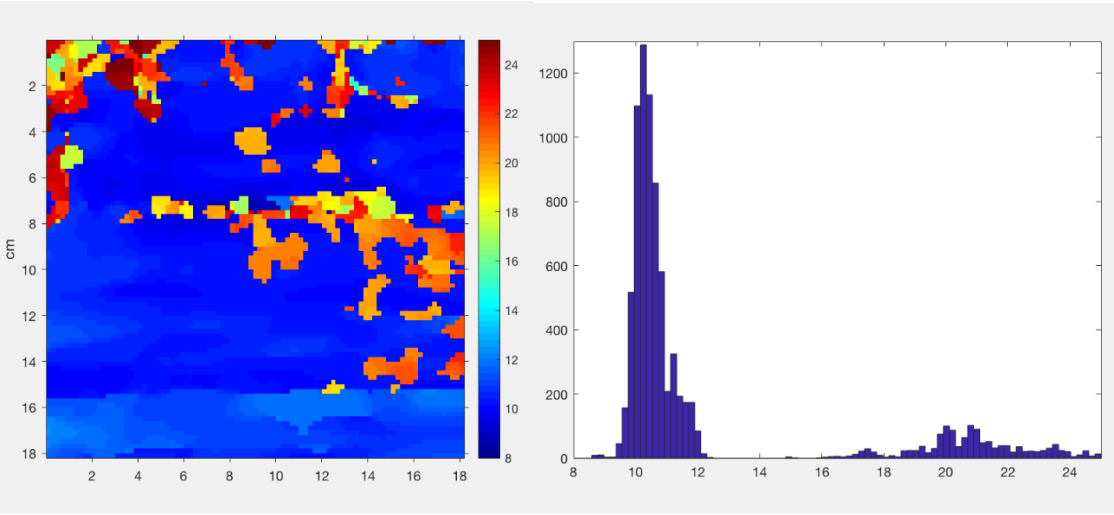

Figura 7-16. Mapa de colores e histograma de hilos horizontales sobre imagen original de *MNP07906a00xf*

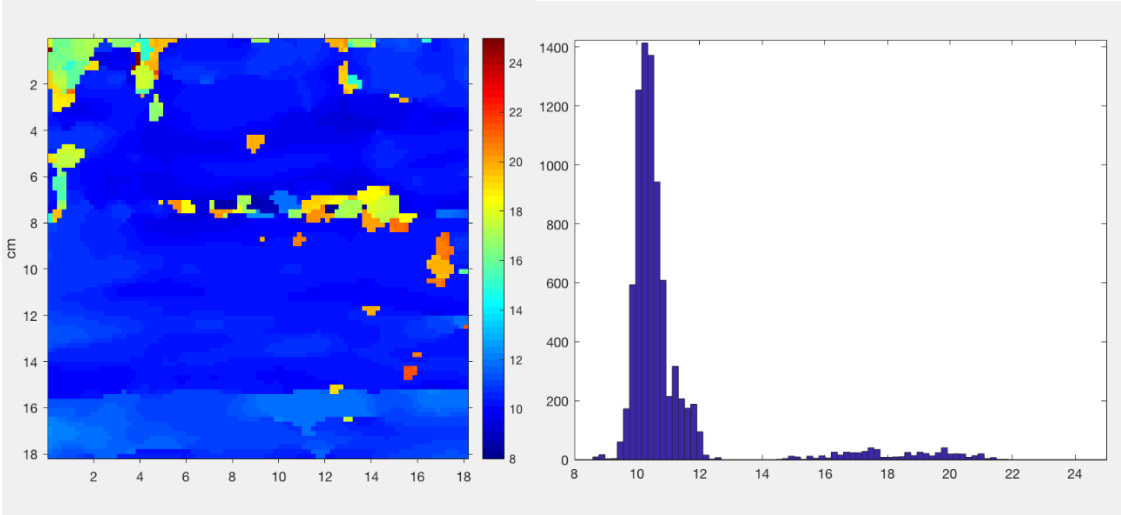

Figura 7-17. Mapa de colores e histograma de hilos horizontales sobre salida con AE de *MNP07906a00xf*

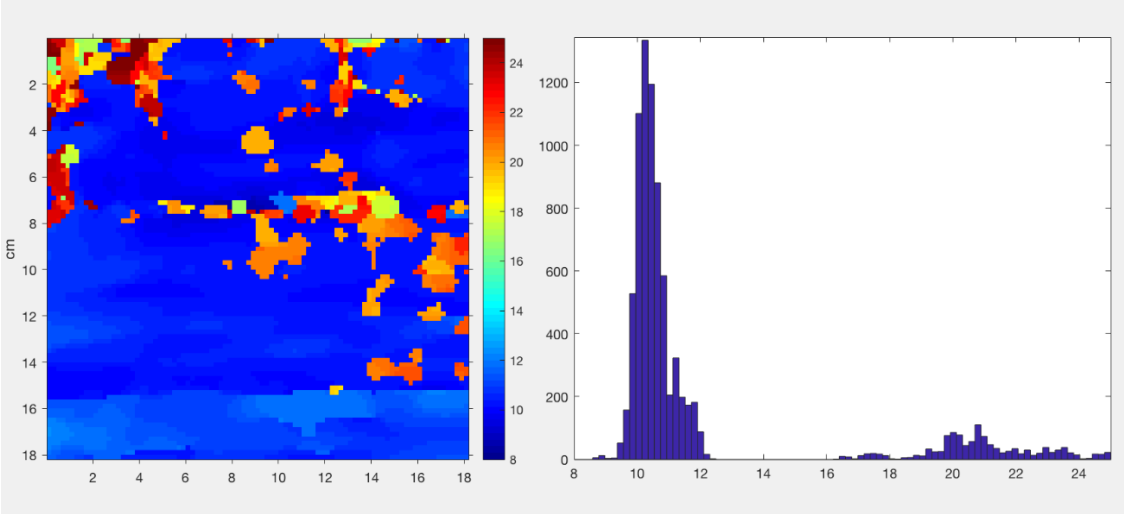

Figura 7-18. Mapa de colores e histograma de hilos horizontales sobre salida con PCA de *MNP07906a00xf*

- Comparación media y desviación estándar:

En la comparativa, poniendo el foco sobre la desviación estándar podemos ver que el auto-codificador es quien obtiene mejores resultados (los más pequeños) con respecto a la imagen original y a la resultante con PCA. La disminución de la desviación estándar es mucho más significativa en el caso del conteo de hilos horizontales, donde la bajada de la desviación ha sido de un 46.2% con respecto a la imagen original.

Tabla 7–3. Comparativa media y desviación estándar para imagen MNP07906a00xf

Conteo de hilos	Media			Desviación estándar		
	Img. Original	Salida AE	Salida PCA	Img. Original	Salida AE	Salida PCA
Verticales	12.86	12.79	12.86	1.43	1.12	1.42
Horizontales	12.16	10.98	11.97	3.94	2.12	3.78

7.3 Análisis del tercer cuadro: “Mercurio y Argos” de Diego Velázquez

Para esta obra de Velázquez, el fragmento obtenido presenta también numerosas imperfecciones, con un rasgado bastante notable en la parte central de la captura obtenida. De esta imagen es uno de los bloques de 200x200 que se ha tomado como referencia en el capítulo 6 para mostrar los resultados de cada una de las pruebas.

Ahora que se tiene una visión más completa de las salidas, se puede observar que para el caso del auto-codificador presenta buenos resultados en las partes que no tienen demasiado ruido. En las zonas en las que la imagen de entrada se vuelve mucho más irregular, el auto-codificador intenta mitigar el ruido pero eso propicia un difuminado de dicha zona en la imagen reconstruida. Esto son las áreas con un tono más claro que presenta el fragmento resultante (el resto se mantiene en un tono más oscuro), que como se puede ver no es posible distinguir en ellas los hilos. Por ello, en dichas zonas se vuelve muy complicado intentar realizar un análisis del tejido, pero al igual que se comentó para el caso anterior, es posible realizar el conteo en zonas más correctas, que no presenten tanto ruido.

En PCA, no mejora demasiado la situación, de hecho el ruido se mantiene casi de igual manera que en la entrada o incluso llegando a verse incrementado.

Comparado con los dos cuadros anteriores, se ha podido observar que a pesar de que los tejidos son diferentes, los resultados son similares, siendo más robusto al ruido el auto-codificador que el algoritmo de PCA. El primer cuadro ha sido el que mejor salida se ha obtenido, siendo también el que menores imperfecciones presentaba. Entre el segundo cuadro y el tercero, los resultados son muy parecidos, ya que lo que el auto-codificador es capaz de eliminar de ruido se ve en contraposición afectado por un difuminado de las zonas más deterioradas.

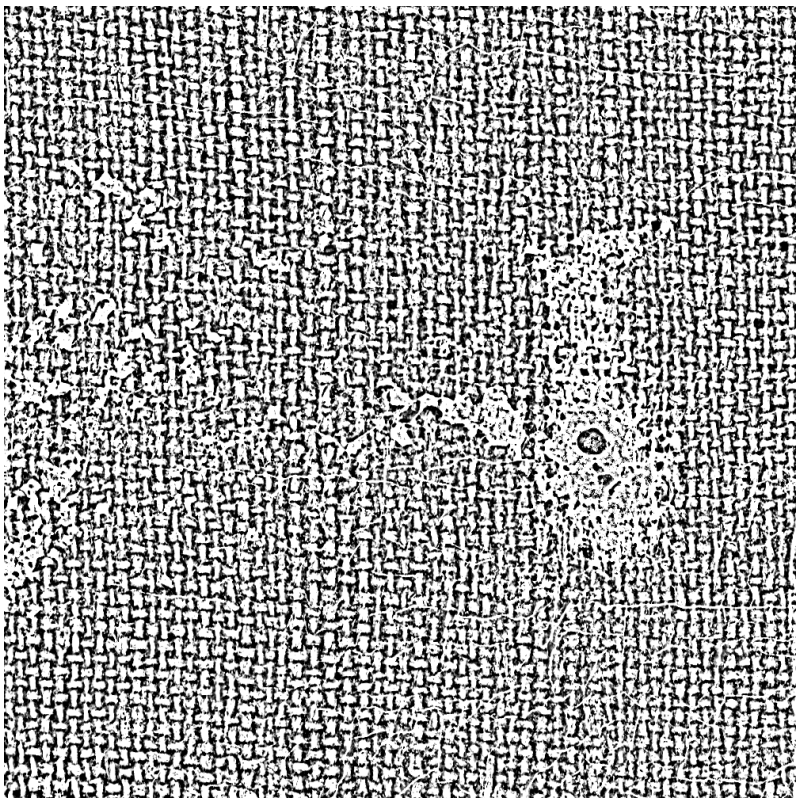

Figura 7-19. Fragmento de original preprocesada para cuadro 3 (*p01175p02xf2017*)

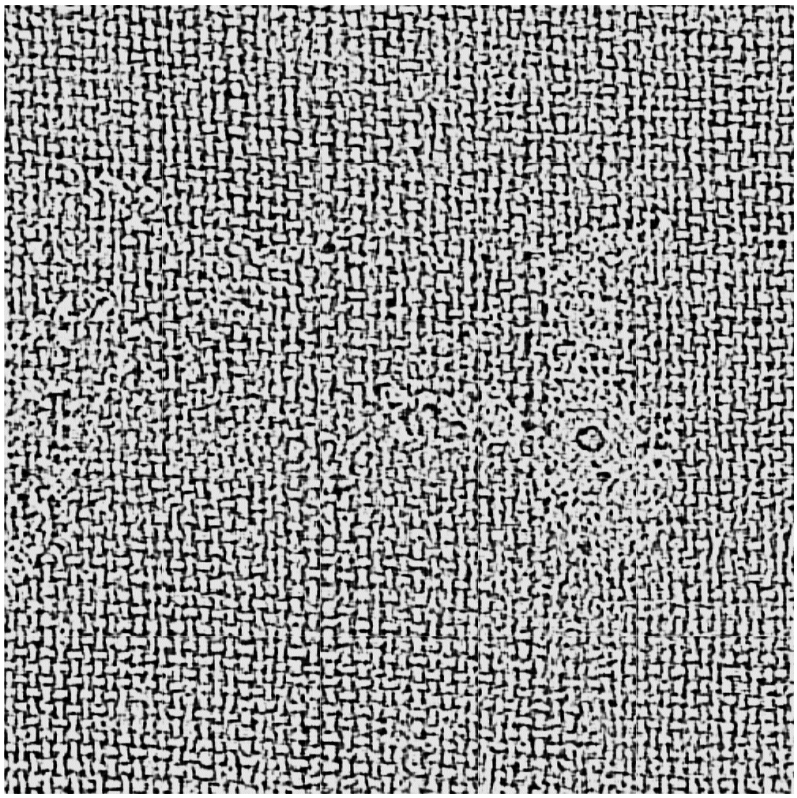

Figura 7-20. Fragmento de imagen reconstruida con auto-codificador para cuadro 3 (*p01175p02xf2017*)

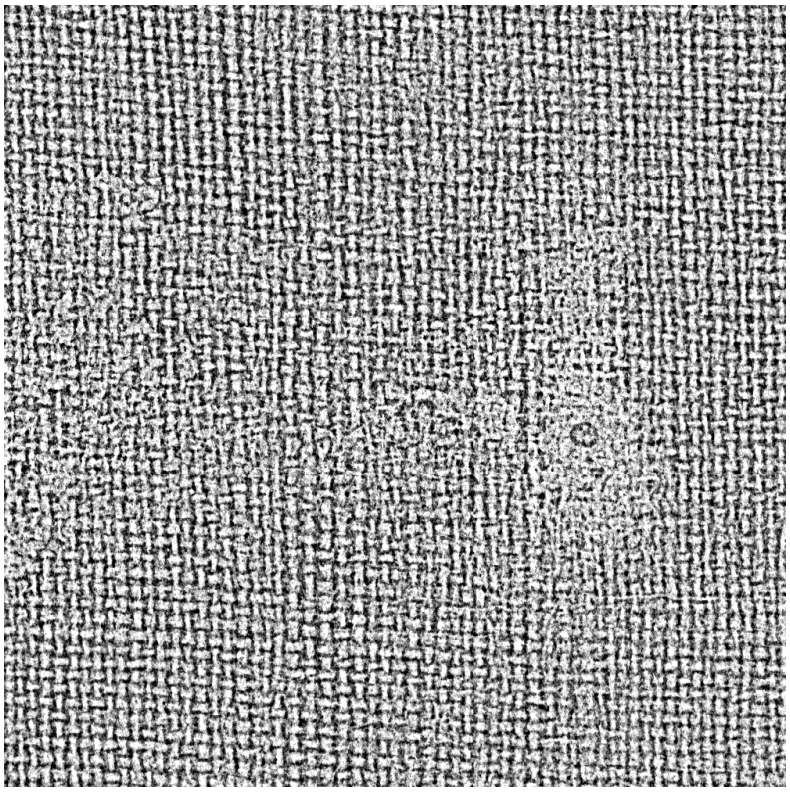

Figura 7-21. Fragmento de imagen reconstruida con PCA para cuadro 3 (*p01175p02xf2017*)

Con Aracne, los resultados obtenidos para este cuadro de Velázquez son los que se detallan a continuación:

- Conteo de hilos verticales:

Para esta imagen la diferencia entre la original y las salidas tras aplicar los distintos algoritmos tampoco llega a ser muy significativa. El auto-codificador sí que llega a eliminar algunos de los valores representados con colores rojos sobre el mapa de colores, pero no se ven muchas más diferencias a nivel de hilos verticales.

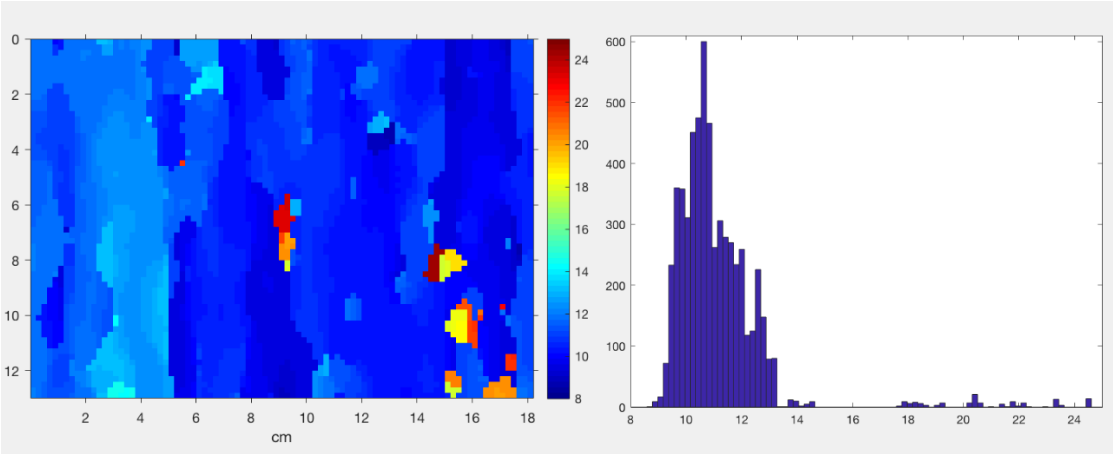

Figura 7-22. Mapa de colores e histograma de hilos verticales sobre imagen original de *p01175p02xf2017*

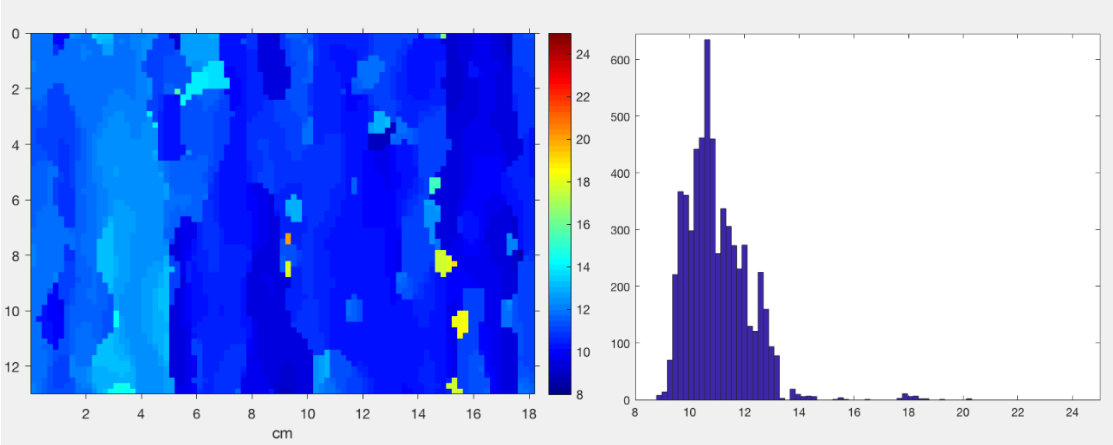

Figura 7-23. Mapa de colores e histograma de hilos verticales sobre salida con AE de *p01175p02xf2017*

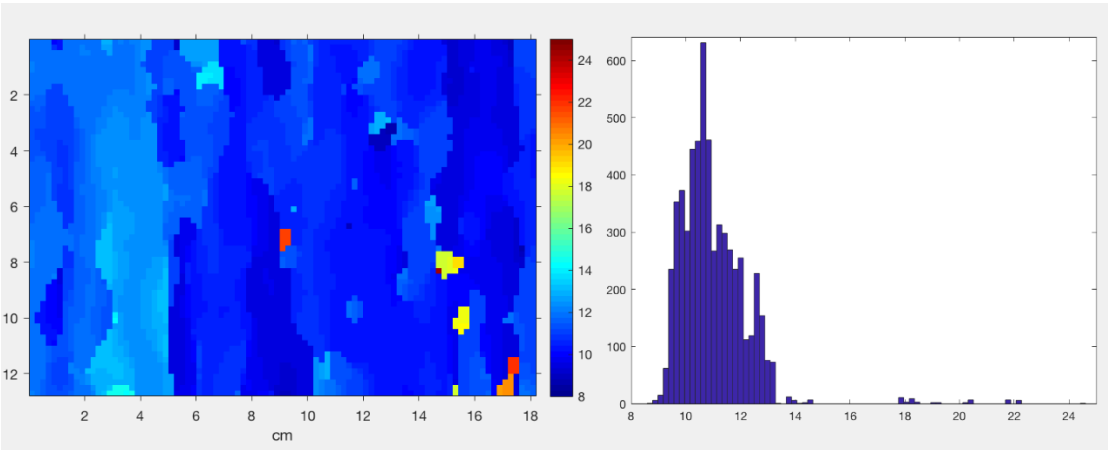

Figura 7-24. Mapa de colores e histograma de hilos verticales sobre salida con PCA de *p01175p02xf2017*

- Conteo de hilos horizontales:

En el caso de los hilos horizontales sí que se observan más discontinuidades en el mapa de colores, que ambos algoritmos intentan eliminarlas de alguna forma, siendo de nuevo el auto-codificar el que más éxito tiene como se puede ver en las gráficas.

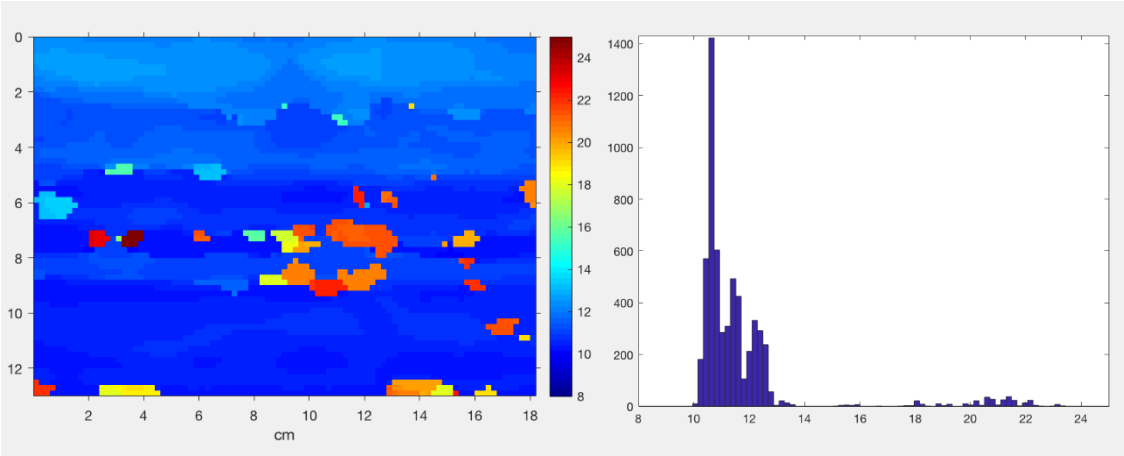

Figura 7-25. Mapa de colores e histograma de hilos horizontales sobre imagen original de *p01175p02xf2017*

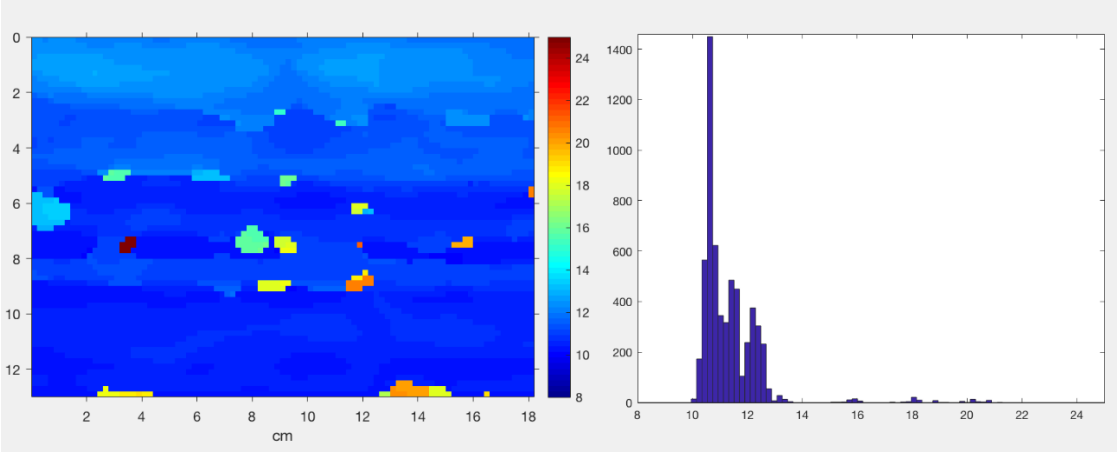

Figura 7-26. Mapa de colores e histograma de hilos horizontales sobre salida con AE de *p01175p02xf2017*

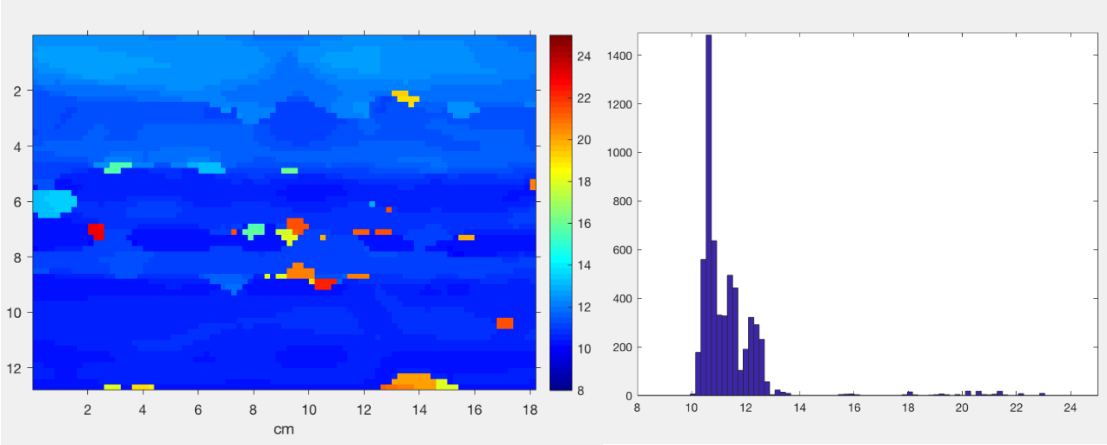

Figura 7-27. Mapa de colores e histograma de hilos horizontales sobre salida con PCA de *p01175p02xf2017*

- Comparación media y desviación estándar:

Centrando de nuevo la atención sobre la desviación estándar, es el auto-codificador quien mejores resultados obtiene tanto en el conteo de hilos verticales como en los horizontales. No obstante, en esta ocasión PCA no hace mal trabajo y también obtiene resultados que se acercan al del auto-codificador.

Tabla 7–4. Comparativa media y desviación estándar para imagen p01175p02xf2017

Conteo de hilos	Media			Desviación estándar		
	Img. Original	Salida AE	Salida PCA	Img. Original	Salida AE	Salida PCA
Verticales	11.07	10.91	10.92	1.81	1.16	1.30
Horizontales	11.63	11.28	11.31	2.23	1.25	1.50

8 CONCLUSIONES

Sólo podemos ver un poco del futuro, pero lo suficiente
para saber que hay mucho por hacer.

- Alan Turing -

El estudio previo realizado sobre el fundamento teórico de las redes neuronales y en concreto de los auto-codificadores ha sido el punto de partida para el diseño de un auto-codificador que sea capaz de aprender las condiciones que se requerían a la salida de éste.

Se han analizado los parámetros de diseño posibles y se han realizado diversas pruebas previas sobre una base de datos de dígitos manuscritos, que han dado una idea general de posibles resultados para la aplicación final, en la que se pudieran extraer características importantes de la imagen a partir de una compresión previa.

Los modelos se han trasladado para aplicarlo sobre imágenes de Rayos X de cuadros, con la búsqueda de una salida en la que se haya reducido el ruido que contienen las imágenes y sea posible la visualización para futuros análisis del tejido de la obra de arte. También se ha realizado una comparativa de los resultados obtenidos con PCA, otro algoritmo de *Machine Learning* empleado para la reducción de la dimensionalidad.

Sobre los resultados obtenidos tras aplicar todo esto a las imágenes de Rayos X, se puede concluir que para el caso del auto-codificador, se alcanza un buen resultado con respecto a robustez frente al ruido (siendo capaz de eliminar ciertas grietas y otras imperfecciones), pero no se llega a alcanzar el objetivo de poder comprimir la imagen, ya que si esto se realizaba no era capaz de obtener todas las características que se requerían, dejando una salida borrosa y sin contenido.

En contraposición, los resultados con PCA han sido buenos en compresión, llegando a una imagen codificada con información detallada de un tamaño que para el caso del autoencoder no llegaba a mostrar nada de contenido. Pero se ha comprobado que no es nada robusto ante el ruido, no llegando a eliminarlo, y limitándose a mantenerlo como en la imagen original.

Tabla 8–1. Comparativa de resultados sobre aplicación de Rayos X de cuadros

Característica	Auto-codificador	PCA
Robustez frente a ruido	Capaz de quitar imperfecciones básicas	Mal resultado, ruido similar a imagen original
Compresión	A mayor compresión, salida más difusa	Buena compresión, resultado similar a entrada con tamaños pequeños
Resultado final requerido	OK, buena visualización	No OK, mucho ruido

Con el foco puesto sobre el objetivo de la aplicación que es el de poder realizar futuros conteos de los hilos de la imagen o análisis de patrones de entrelazado de los mismos, la mejor solución es la del auto-codificador, que ha sido capaz de eliminar parte de las irregularidades que presentaba la imagen original.

Respecto a una visión general del trabajo, se ha podido comprender que cada aplicación en la que se quiera aplicar técnicas de aprendizaje profundo requiere un análisis de los algoritmos a utilizar, diseño del modelo correcto y la implementación concreta para dicha aplicación. Como se ha podido ver, para una misma arquitectura, los resultados son diferentes en función de la aplicación (y en su defecto, de las entradas).

Como líneas futuras para la aplicación del autoencoder sobre las imágenes de Rayos X de cuadros, destacar que el siguiente paso sería profundizar en el análisis del conteo de hilos sobre las imágenes reconstruidas (ya se ha realizado un primer análisis de esto con el programa Aracne), o realizar otro tipo de estudios sobre el tejido de la obra a partir de estos resultados.

En este proyecto se han tomado unos valores de diseño para el auto-codificador que parecían razonables y con ellos se ha realizado el estudio presentado. No obstante, como mejora podría buscarse qué valores óptimos de diseño en el AE darían mejores prestaciones de conteo que las presentadas.

Además, sería objeto de estudio el poder eliminar los bordes que se aprecian tras reconstruir la imagen a la salida de los algoritmos con los recortes obtenidos. No se ha profundizado en este tema en el presente proyecto pero se pueden aplicar técnicas de procesado de imagen para reducir dichos bordes.

Otra mejora a contemplar es la integración de los entornos software empleados. Se ha utilizado tanto Matlab como Python para el tratamiento de la imagen y aplicación de las técnicas correspondientes. Como se comentó, se ha creado una función en Python que a partir del modelo entrenado se pueden obtener las imágenes reconstruidas. La idea sería integrar dicha función con Matlab, para que desde éste pueda ser llamada directamente. Existen librerías en las últimas versiones de Matlab que permiten dicha integración con funciones de Python.

Se ha podido ver a través de los resultados que no se ha llegado a una solución perfecta en materia de compresión y robustez frente al ruido. Otra línea futura sería continuar con el estudio de otras técnicas de *Deep Learning* o mejoras sobre el auto-codificador, como probar nuevas arquitecturas de éste que se apliquen sobre las imágenes y se obtengan mejores resultados que los actuales, siendo capaces de reducir valores como la desviación estándar en el conteo de los hilos del lienzo.

ANEXO: GUÍA DE EJECUCIÓN

Inteligencia es la habilidad para adaptarse a los cambios.

- Stephen Hawking -

En este anexo se van a exponer los pasos a seguir para la ejecución del código de la aplicación del autoencoder sobre las imágenes de Rayos X de cuadros. El código completo no se ha incluido en la memoria debido a su extensión. Tampoco se detallan las ejecuciones de las pruebas intermedias que se han realizado a lo largo del proyecto.

Los requisitos y configuración inicial para poder ejecutar la aplicación son los que se exponían en el apartado 4.1, siendo principal requisito tener en el sistema instalado Matlab y un entorno de desarrollo para Python: en este caso se ha emplado Anaconda y las ejecuciones se han realizado desde su módulo Spyder, así como las librerías necesarias de Keras y Tensorflow.

En la siguiente tabla se resume el conjunto de ficheros que componen la aplicación:

Tabla A–1. Ficheros de código que componen la aplicación

Nombre	Lenguaje	Funcionalidad
preprocesado_imagenRayosX.m	Matlab	Preprocesado y recorte de imágenes de Rayos X en bloques de 200x200
entrenamientoAE_rayosx.py	Python	Entrenamiento del autoencoder a partir de imágenes varias de entrada
ejecucionAE_rayosx.py	Python	Predicción autoencoder y reconstrucción de imagen completa
entrenamientoPCA_rayosx.py	Python	Entrenamiento del algoritmo PCA a partir de imágenes varias de entrada
ejecucionPCA_rayosx.py	Python	Predicción con PCA y reconstrucción de imagen completa
main.py	Python	Programa principal que llama a las funciones anteriores y contiene los parámetros necesarios

A continuación se muestran las diversas fases asociadas a cada uno de estos ficheros de código. La forma recomendada de ejecución es siguiendo este orden que se presenta.

Preprocesado de imágenes en Matlab

Esta fase se ejecuta completamente en Matlab con el fichero *preprocesado_imagenRayosX.m*. Es importante que este código sea ejecutado previo a los siguientes en Python para poder preprocesar las imágenes de Rayos X originales y obtener los recortes que servirán de entrada para los algoritmos.

Cada ejecución de este programa permite obtener los recortes para una única imagen, por lo que habrá que ejecutarlo tantas veces como imágenes a procesar (esto será necesario hacerlo para poder realizar el entrenamiento posterior de los algoritmos con recortes de diferentes imágenes).

Antes de ejecutarlo hay que tener en cuenta varios aspectos o parámetros a modificar directamente sobre el código en el comienzo del mismo:

- Parámetros generales:
    - Directorio imagen: las variables '*carpeta*' e '*imagen*' deben ser modificadas con el lugar del sistema donde se encuentre la imagen almacenada.
    - Tamaño de recortes: la varíable '*width*' indicará el tamaño en píxeles que se utilizará para recortar la imagen original (en bloques de tamaño '*width*'x'*width*'). Por defecto, el valor utilizado será de 200 píxeles.
    - Directorio destino: la variable '*carpeta_destino*' contiene la ruta del directorio donde se guardarán las imágenes de salida (recortes) generadas. Es importante que al finalizar la ejecución en dicho directorio solo se encuentren los recortes pertenecientes a una misma imagen y un fichero de texto originado con información sobre la imagen original, que se denomina '*info_img_orig.txt*'.
- Variables binarias: Si están a 1 (activadas), se ejecuta el código que contienen. Por defecto, todas activadas. Son las siguientes:
    - imageCut: Principal para la ejecución. Ejecuta el recortado y el guardado de todos los recortes realizados.
    - verbose: Para mostrar por pantalla información adicional.
    - Preprocesado: Si no se activa no se realiza la normalización de media y varianza sobre la imagen completa.
    - guardar_recortes: Esto guarda los diferentes recortes generados.
    - guardar_original: Esto guarda la imagen original sin los bordes.

Función principal

Este es el programa principal (*main.py*) a ejecutar en Python y que recoge los parámetros que serán necesarios configurar antes de ejecutarlo y la llamada a las funciones de las diferentes fases, tanto para el auto-codificador como para PCA.

A partir de aquí todos los códigos son en Python por lo que se empleará Spyder (módulo de Anaconda) para las ejecuciones.

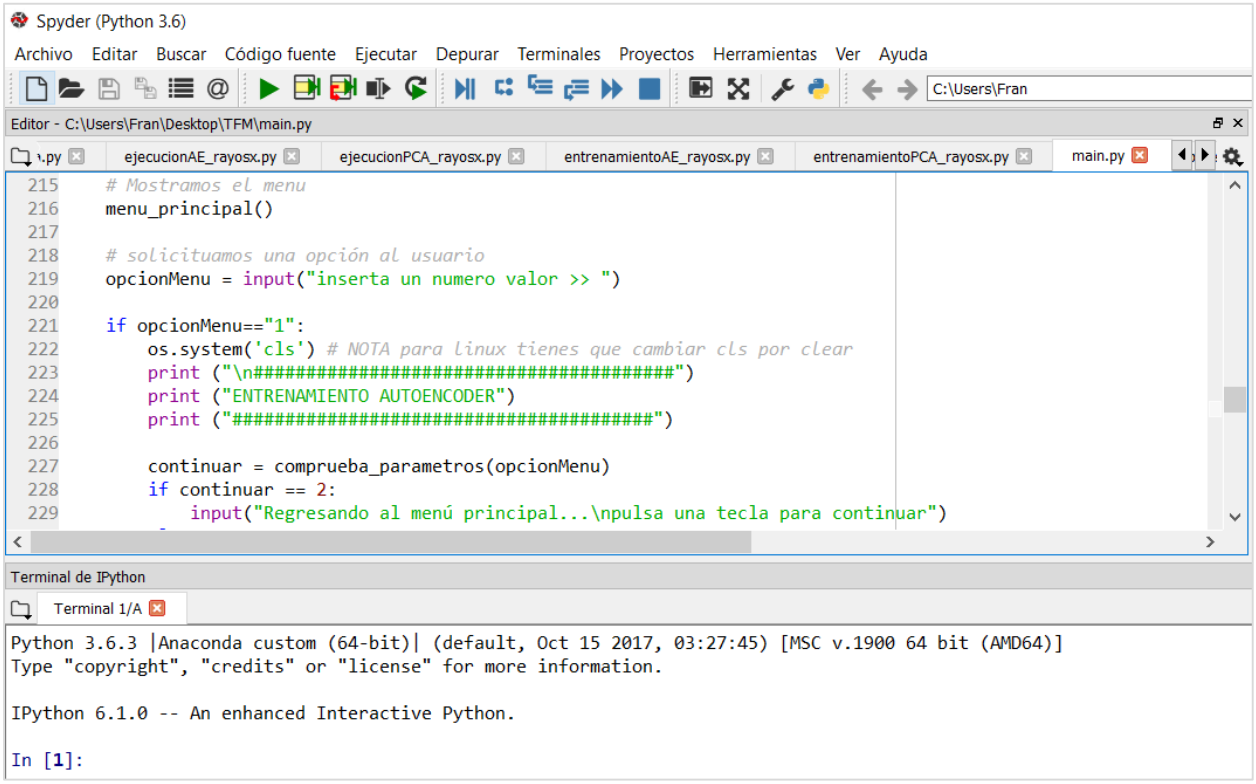

Figura A-0-1. Captura de Spyder para ejecuciones en Python

Un primer bloque de dicho programa recoge el conjunto de parámetros necesarios para ejecutar la aplicación, y están agrupados por fases, que serán detallados en cada una de ellas que se describen en los siguientes sub-apartados.

La ejecución de este main.py genera un menú desde el cuál se puede acceder a las diferentes funciones de la aplicación. Desde cada una de las opciones se puede acceder a las fases o funciones que se detallarán a continuación.

```
In [1]: runfile('C:/Users/Fran/Desktop/TFM/main.py', wdir='C:/Users/Fran/Desktop/TFM')
Selecciona una opción
        1 - ENTRENAMIENTO AUTOENCODER
                Función para entrenar el AE con recortes de diversos recortes
                de imágenes de Rayos X de cuadros diferentes.

        2 - EJECUCIÓN APLICACIÓN AUTOENCODER
                Función para ejecutar aplicación con AE de una imagen de Rayos X
                de cuadros que devolverá como salida dicha imagen reconstruida
                tras pasar por el AE con el modelo previamente entrenado.

        3 - ENTRENAMIENTO PCA
                Función para entrenar PCA con recortes de diversos recortes
                de imágenes de Rayos X de cuadros diferentes.Se realiza
                previamente una visualización con diversos numeros de componentes.

        4 - EJECUCIÓN APLICACIÓN PCA
                Función para ejecutar aplicación con PCA de una imagen de Rayos X
                de cuadros que devolverá como salida dicha imagen reconstruida
                tras pasar por el algoritmo PCA.

        5 - salir

inserta un numero valor >>
```

Figura A-0-2. Menú de opciones de función principal *main.py*

Entrenamiento autoencoder

En esta fase se define la arquitectura del autoencoder y lo entrena guardando el modelo y obteniendo las gráficas de "accuracy" y "loss".

Algunas consideraciones a tener en cuenta previa a la ejecución y los parámetros necesarios son:

- La variable 'carpeta_entrenamiento' es el directorio donde se encuentran las imágenes (recortes de 200x200) que van a servir de entrada para el entrenamiento. Aquí se han incluido recortes de varias imágenes de Rayos X diferentes (como se mostró en el apartado 5.2.1). Estos recortes se deben guardar en dicho directorio de forma manual. Es importante que en el directorio indicado solo se encuentren las imágenes para evitar problemas en la ejecución.
- En este código se llama a la función '*entrenamientoAE_rayosx'* donde se define la arquitectura del auto-codificador y los parámetros a utilizar. Esta ha sido una de las partes más modificadas durante la realización del proyecto para probar las diferentes alternativas que ofrece el autoencoder. Se ha dejado la arquitectura final de la solución expuesta en el apartado 6.3.
- Desde la función principal se pueden definir parámetros importantes para el entrenamiento del autoencoder:
    - *epochs*: número de epochs empleado. Por defecto 50.
    - *batch_size*: valor del 'batch_size' a utilizar. Por defecto 50.
    - *optimizer*: función de optimización para el entrenamiento del modelo. Por defecto '*adadelta'*.
    - *loss*: función de pérdidas para el entrenamiento del modelo. Por defecto '*binary_crossentropy*'.
- En el código se realiza un guardado del modelo una vez entrenado. Este guardado se realiza sobre la variable 'modelo_AE_a_guardar', que deberá indicar la ruta del fichero en formato .h5.

```
########################################
ENTRENAMIENTO AUTOENCODER
########################################
--------------------------------------------------
Estos son los parámetros que se van a utilizar:

ENTRENAMIENTO AE:
        - carpeta_entrenamiento = C:/Users/Fran/Desktop/TFM/RayosX/muestras_varias/
        - modelo_AE_a_guardar = C:/Users/Fran/Desktop/TFM/RayosX/modelo_autoencoder.h5
        - parámetros entrenamiento:
                - epochs = 50
                - batch_size = 50
                - optimizer = adadelta
                - loss = binary_crossentropy

IMPORTANTE: Antes de ejecutar este código, los recortes deben de haberse generado desde el código en
Matlab: preprocesado_imagenRayosX.m
El directorio de entrenamiento debe contener imágenes de diversos cuadros, que deben guardarse de forma
manual en la carpeta indicada (por ejemplo con nombre "imágenes varias").
--------------------------------------------------

¿Está conforme con estos parámetros? [s/n]
```

Figura A-0-3. Captura de ejecución de opción "Entrenamiento Autoencoder"

Ejecución aplicación autoencoder

Esta fase llama a la función *ejecucionAE_rayosx* que será donde se aplicarán a las imágenes del directorio que se le pasa como parámetro el modelo del autoencoder ya entrenado. Además se realizará la reconstrucción de la imagen completa a partir de las salidas del autoencoder.

Algunos aspectos a tener en cuenta previo a la ejecución de dicha función son:

- El directorio donde se encuentren los recortes (imágenes a la entrada del autoencoder) que se indique con el parámetro 'carpeta_recortes' debe contener solo las imágenes a utilizar y el fichero de información '*info_img_orig.txt*'. De este fichero de información se obtendrán los parámetros necesarios de tamaños, resolución, etc. para la llamada de la función.
- Las imágenes de entrada deben estar ordenadas por nombre de forma que el orden sea el mismo que cuando se crearon los recortes con Matlab, ya que es de esta forma como se realiza la posterior reconstrucción. Si no han sido modificados, desde el preprocesado en Matlab ya se guardan con su nombre correcto para no tener que cambiar nada aquí.
- Aquí no se entrena el modelo del autoencoder, se carga el modelo previamente guardado durante el entrenamiento. Si esto se ha guardado en una ubicación diferente a donde se ejecuta este código, debe modificarse la ruta correspondiente cuando se llama a la función 'load_model'.
- Los parámetros a definir desde la función principal son:
    - *carpeta_recortes*: directorio donde se han generado los recortes de un cuadro determinado (para poder reconstruirlo a la salida). Importante: este parámetro es compartido para PCA.
    - *modelo_AE_guardado*: nombre fichero .h5 donde se ha guardado el modelo de AE entrenado previamente.
    - *img_reconstruida_a_guardar_AE*: donde se guardará la salida del AE una vez reconstruida.

```
########################################
EJECUCIÓN APLICACIÓN AUTOENCODER
########################################
------------------------------------------------------
Estos son los parámetros que se van a utilizar:

EJECUCIÓN APLICACIÓN AE:
         - carpeta_recortes = /Users/Fran/Desktop/TFM/RayosX/p01175p02xf2017_EnSim_mitad/
         - modelo_AE_guardado = C:/Users/Fran/Desktop/TFM/RayosX/modelo_autoencoder.h5
         - img_reconstruida_a_guardar = C:/Users/Fran/Desktop/TFM/RayosX/recon/recon_AE_F00205p01.tif

Imagen a la que se aplicará el AE: p01175p02xf2017

IMPORTANTE: Antes de ejecutar este código, los recortes deben de haberse generado desde el código en
Matlab: preprocesado_imagenRayosX.m
IMPORTANTE: Debe de haberse entrenado el modelo previamente a esta ejecución.
------------------------------------------------------

¿Está conforme con estos parámetros? [s/n]
```

Figura A-0-4. Captura de ejecución de opción "Ejecución aplicación Autoencoder"

Entrenamiento PCA

Se realiza la llamada a la función *entrenamientoPCA_rayosx*, que siguiendo la misma estructura que para el caso del autoencoder, toma un conjunto de recortes de diferentes imágenes y les aplica el algoritmo de PCA y genera un modelo que será utilizado posteriormente.

Algunos aspectos a tener en cuenta son:

- La ruta del directorio donde se encuentran las imágenes de entrada será compartida con el entrenamiento del auto-codificador, por lo que se reutiliza la variable con nombre de 'carpeta_entrenamiento', ya empleada para el auto-codificador en la función principal.
- En la variable 'modelo_PCA_a_guardar' se indica la ruta del fichero donde se guardará el modelo PCA entrenado.
- El número de componentes utilizado para aplicar el algoritmo de PCA puede incluirse desde el "main" con la variable 'num_componentes'. Por defecto se utiliza el valor de 500 componentes.
- Se tiene además una variable binaria que se pasa como parámetro a la función que se llama 'pruebas_visualizacion'. Se marcará a '1' si se quieren ver las pruebas de visualización o a '0' si se desea pasar directamente al entrenamiento.

```
#######################################
ENTRENAMIENTO PCA
#######################################
-------------------------------------------------
Estos son los parámetros que se van a utilizar:

ENTRENAMIENTO PCA:
         - carpeta_entrenamiento = C:/Users/Fran/Desktop/TFM/RayosX/muestras_varias/
         - modelo_PCA_a_guardar = C:/Users/Fran/Desktop/TFM/pca_model
         - num_componentes = 500
         - pruebas_visualizacion = 1

IMPORTANTE: Antes de ejecutar este código, los recortes deben de haberse generado desde el código en
Matlab: preprocesado_imagenRayosX.m
El directorio de entrenamiento debe contener imágenes de diversos cuadros, que deben guardarse de forma
manual en la carpeta indicada (por ejemplo con nombre "imágenes varias".
-------------------------------------------------

¿Está conforme con estos parámetros? [s/n]
```

Figura A-0-5. Captura de ejecución de opción "Entrenamiento PCA"

Ejecución aplicación PCA

Al igual que para el caso del autoencoder, en esta opción se llama a la función *ejecucionPCA_rayosx* para que utilice el modelo de PCA entrenado previamente y lo aplique a los recortes de una imagen completa, siendo la salida reconstruida.

Las consideraciones previas a tener en cuenta son las siguientes:

- El directorio donde se encuentren los recortes (imágenes a la entrada del autoencoder) que se indique con el parámetro 'carpeta_recortes' debe contener solo las imágenes a utilizar y el fichero de información '*info_img_orig.txt*'. Dicha variable será compartida con el autoencoder.
- Las imágenes de entrada deben estar ordenadas por nombre de forma que el orden sea el mismo que cuando se crearon los recortes con Matlab, ya que es de esta forma como se realiza la posterior reconstrucción. Si no han sido modificados, desde el preprocesado en Matlab ya se guardan con su nombre correcto para no tener que cambiar nada aquí.

- Además, se tienen dos parámetros adicionales que son necesarios indicar desde la función principal:
    - *modelo_PCA_guardado*: nombre fichero donde se ha guardado el modelo de PCA entrenado previamente.
    - *img_reconstruida_a_guardar_PCA*: ruta donde se guardará la salida de PCA con la imagen reconstruida.
- El resto de parámetros necesarios para pasarle a la función *ejecucionPCA_rayosx* son tomados directamente desde el fichero de información de la imagen original ('*info_img_orig.txt*'), con datos como el tamaño, la resolución, etc.

```
########################################
EJECUCIÓN APLICACIÓN PCA
########################################
--------------------------------------------------
Estos son los parámetros que se van a utilizar:

EJECUCIÓN APLICACIÓN PCA:
         - carpeta_recortes = /Users/Fran/Desktop/TFM/RayosX/p01175p02xf2017_EnSim_mitad/
         - modelo_PCA_guardado = C:/Users/Fran/Desktop/TFM/pca_model
         - img_reconstruida_a_guardar = C:/Users/Fran/Desktop/TFM/RayosX/recon/
recon_PCA_p01175p02xf2017.tif

Imagen a la que se aplicará el AE: p01175p02xf2017

IMPORTANTE: Antes de ejecutar este código, los recortes deben de haberse generado desde el código en
Matlab: preprocesado_imagenRayosX.m
IMPORTANTE: Debe de haberse entrenado el modelo previamente a esta ejecución.
--------------------------------------------------

¿Está conforme con estos parámetros? [s/n]
```

Figura A-0-6. Captura de ejecución de opción "Ejecución aplicación PCA"

REFERENCIAS

[1] J. Patterson y A. Gibson, Deep learning : a practitioner's approach, O'Reilly, 2017.

[2] J. Johnson y A. Karpathy, «CS231n: Convolutional Neural Networks for Visual Recognition.,» [En línea]. Available: http://cs231n.github.io/convolutional-networks/.

[3] I. Shafkat, «Intuitively Understanding Variational Autoencoders,» 2018. [En línea]. Available: https://towardsdatascience.com/intuitively-understanding-variational-autoencoders-1bfe67eb5daf.

[4] R. López, «¿Qué es y cómo funciona "Deep Learning"?,» 2014. [En línea]. Available: https://rubenlopezg.wordpress.com/2014/05/07/que-es-y-como-funciona-deep-learning/.

[5] «Convolutional Autoencoder: Clustering Images with Neural Networks,» 2018. [En línea]. Available: https://sefiks.com/2018/03/23/convolutional-autoencoder-clustering-images-with-neural-networks/.

[6] J. J. Murillo Fuentes, I. Fondón García, M. Ternero Gutiérrez, L. Córdoba Saborido y P. Aguilera Bonet, «Manual Aracne - DTSC. Universidad de Sevilla,» [En línea]. Available: http://grupo.us.es/gapsc/aracne/manual/.

[7] I. Goodfellow, Y. Bengio y A. Courville, «Deep Learning Book - Chapter 14: Autoencoders,» 2016. [En línea]. Available: https://www.deeplearningbook.org/contents/autoencoders.html.

[8] «UFLDL Stanford Tutorial: Autoencoders,» [En línea]. Available: http://ufldl.stanford.edu/tutorial/unsupervised/Autoencoders/.

[9] A. Dertat, «Applied Deep Learning - Part 3: Autoencoders,» 2017. [En línea]. Available: https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798.

[10] M. Alberti, «Introducción al autoencoder,» 2018. [En línea]. Available: https://www.deeplearningitalia.com/introduzione-agli-autoencoder-2/.

[11] K. Srinivasan, «Yale Data Science: Guide to Autoencoders,» 2016. [En línea]. Available: https://yaledatascience.github.io/2016/10/29/autoencoders.html.

[12] A. Kristiadi, «Deriving Contractive Autoencoder and Implementing it in Keras,» 2018. [En línea]. Available: https://wiseodd.github.io/techblog/2016/12/05/contractive-autoencoder/.

[13] R. R. Kabra, «Contractive autoencoder and SVM for recognition of handwritten Devanagari numerals,» *IEEE Xplore,* 2017.

[14] «What is the the difference between performing upsampling together with strided transpose convolution and transpose convolution with stride 1 only?,» [En línea]. Available: https://stackoverflow.com/questions/48226783/what-is-the-the-difference-between-performing-

upsampling-together-with-strided-t.

[15] N. Shibuya, «Up-sampling with Transposed Convolution,» 2017. [En línea]. Available: https://towardsdatascience.com/up-sampling-with-transposed-convolution-9ae4f2df52d0.

[16] O. Cohen, «PCA vs Autoencoders,» 2018. [En línea]. Available: https://towardsdatascience.com/pca-vs-autoencoders-1ba08362f450.

[17] «Apuntes de la asignatura Sistemas Inteligentes,» de *2º curso en Máster en Ingeniería de Telecomunicación*, 2018.

[18] K. Siwek y S. Osowski, «Autoencoder versus PCA in face recognition,» *IEEE Xplore,* 2017.

[19] S. Ioffe y C. Szegedy, «Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift,» 2015. [En línea]. Available: https://arxiv.org/pdf/1502.03167.pdf.

[20] «Tensorflow,» [En línea]. Available: https://www.tensorflow.org/?hl=es.

[21] «Keras: The Python Deep Learning library,» [En línea]. Available: https://keras.io/.

[22] «Scikit-image,» [En línea]. Available: http://scikit-image.org/docs/stable/.

[23] N. S. Mutha, «Install TensorFlow with GPU for Windows 10,» 2017. [En línea]. Available: http://blog.nitishmutha.com/tensorflow/2017/01/22/TensorFlow-with-gpu-for-windows.html.

[24] A. Ng, «Sparse autoencoder - Stanford CS294A Lecture notes,» [En línea]. Available: https://web.stanford.edu/class/cs294a/sparseAutoencoder.pdf.

[25] F. Chollet, «Building Autoencoders in Keras,» 2016. [En línea]. Available: https://blog.keras.io/building-autoencoders-in-keras.html.

[26] «Principal Component Analysis in an image with scikit-learn and scikit-image,» [En línea]. Available: https://pakallis.wordpress.com/2013/06/20/principal-component-analysis-in-an-image-with-scikit-learn-and-scikit-image/.

[27] A. Baaj, «Keras Tutorial: Content Based Image Retrieval Using a Convolutional Denoising Autoencoder,» 2017. [En línea]. Available: https://blog.sicara.com/keras-tutorial-content-based-image-retrieval-convolutional-denoising-autoencoder-dc91450cc511.

[28] A. Sharma, «Implementing Autoencoders in Keras: Tutorial,» 2018. [En línea]. Available: https://www.datacamp.com/community/tutorials/autoencoder-keras-tutorial.

[29] A. Rosebrock, «Understanding regularization for image classification and machine learning,» Septiembre 2016. [En línea]. Available: https://www.pyimagesearch.com/2016/09/19/understanding-regularization-for-image-classification-and-machine-learning/.

[30] J. Brownlee, «How to Reduce Generalization Error With Activity Regularization in Keras,» Noviembre 2018. [En línea]. Available: https://machinelearningmastery.com/how-to-reduce-generalization-error-in-deep-neural-networks-with-activity-regularization-in-keras/.

GLOSARIO

IOT	Internet Of Things
ANN	Artificial Neural Networks
AI	Artificial Intelligence
CNN	Convolutional Neural Networks
MSE	Mean Squared Error
AE	Autoencoder
PCA	Principal Component Analysis
DAE	Denoising Autoencoder
CAE	Contractive Autoencoder
VAE	Variational Autoencoder
SVD	Singular Value Decomposition
CPU	Central Processing Unit
GPU	Graphics Processing Unit
TPU	Tensor Processing Unit