UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica

Grado en Ingeniería Informática en Ingeniería del Software

Trabajo Fin de Grado

Coloreado automático de fotografías usando técnicas de *Deep Learning*

UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica

Grado en Ingeniería Informática en Ingeniería del Software

Trabajo Fin de Grado

Coloreado automático de fotografías usando técnicas de *Deep Learning*

Autor: Sofía Isabel Mendoza Gutiérrez

Tutor: Alberto Gómez Mancha

ÍNDICE GENERAL DE CONTENIDOS

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

RESUMEN

La coloración de imágenes (*image colorization* en inglés) consiste en colorear fotografías que están en blanco y negro. Este proceso puede parecer tener poca relevancia ya que para los seres humanos es relativamente sencillo averiguar los colores reales que tienen algunos de los elementos, como el cielo o la hierba, que aparecen en una imagen que estamos viendo en blanco y negro. Sin embargo, la tarea de coloración no consiste en averiguar los colores sino en pintarlos sobre la imagen y, esta tarea, sí es compleja y requiere a una persona con habilidades para realizarla.

Por supuesto, todos estos motivos han dado lugar a que, en los últimos años, el problema del coloreado de fotografías haya cobrado gran importancia dentro del ámbito de la informática y se hayan desarrollado distintos algoritmos para solucionarlo.

Algunos de estos algoritmos son completamente automáticos y están basados, la mayor parte de ellos, en *Deep Learning* [1].

Este proyecto consiste en construir una herramienta que convierta imágenes en blanco y negro en imágenes a color utilizando técnicas de *Deep Learning*. Para ello, se construyen varias arquitecturas de redes neuronales y se entrenan con imágenes en blanco y negro y sus correspondientes imágenes coloreadas. Esto implica que estas redes aprenden y, posteriormente, son capaces de llevar a cabo el coloreado de fotografías de manera automática.

1. INTRODUCCIÓN

La coloración de imágenes (*image colorization* en inglés) consiste en colorear fotografías que están en blanco y negro. Este proceso puede parecer tener poca relevancia ya que para los seres humanos es relativamente sencillo averiguar los colores reales que tienen algunos de los elementos, como el cielo o la hierba, que aparecen en una imagen que estamos viendo en blanco y negro. Sin embargo, la tarea de coloración no consiste en averiguar los colores sino en pintarlos sobre la imagen y, esta tarea, sí es compleja y requiere a una persona con habilidades para realizarla.

Por supuesto, todos estos motivos han dado lugar a que, en los últimos años, el problema del coloreado de fotografías haya cobrado gran importancia dentro del ámbito de la informática y se hayan desarrollado distintos algoritmos para solucionarlo. Estos algoritmos pueden ser de dos tipos.

Por un lado, existen técnicas semiautomáticas a partir de imágenes de referencia en las que es necesaria la intervención del usuario.

Por otro lado, se desarrollan algoritmos completamente automáticos basados, la mayor parte de ellos, en *Deep Learning* [1].

En esta memoria se presenta el proyecto desarrollado en este trabajo fin de grado.

En primer lugar, se establecen los objetivos del proyecto y se muestra la planificación de horas que se ha seguido junto a un diagrama de *Gantt* en el que se puede observar una visión general del desarrollo.

En el tercer capítulo se explican las bases del proyecto en las cuales se erige el proyecto. Se dan distintas nociones fundamentales relacionadas con el color y la imagen digital que resultan necesarias para comprender el desarrollo de este proyecto.

En el cuarto apartado se aclaran los conceptos básicos relacionados con la Inteligencia Artificial, el *Deep Learning* y las redes neuronales. Asimismo, se analizan algunos estudios que resuelven el problema de coloreado de imágenes haciendo uso de redes neuronales profundas.

Dentro del quinto punto se enumeran las fases de creación de un modelo de *Deep Learning* y se comentan las herramientas que van a ser usadas para la llevar a cabo la implementación.

A continuación, se presenta de qué forma se ha desarrollado la implementación y se explican las distintas arquitecturas que se han construido.

El penúltimo capítulo trata de analizar los resultados obtenidos con las distintas arquitecturas y realizar comparaciones entre ellos.

Por último, se concluye el proyecto y se mencionan algunas mejoras o avances que se podrían llevar a la práctica en el futuro.

2. OBJETIVOS

El objetivo principal de este proyecto consiste en construir una herramienta que sea capaz de colorear imágenes en blanco y negro de forma automática. Para ello, se fijan otros objetivos complementarios.

Por un lado, se realiza una investigación exhaustiva de los distintos modelos de *Deep Learning* y de los artículos que existen en la actualidad y aportan soluciones al problema planteado.

Una vez se haya indagado lo suficiente en el estado de la cuestión, se estudian qué arquitecturas resultan interesantes y se construye un primer modelo. A continuación, se ajustan los parámetros de este llevando a cabo una optimización de hiperparámetros.

Asimismo, se desarrollan distintos modelos para realizar comparaciones entre los resultados obtenidos con cada uno de ellos.

Por último, cuando el proyecto estaba a punto de finalizar, se planteó desarrollar una interfaz de usuario que permitiese adjuntar una imagen en blanco y negro para luego colorearla.

2.1. Planificación

La planificación del proyecto se ha llevado a cabo usando la herramienta *Click up*. Esta plataforma permite definir distintas tareas, indicar el estado de estas y representarlas en forma de diagrama de *Gantt*.

El desarrollo del proyecto ha tenido una duración total de 340 horas aproximadamente, separadas en 4 tareas principales que se visualizan en la Tabla 1.

Análisis del estado de la cuestión	105 horas
Implementación y desarrollo	125 horas
Evaluación de resultados	10 horas (sin las horas dedicadas al entrenamiento)
Documentación	100 horas
Total	**340 horas**

Tabla 1: Planificación del desarrollo del proyecto.

Estas tareas, a su vez, se dividen en subtareas. Para plasmar en un cronograma tanto las tareas principales como las subtareas que emanan de ellas, estas se representan mediante una matriz de *Gantt* (Figura 1).

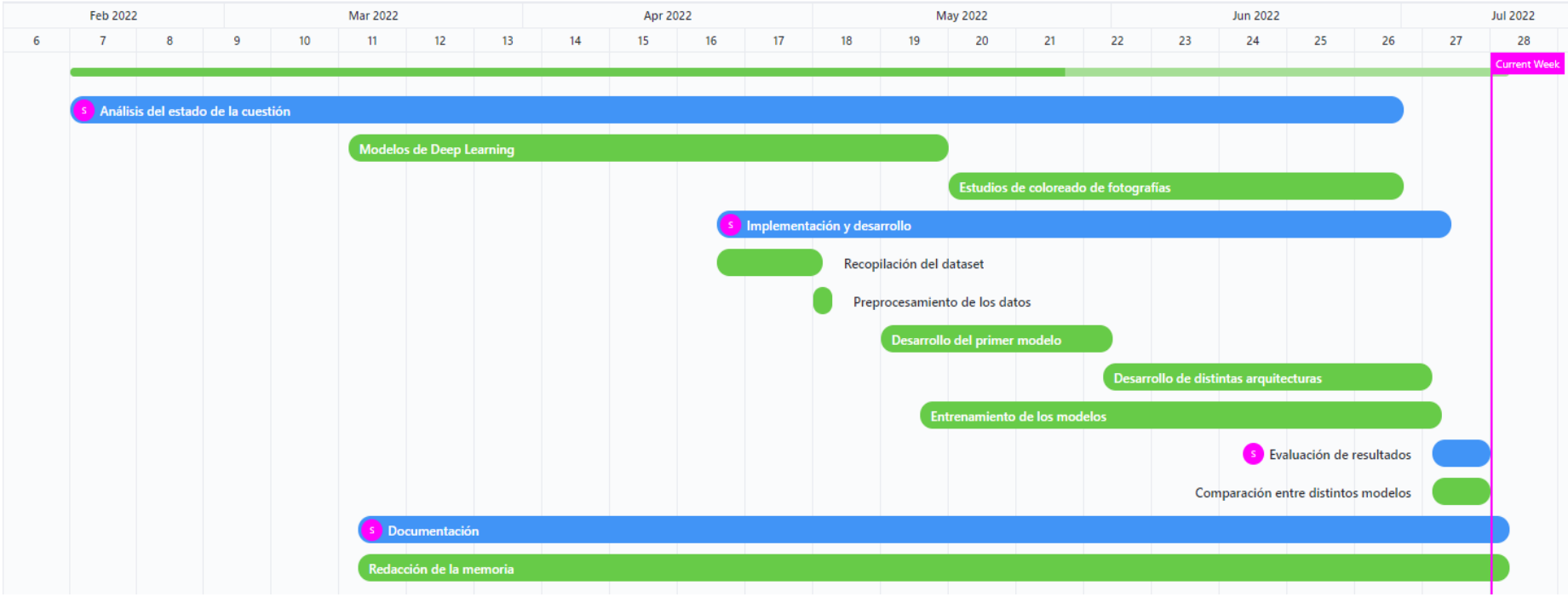

Figura 1: Matriz de Gantt.

3. BASES DEL PROYECTO

En este apartado se explican los cimientos sobre los que se construye el proyecto.

Por una parte, se describe qué es el color y se distinguen los dos tipos de modelos que existen para la obtención de colores.

A continuación, se definen los términos modelo de color y espacio de color, que, aunque en general se usan indistintamente, presentan diferencias.

Asimismo, se realiza una introducción sobre cómo se representa el color en el ámbito de la informática.

Por último, se detallan las características tanto del modelo *RGB* como del espacio de color *CIE L*a*b*.*

3.1. Coloración y teoría del color

El color, según la RAE, es una sensación producida por los rayos luminosos que impresionan los órganos visuales y que depende de la longitud de onda [2].

El ojo humano solo es capaz de captar las longitudes de onda cuando existe una gran cantidad de luminosidad.

A lo largo del tiempo, se han definido los colores primarios, a partir de los cuales pueden obtenerse toda la gama de colores que se conocen en la actualidad. Sin embargo, estos colores primarios no son siempre los mismos, ya que dependen de si provienen de una fuente de luz o de pigmento.

La síntesis aditiva es un modelo basado en la obtención de colores a partir de la superposición de luces de color rojo, verde y azul, ya que, en este modelo, estos son considerados los colores primarios. En este caso, los colores complementarios, generados a partir de la suma de dos primarios, son cian (producto de la suma del azul y el verde), magenta (producto de la suma del azul y el rojo) y amarillo (producto de la suma del rojo y el verde). Además, cabe destacar que la superposición de los tres colores primarios da lugar a la luz blanca [3].

La síntesis sustractiva, por el contrario, es un modelo basado en la creación de colores a partir de la mezcla de pigmentos o tintes. En un modelo sustractivo ideal, los colores primarios son cian, magenta y amarillo, los cuales dan lugar a los complementarios, que son rojo (obtenido a partir del magenta y el amarillo), azul (obtenido a partir del cian y el magenta) y verde (obtenido a partir del cian y el amarillo) [4]. Sin embargo, en la realidad, los colores primarios no suelen corresponderse con los que se consideran en la teoría debido a la multitud de matices que pueden tener las pinturas usadas [5].

3.2. Modelo de color y espacio de color

Un modelo de color es una manera de representar los colores de forma numérica, normalmente, a partir de tres o cuatro valores que se corresponden con componentes cromáticas. Es decir, un modelo de color establece una asociación entre un grupo de valores numéricos y un espacio de color. Sin embargo, si un modelo de color no tiene una función de asignación a un espacio de color absoluto, este modelo puede llegar a ser un sistema de color arbitrario que no está relacionado con ningún sistema de interpretación de color. Los modelos más usados son *RGB*, *CMYK*, *HSL*, *HSV* y el modelo tradicional de coloración [6].

Un espacio de color indica la manera en que un color está definido. Dicho de otra forma, es un modo de interpretación del color, es decir, una forma concreta de estructurar los colores en una imagen o video. Este viene dado tanto por el modelo de color que utiliza, como por los dispositivos físicos donde se representa el color.

El espacio de color *CIE 1931 XYZ* se creó en base a intentar reproducir de la mejor manera posible los colores que percibe el ser humano y constituye los cimientos sobre los que se desarrollan la mayor parte de los espacios de color. *CIELUV*, *CIEUVW* y *CIELAB* son algunos ejemplos que proceden de *CIE XYZ* [7].

Aunque se puede observar que existe diferencia entre los términos modelo de color y espacio de color, ambos se suelen utilizar indistintamente. Sin embargo, este pensamiento es totalmente incorrecto ya que, por ejemplo,

aunque existen distintos espacios de color basados en el modelo de *RGB*, este no se corresponde con el espacio de color *RGB*.

3.3. Imagen digital

Al analizar una imagen digital se pueden observar distintas propiedades (Figura 2) sobre ella como las dimensiones, la extensión, el espacio de color, etc. Los números que hacen referencia a las dimensiones informan sobre la cantidad de píxeles que una imagen tiene tanto de anchura como de altura. Pero ¿qué es un píxel?

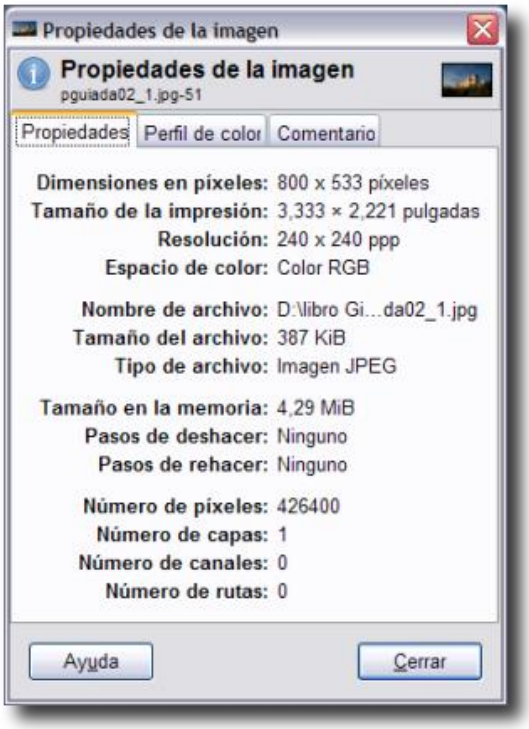

Figura 2: Propiedades de una imagen [8].

Un píxel es la menor unidad en color que forma parte de una imagen digital. Es decir, al ampliar lo suficiente una imagen en una pantalla de un dispositivo, pueden observarse los distintos píxeles de los que está compuesta.

Los ordenadores utilizan el modelo de color *RGB*, el cual será explicado en el siguiente apartado. Dicho de otro modo, cada píxel de la pantalla está formado, en realidad, por tres capas; una roja, una verde y una azul, cada una de las cuales tiene una intensidad concreta.

En la vida cotidiana, solemos hablar sobre colores haciendo referencia a su tono principal: verde, rojo, negro, etc. Los ordenadores, sin embargo,

procesan la información de forma numérica, y los colores no son una excepción. Por lo tanto, existen códigos diseñados para representar espacios de color [9].

3.4. Modelo de color *RGB*

RGB (siglas en inglés de *Red*, *Green*, *Blue*, en español «rojo, verde y azul») es un modelo de color basado en la síntesis aditiva de las tres capas que lo forman. En el modelo *RGB* (Figura 3), cada color está compuesto de tres canales (rojo, verde y azul) y se representa mediante la mezcla aditiva de estos [10].

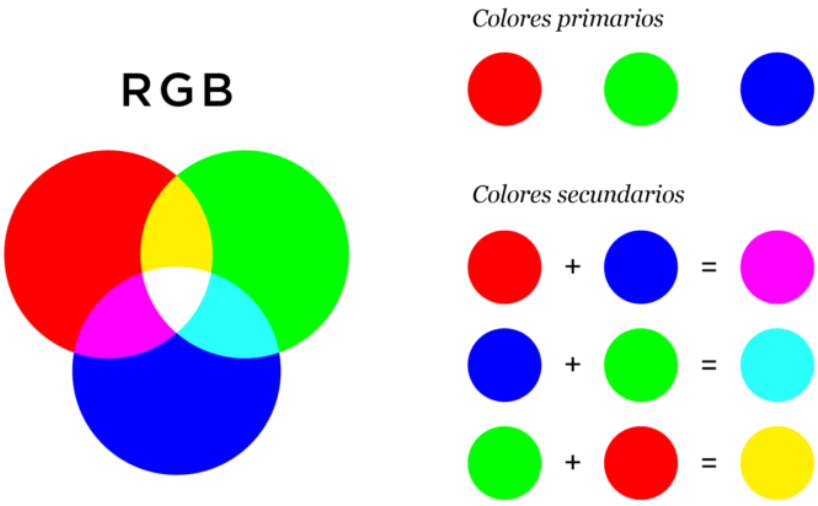

Figura 3: Modelo de color RGB [11].

Generalmente, este modelo de color se utiliza en el área del procesamiento digital de imágenes y elementos gráficos que necesitan reproducirse en canales digitales.

Las imágenes digitales *RGB* contienen un total de 3 bytes (24 bits) por píxel. Cada píxel usa 3 canales de información, *R*, *G* y *B*, uno por cada color primario (Figura 4). Por lo tanto, cada color está representado por un canal de un byte (8 bits), cuyo valor oscila entre 0 y 255 [11].

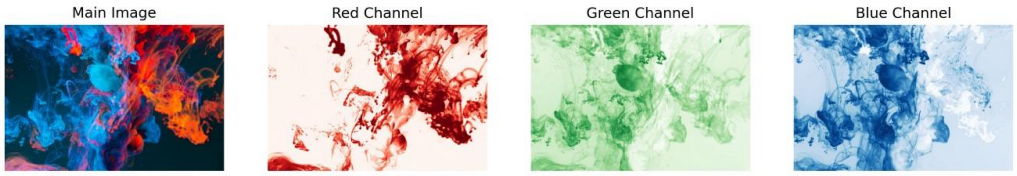

Figura 4: Canales R, G y B de una imagen [12].

Para indicar con qué proporción se mezcla cada color, se asigna un valor a cada uno de los colores primarios. Así, el valor 0 hace referencia a que el color no interviene en la mezcla y, según se incrementa esta cifra, más cantidad de este color se está aplicando a la mezcla. De este modo, cada píxel se representa mediante tres valores escritos en decimal, separados por comas y, todos ellos entre paréntesis.

Así pues, el rojo se obtiene con (255,0,0), el verde con (0,255,0) y el azul con (0,0,255), produciendo, en cada caso, un color resultante monocromático. La ausencia de color, es decir, el color negro (0,0,0), se obtiene cuando las tres componentes toman el valor 0. La combinación de dos colores a su máximo valor de 255 con un tercero con valor 0 da lugar a tres colores intermedios. De esta forma, aparecen el color amarillo (255,255,0), el cian (0,255,255) y el magenta (255,0,255).

Debido a que el modelo *RGB* es un modelo que se basa en la síntesis aditiva, la superposición de los tres colores primarios da lugar al color blanco. Por esta razón, el color blanco (255,255,255) se forma a partir de la combinación del valor máximo de los tres colores primarios [13].

3.5. Espacio de color *Lab*

Lab es el nombre resultante de la combinación de dos espacios de color. Uno de ellos, el más conocido, es *CIELAB* (*CIE 1976 L*a*b**) y el otro es *Hunter Lab* (*Hunter L, a, b*). Como se comentó anteriormente, estos dos espacios son derivados del espacio de color *CIE 1931 XYZ*. Sin embargo, la gran diferencia entre ambos es que *CIELAB* hace uso de raíces cúbicas para obtener resultados y, por el contrario, *Hunter Lab* utiliza raíces cuadradas en el cálculo [14].

Es importante resaltar la manera en la que el ojo humano puede percibir millones de colores. Cada individuo percibe e interpreta los colores de distinta forma y estas diferencias pueden derivar en algunos problemas de comprensión o acuerdo entre distintas partes.

Es por esta razón que la Comisión Internacional de la Iluminación (*CIE*), organización sin ánimo de lucro que es considerada como la máxima autoridad en la ciencia de la luz y el color, ha definido espacios de color, como *CIE XYZ* y *CIE L*a*b**, para expresar el color desde un punto de vista objetivo [15].

El espacio de color *CIE L*a*b** o, en definitiva, *Lab*, es ampliamente usado porque está inspirado en la forma en que los seres humanos percibimos los colores y es capaz de codificar más tonos que, por ejemplo, los espacios derivados del modelo de color *RGB*. Dicho de otro modo, correlaciona de forma consistente los valores numéricos que representan un color con la percepción visual humana. Sin embargo, es por esto por lo que se trata de un espacio de color teórico y, por tanto, no puede ser utilizado por ningún dispositivo [16].

Asimismo, existe una teoría, llamada teoría de los colores oponentes, la cual se basa en que dos colores no pueden ser rojo y verde al mismo tiempo, o amarillo y azul a la vez. Esta teoría sirvió como fundamento para la creación del espacio de color *Lab* (Figura 5).

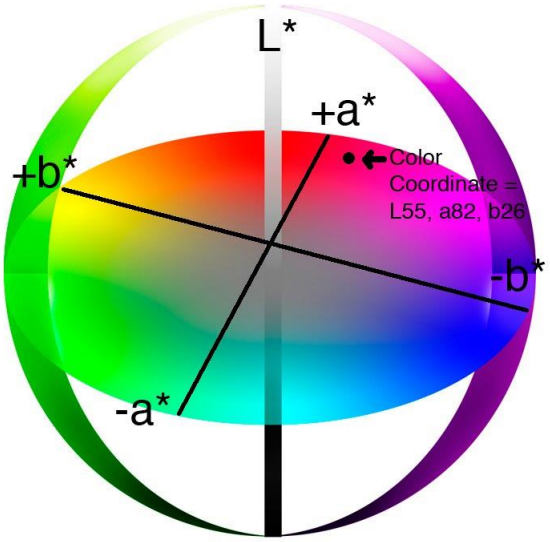

Figura 5: Espacio de color Lab [17].

El nombre de este espacio de color procede del esquema de representación de los colores que utiliza, que está dividido en tres ejes: el *L*, que permite medir la luminosidad e indica el brillo de los colores; el *a*, que es una de las coordenadas cromáticas y refleja la variación entre los colores verde y rojo; y el *b*, que es la otra coordenada cromática y representa la variación entre los colores azul y amarillo.

Asimismo, mientras que el valor de la componente *L* está dentro del rango 0-100, el valor de los ejes *a* y *b* oscila entre -127 y +127. En el caso de la componente *a*, los valores positivos hacen referencia al color rojo y los negativos al verde. Ahora bien, en el eje *b*, los valores positivos indican mayor cantidad de color amarillo y los negativos indican mayor cantidad de color azul [17].

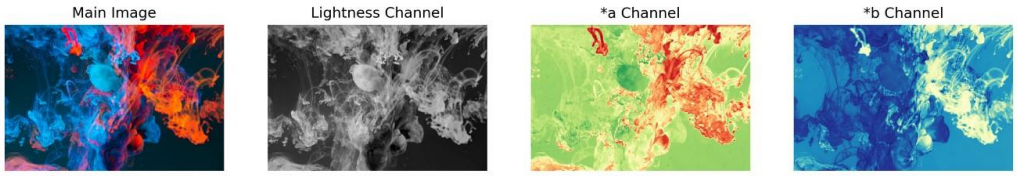

Figura 6: Canales L, a y b* de una imagen* [12].

En la mayor parte de los artículos que resuelven el problema de coloración de imágenes se recurre a utilizar el espacio de color *Lab* para entrenar el modelo. Esto se debe a que utilizando *Lab* es posible tomar el canal *L* (imagen en blanco y negro) como entrada para predecir los canales *a* y *b* (canales en color) y, finalmente, combinar los tres canales para formar la imagen coloreada (Figura 6). Sin embargo, al usar *RGB* sería necesario convertir la imagen de entrada a una imagen en blanco y negro, tomarla como entrada, y predecir tres canales en lugar de dos, lo cual hace que alcanzar un buen resultado sea aún más complicado [12].

4. ESTADO DE LA CUESTIÓN

4.1. Inteligencia Artificial o *Artificial Intelligence*

La Inteligencia Artificial o *Artificial Intelligence* (*AI*) es, en el ámbito de la informática, una rama que pretende construir máquinas y sistemas que actúen como un ser humano, es decir, que se comporten de forma inteligente [18]. La Inteligencia Artificial está compuesta de un conjunto de algoritmos lógicos que deben estar entrenados, los cuales permiten a los ordenadores poder tomar decisiones en casos concretos a partir de casos generales [19]. Andreas Kaplan y Michael Haenlein definen la Inteligencia Artificial como «la capacidad de un sistema para interpretar correctamente datos externos, para aprender de dichos datos y emplear esos conocimientos para lograr tareas y metas concretas a través de la adaptación flexible» [20].

4.2. Aprendizaje Automático o *Machine Learning*

El aprendizaje automático o *Machine Learning (ML)* es un subconjunto de la Inteligencia Artificial que permite que un ordenador aprenda de la experiencia [21]. Dicho de otra forma, es el conjunto de métodos y algoritmos que permiten a los ordenadores tener la capacidad de identificar patrones en datos masivos y predecir un resultado a partir de estos, es decir, aprender de forma automática [22].

Es importante saber qué diferencia a la programación explícita de los algoritmos basados en aprendizaje automático. En la programación explícita, como su propio nombre indica, los desarrolladores programan el código que se debe ejecutar explícitamente. Es el tipo de programación que todos conocemos y hemos usado, y es la base de casi todas las aplicaciones. Sin embargo, la utilización de este tipo de programación presentaría demasiadas dificultades a la hora de realizar tareas complejas como lo son, por ejemplo, el reconocimiento de facial y la coloración de imágenes.

Por otro lado, el ser humano es capaz de realizar este tipo de tareas de manera intuitiva casi sin darse cuenta. Es este el comportamiento que deseamos que adquieran los ordenadores mediante el aprendizaje

automático. Por lo tanto, para solucionar problemas como los mencionados anteriormente se recurre a algoritmos basados en *Machine Learning.*

En la siguiente sección, se define la tipología de métodos y la tipología de tareas que se pueden emplear dentro del *Machine Learning*.

4.2.1. Tipología de métodos

Los algoritmos de *Machine Learning* necesitan construir un modelo a partir del aprendizaje que van obteniendo de los datos, los cuales reciben el nombre de conjunto de entrenamiento. Durante la fase de entrenamiento, el algoritmo contrasta la salida ideal que deberían presentar los modelos con la salida real de los que se van construyendo. Es así como el modelo realiza el aprendizaje, el cual puede ser supervisado, no supervisado o por refuerzo [22].

- Aprendizaje supervisado: abarca algoritmos que hacen predicciones a partir de un aprendizaje previo basado en un conjunto de datos de entrada y los resultados que se esperan obtener asociados a estos datos. El problema de *image colorization* que se va a abordar se incluye en este tipo de aprendizaje.
- Aprendizaje no supervisado: abarca técnicas de *ML* que no emplean un conocimiento previo. Tratan de realizar observaciones con el objetivo de encontrar patrones, características y correlaciones que den la posibilidad de organizarlos de alguna forma.
- Aprendizaje por refuerzo: abarca algoritmos que intentan aprender de la propia experiencia. Esto es, mediante un proceso de prueba y error en el que las respuestas correctas se ven recompensadas, el algoritmo debe ser capaz de tomar la mejor decisión.

4.2.2. Tipología de tareas

Dentro del *Machine Learning*, existen distintos tipos de tareas dependiendo del tipo de problema que se desee resolver. Dos de los tipos de tareas más importantes son:

- Clasificación: intenta predecir a qué clase pertenece un dato introducido como entrada.

Coloreado automático de fotografías usando técnicas de *Deep Learning*

- Regresión: a partir de un dato de entrada, trata de predecir un número real como salida, basándose en los datos de entrenamiento.

4.3. Aprendizaje Profundo o *Deep Learning*

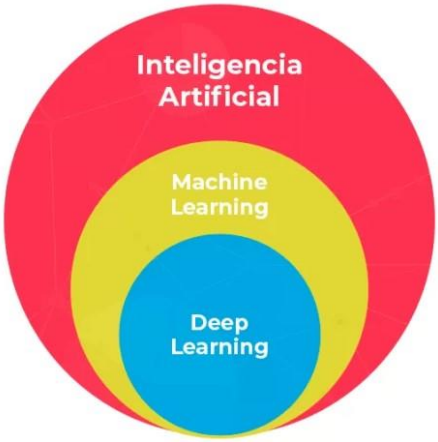

Figura 7: Relación entre IA, ML y DL [23].

El Aprendizaje Profundo o *Deep Learning (DL)* es un subconjunto del *Machine Learning* (Figura 7), cuyo objetivo es crear modelos que puedan representar ideas complejas a partir de ideas más simples. Es decir, el proceso que sigue el modelo consiste en crear automáticamente una jerarquía de conceptos en la que se parte de conceptos simples que se combinan para formar conceptos complejos.

En otras palabras, el *Deep Learning* se denomina así debido al funcionamiento en el que se basa: el aprendizaje por capas. Este tipo de aprendizaje consiste en que una primera capa reconoce distintas características del elemento de entrada que se le ha pasado al modelo. A continuación, estas características son enviadas a la siguiente capa, la cual se entrena para aprender características más abstractas, y así sucesivamente hasta conseguir que todas las capas del modelo hayan sido entrenadas. La profundidad del modelo viene dada por el número de capas que componen el modelo de datos [24].

Actualmente, por un lado, la cantidad datos que se almacena en la red y a la que es posible acceder fácilmente es inmensa y, por otro lado, las máquinas con las que se trabaja procesan la información cada vez más rápido. Esto hace que el *Deep Learning* adquiera cada vez relevancia ya que su dinámica

radica en desarrollar modelos entrenados muchas veces (se necesita alta capacidad de procesamiento de máquinas) a partir de muchos datos [25].

4.4. Redes Neuronales Artificiales

Las redes neuronales artificiales o *Artificial Neural Networks* (*ANN*) son un conjunto de algoritmos basados en la interconexión de las neuronas biológicas del cerebro.

4.4.1. Historia de las Redes Neuronales Artificiales

En los años 40, McCulloch y Pitts presentaron el primer modelo de red neuronal, cuyo objetivo es intentar explicar cómo funciona el cerebro.

Más tarde, entre los años 70 y 80, aun habiendo avanzado en la investigación gracias a ideas como la del perceptrón, estos estudios se vieron limitados debido a la falta de aplicabilidad de la computación neuronal.

En 1982, John Hopfield desarrolló el algoritmo de *backpropagation*, simplificando así el entrenamiento de las redes.

Es a principios del siglo XXI cuando las redes neuronales consiguen adquirir mayor importancia, acuñando el término de *Deep Learning*. Sin embargo, aunque algunos de los conceptos importantes del *Deep Learning* se inspiraron en el funcionamiento del cerebro, los modelos de aprendizaje profundo no son modelos del cerebro como tal, solo se inspiran en ellos [26].

4.4.2. Perceptrón

El perceptrón (Figura 8) es la unidad básica de procesamiento que compone una red neuronal, ya que actúa como una neurona artificial. Al igual que sucede con las neuronas biológicas, los perceptrones poseen conexiones de entrada a través de las que reciben estímulos externos, es decir, los valores de entrada. A partir de estos valores, la neurona se encarga de realizar distintos cálculos internos que dan lugar a un valor de salida.

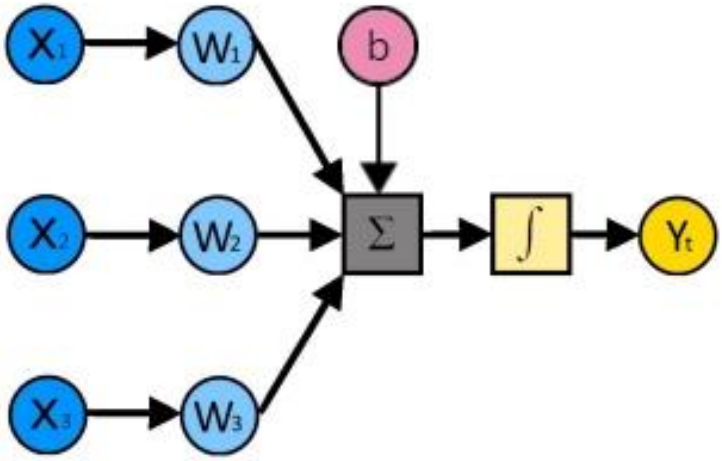

Figura 8: Perceptrón. Los datos de entrada (X_N) se multiplican por sus pesos correspondientes (W_N) y se suman entre sí y al sesgo (b). A este resultado se le aplica una función de activación y se obtiene una salida temporal (Y_N) [27].

El cálculo realizado en el interior de la neurona consiste en una suma ponderada de los valores de entrada junto a un parámetro fijo denominado sesgo o *bias*. Esta ponderación depende del peso o *weight* que se le asigna a cada una de las conexiones de entrada, es decir, cada una de las conexiones está asociada a un peso que define con qué intensidad cada variable afecta a la neurona [28]. De esta forma, este cálculo se podría representar con la siguiente fórmula:

$$y = \Sigma_i w_i x_i + b$$

Tanto los pesos como el sesgo son parámetros de la red neuronal y, como tal, deben ser ajustados durante el proceso de entrenamiento para que el modelo genere los mejores resultados posibles y el valor que tome el coste sea mínimo.

El coste o *loss* es un valor numérico que indica lo bueno o malo que es un modelo. Es decir, cuando el modelo realiza predicciones incorrectas, este valor aumenta. Por el contrario, cuanto más se ajusta la predicción generada con el resultado que se desea obtener menor es este valor [29].

Por otro lado, al resultado de la suma ponderada se le aplica una función de activación no lineal. La no linealidad de esta función provoca que el perceptrón sea capaz de identificar patrones y realizar predicciones a partir de ellos en un mayor número de situaciones [28].

4.4.3. Arquitectura de una Red Neuronal Artificial

El perceptrón, como red neuronal que es, también puede ser definido como la red neuronal con la arquitectura más sencilla, ya que, en este caso, esta estructura está compuesta de una única neurona.

Por otra parte, los perceptrones se pueden combinar para construir redes neuronales más complejas llamadas *Multi-Layer Perceptron* (*MLP*). Dentro de estas redes neuronales, las neuronas artificiales se distribuyen en forma de capas (Figura 9). Todas las neuronas que componen una capa comparten los mismos datos de entrada que utilizan para realizar los cálculos internos pertinentes.

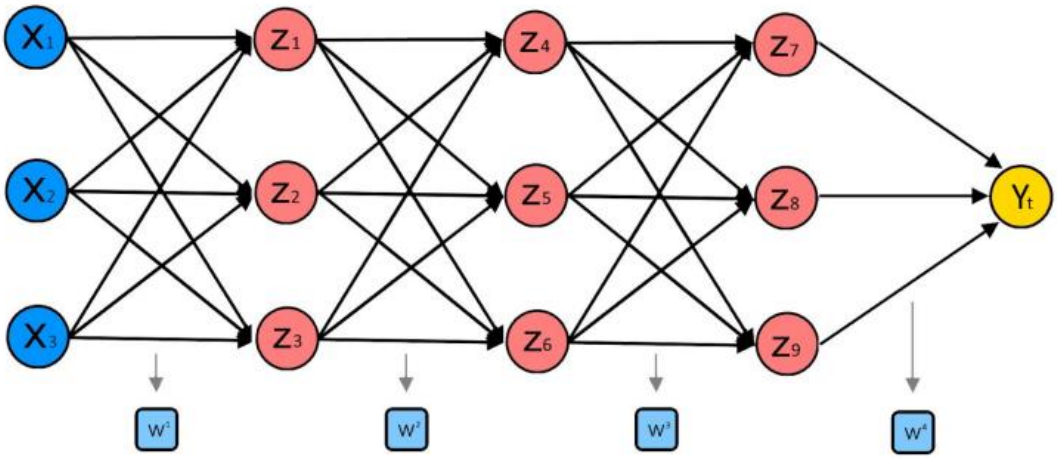

Figura 9: MLP con tres capas ocultas [27].

La primera capa de una red neuronal, la cual está formada por los datos iniciales, se denomina capa de entrada o *input layer*. Estos datos se envían a la siguiente capa, en la que las neuronas realizan cálculos y dan lugar a distintas salidas. Los datos de salida que han sido calculados por los perceptrones de esta capa son enviados a la siguiente capa de la red, actuando como datos de entrada de esta. De esta misma manera, los cálculos generados esta capa serán la entrada de la siguiente capa, y así sucesivamente. Por último, existe una última capa de neuronas, llamada capa de salida u *output layer*, que se corresponde con salida final de la red neuronal. Todas las capas que se encuentran entre la *input layer* y la *output layer* se denominan capas ocultas o *hidden layers*.

Tanto la capacidad y calidad de predicción como la capacidad de clasificación son características que dependen directamente del número de capas ocultas y de la cantidad de neuronas que haya en cada capa. Sin embargo, el aumento excesivo de una de estas dos propiedades puede desembocar en un problema de sobreentrenamiento, sobreajuste u *overfitting* [26]. El concepto de sobreentrenamiento se produce cuando una red aprende a realizar predicciones excesivamente buenas del conjunto de datos previo que se utiliza en el proceso de entrenamiento. Tanto es así, que no es capaz de producir resultados correctos a partir de datos nuevos con los que nunca ha sido entrenado.

La distribución de las neuronas en capas aporta una gran ventaja: la red adquiere la capacidad de aprender conocimiento jerarquizado. En una misma capa, cada neurona tiene una función distinta y se ocupa de predecir una característica concreta. Una única neurona se encarga de procesar los datos de entrada que le llegan obteniendo como resultado información más elaborada y compleja. Esta información forma parte de la entrada de las neuronas de la siguiente capa, y así sucesivamente. De esta manera, en cada una de las capas de la red se logran resultados cada vez más complejos.

Las redes neuronales artificiales o modelos que incluyen múltiples capas ocultas y, por lo tanto, elaboran un conocimiento jerarquizado en el que a partir de ideas simples se construyen conceptos complejos, son las que dan nombre al Aprendizaje Profundo.

La fase de aprendizaje de un modelo se realiza en dos pasos. El primero de ellos, *forwardpropagation*, consiste en que el *dataset* de entrenamiento atraviesa todas las capas de la red y realiza predicciones para obtener los resultados. Después, se utiliza una función de pérdida o *loss function* para comparar las predicciones generadas con los resultados deseados y evaluar cómo de correcto es el modelo. Tras esto, el siguiente paso es retropropagar la información con el fin de que se ajusten los pesos de las conexiones de cada una de las capas, empezando por la última. Dicho de otra manera, los valores que toman los pesos se ajustan desde la primera capa hasta la última teniendo en cuenta el coste de la salida de la red neuronal, para lo cual se utiliza el algoritmo denominado *backpropagation* [29].

Se denomina *epoch* o época a cada una de las veces que el conjunto de datos de entrenamiento completo atraviesa la red neuronal, tanto hacia adelante como hacia atrás. Si el número de *epochs* es demasiado bajo, la red se ve afectada por un problema de *underfitting*, es decir, el modelo no ha aprendido lo suficiente. Sin embargo, si el número de *epochs* aumenta excesivamente, es probable que se produzca el llamado *overfitting* [30].

El tamaño del *dataset* de entrenamiento está íntimamente relacionado con el tiempo que dura el proceso de entrenamiento. Existen distintas formas de agilizar esta fase y una de ellas consiste en separar los datos en lotes o *batches*. Es decir, a la hora de entrenar el modelo, se especifica un tamaño de lote o *batch size*. Esto provoca que en cada iteración dentro de una *epoch* se utilice únicamente una parte de los datos de entrada del tamaño determinado por el *batch size*. Por ejemplo, si se dispone de 1000 unidades de datos y el tamaño de lote es 100, en cada *epoch* se ejecutarían 10 iteraciones y cada una de estas se entrenaría con un lote distinto de 100 datos. En este caso, la rapidez del entrenamiento viene dada porque no es necesario cargar todos los datos en memoria a la vez cuando se ejecuta una *epoch*, ya que esta está dividida en iteraciones. Además, otra de las ventajas que aporta esta solución es que los parámetros de peso y sesgo son actualizados más veces [31].

4.4.4. Redes Neuronales Convolucionales o *Convolutional Neural Network* (*CNN*)

Las Redes Neuronales Convolucionales o *Convolutional Neural Network* (*CNN*) son un tipo de redes neuronales que se emplean en el procesamiento de datos que vienen dados en forma de cuadrícula, como pueden ser las imágenes.

Uno de los mayores problemas a la hora de utilizar una imagen como entrada de una red neuronal es el gran tamaño que puede llegar a tener dicha entrada. Este tamaño viene dado por las dimensiones de la imagen [26].

Para resolver este problema, las *CNN* usan capas convolucionales que se encargan de extraer características de los datos de entrada. Las primeras capas de convolución tratan de extraer características de bajo nivel como

bordes, colores, líneas y curvas. Las siguientes capas de convolución capturan características cada vez más complejas [32].

Este tipo de arquitectura consigue más precisión y sencillez dividiendo el problema en porciones para luego combinarlas, lo cual permite a las redes aprender representaciones de formas complejas a partir de otros patrones más simples.

Los modelos de este tipo están formados por distintos bloques convolucionales que contienen una capa convolucional seguida de una capa de submuestreo o *pooling*.

La capa de convolución se encarga de aplicar un filtro o *kernel* (matriz de pequeña dimensión) a la imagen de entrada y desplazarlo por ella, reduciendo así el tamaño de la salida generada. En cada paso, la función *kernel* se aplica a la entrada haciendo que, finalmente, la imagen se transforme en lo que se conoce como mapa de características o *feature mapping* [33]. En realidad, en vez de aplicar un solo *kernel*, se emplea un conjunto de varios de ellos al que se denomina conjunto de filtros. Es decir, si se aplican 8 filtros en una capa de convolución se generarían 8 matrices de salida, las cuales formarían el mapa de características [34].

Por otra parte, se debe aplicar una función de activación a cada una de las capas convolucionales. Según varios estudios [32] y [34], la más usada en estos casos es *ReLU*, que se explicará con más detenimiento en la sección 6.3.1. Normalmente, esta función se aplica añadiendo una capa de este tipo tras la capa de convolución.

A modo de ejemplo, en la Figura 10: Ejemplo de kernel de tamaño 3x3 [35].Figura 10 se puede observar un *kernel* de 3x3 con los valores por los que se va a multiplicar cada uno de los elementos de la matriz de entrada en cada paso.

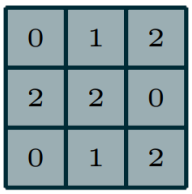

Figura 10: Ejemplo de kernel de tamaño 3x3 [35].

Además, en la Figura 11 la se muestra la aplicación del *kernel* a una matriz de 5x5.

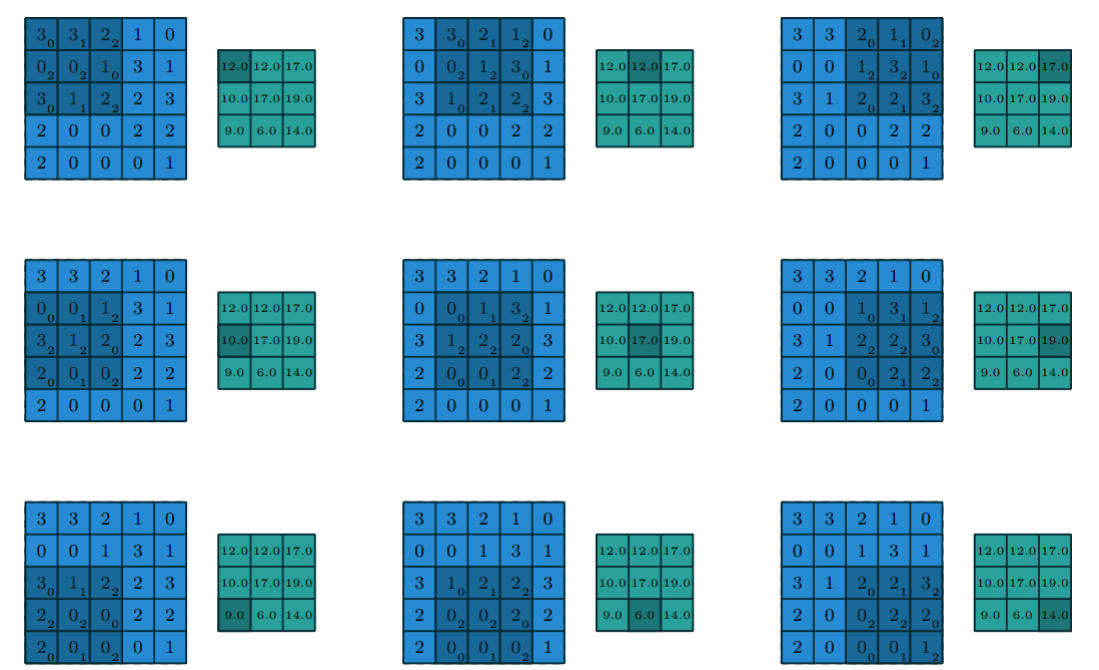

Figura 11: Ejemplo de convolución en la que se aplica el kernel de la Figura 10 a una matriz de 5x5, generando una salida de 3x3 [35].

La capa de submuestreo o *pooling* trata de reducir la dimensión de los datos de salida de la capa de convolución anterior intentando que prevalezcan las características más relevantes detectadas por cada filtro. Para ello, se debe especificar el tamaño de la zona que se desea reducir. Además, es necesario determinar qué tipo de submuestreo se va a realizar, como *Max Pooling* (calcula el máximo valor de la zona) o *Average Pooling* (calcula la media de los valores de la zona) [33]. La diferencia entre aplicar cada uno de ellos se muestra en la Figura 12.

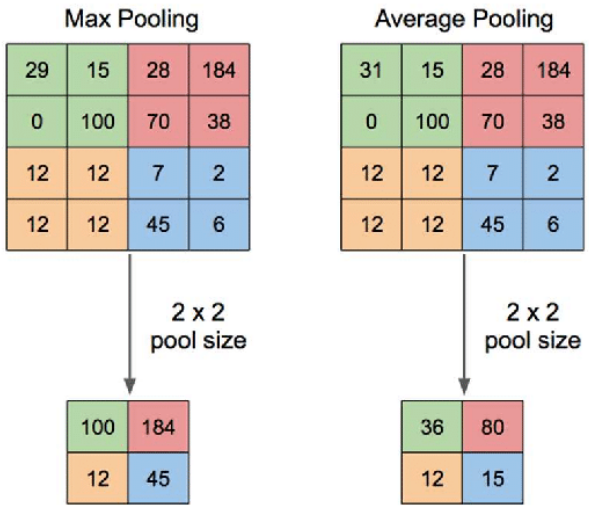

Figura 12: Resultado de aplicar Max Pooling y Average Pooling de tamaño 2x2 a una matriz de 4x4 [36].

Tras la primera convolución (Figura 13), se suceden más convoluciones que toman como entrada la salida generada en el paso previo. Los siguientes mapas de características serán capaces de reconocer formas cada vez más complejas.

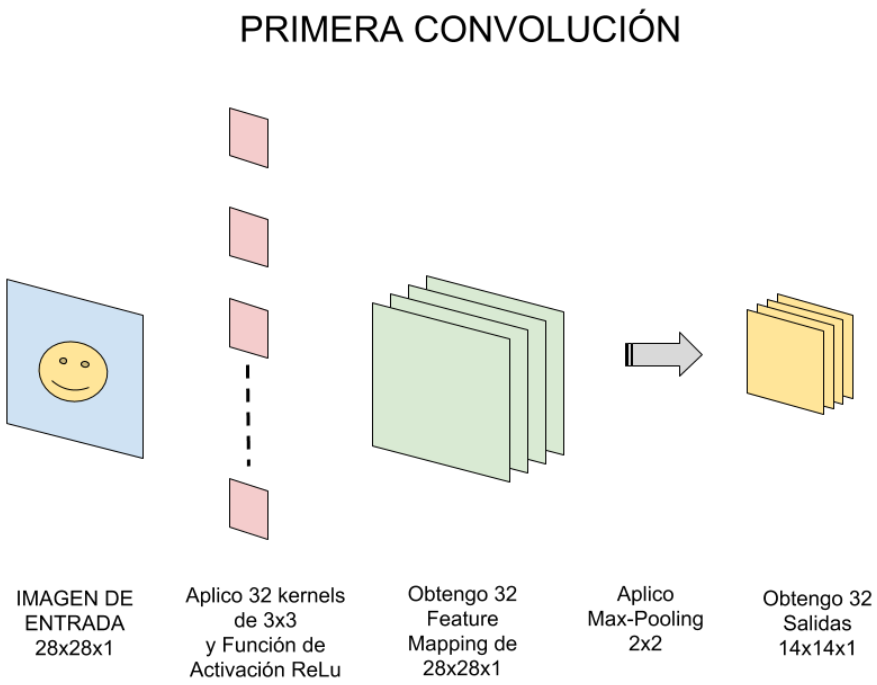

Figura 13: Primera convolución [34].

Otra de las ventajas que ofrece el uso de *CNN* es que resisten la traslación. Es decir, si durante el entrenamiento se extrae una característica en una posición concreta de una imagen, al realizar una predicción con una imagen nueva es posible detectar esta misma característica en cualquier posición de la imagen. Por el contrario, conseguir esto haciendo uso de las redes neuronales tradicionales supondría un aumento desmesurado del número de parámetros [37].

4.4.5. Redes Generativas Antagónicas o *Generative Adversarial Network* (*GAN*)

Las Redes Generativas Antagónicas o *Generative Adversarial Network* (*GAN*) son arquitecturas que están formadas por dos redes neuronales: un generador y un discriminador. El generador es entrenado para producir imágenes que parezcan tan reales que consigan engañar al discriminador. Mientras que la función del discriminador es intentar distinguir los datos falsos que hayan sido elaborados por el generador. Ambas partes de la arquitectura intentan superarse entre sí hasta que se alcanza un equilibrio en el que el discriminador no es capaz de detectar qué datos son reales y cuáles no [38].

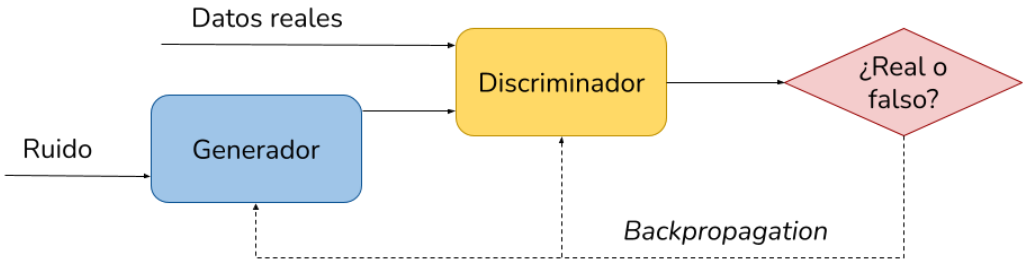

Figura 14: Arquitectura de una GAN.

4.5. Estudios para colorear imágenes usando *Deep Learning*

En cuanto a la imagen digital, existen dos paradigmas posibles para colorear fotografías en blanco y negro: la propagación de la edición guiada por el usuario y la coloración automática basada en datos.

El primer enfoque, dado a conocer por Levin [39], consiste en que el usuario dibuje un punto de color en cada zona de la imagen que tiene un color diferente. A partir de estos trazos y de un proceso de optimización, se genera la imagen coloreada resultante. Sin embargo, como se puede deducir, aunque este procedimiento puede producir muy buenos resultados, requiere una alta intervención del usuario.

Como solución a este problema, algunos autores proponen colorear imágenes a partir de datos. Este procedimiento se puede llevar a cabo de dos maneras. Por un lado, Hertzmann [40] plantea realizar una comparación entre la imagen de entrada en blanco y negro y una imagen a color almacenada en una base de datos, y utilizar los colores de esta última para dibujar sobre la primera.

Por otro lado, otra forma de abordar esta tarea, propuesta por Zhang [41], es aprender, a través de grandes cantidades de datos, la relación entre un píxel de la escala de grises y el color con el que se puede corresponder en realidad.

4.5.1. *CNN* junto a pistas aportadas por el usuario

En este primer artículo [41], los autores plantean un enfoque para resolver el problema del coloreado de fotografías haciendo uso de técnicas de aprendizaje profundo y de la intervención del usuario. El modelo creado se encarga de mapear una imagen en blanco y negro con su correspondiente imagen a color mediante una *CNN* y distinta información aportada por el usuario. Durante el entrenamiento del modelo, se simulan los datos de entrada que aporta el usuario para facilitar el proceso.

En definitiva, el sistema muestra una fotografía en escala de grises y guía al usuario para que decida qué colores cree que se sitúan en cada punto de la imagen. Para ello, sugiere al usuario colores que posiblemente contiene la imagen, basándose en la información proporcionada por usuarios anteriores y los datos a gran escala almacenados (Figura 15).

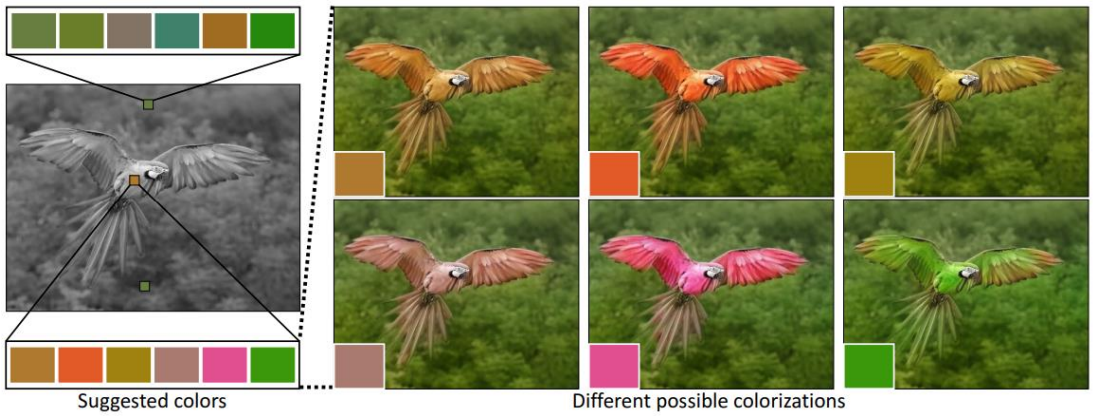

Figura 15: Paletas de colores sugeridas al usuario en distintas zonas de la imagen [41].

ImageNet es un conjunto de datos disponible en la web que incluye 1.3 millones de imágenes a color, y es el que los autores deciden utilizar durante la fase de entrenamiento. En este caso, como en la mayoría de ellos, las imágenes contenidas en el *dataset* se convierten al espacio de color *Lab*, generando así un canal que mide la luminosidad de la imagen (imagen en blanco y negro que se usa como entrada) y dos canales que hacen referencia al color (imágenes que el modelo debe predecir y que, junto con la imagen en blanco y negro de entrada, darán lugar a la fotografía de salida del sistema).

A la hora de elegir una función de pérdida se debe tener en cuenta el problema que se desea solucionar. Algunos desarrolladores como Zhang [41] se decanta por una función de clasificación seguida de un proceso de deducción. Sin embargo, uno de los problemas a los que se enfrentan es que, en ocasiones, dentro de una misma gama de color se encuentran muchos más píxeles con poca saturación que píxeles saturados. Este desequilibrio ocasiona que, tras ejecutar el programa, la imagen resultante tenga zonas con muy poca saturación. Para evitar esto, en la fase de entrenamiento, Zhang lleva a cabo un rebalanceo de clases para tomar las muestras más saturadas de la gama de colores. Como es evidente, usar esta técnica hace que se alcancen soluciones mucho más coloridas.

En este modelo se opta por una versión suavizada de la función *L1 loss*, la cual se aplica primero a cada uno de los píxeles de una fotografía y cuyos resultados se suman para obtener la función pérdida de la imagen completa.

Aunque en el artículo se presentan dos propuestas, nosotros nos centraremos en una de ellas, a la que denominan *Local Hints Network* o Red de Pistas

Locales (en el artículo, las pistas hacen referencia a la información que aporta el usuario cuando interacciona con el sistema).

Este método comienza coloreando las imágenes con colores poco vibrantes, favoreciendo que el usuario intervenga seleccionando los colores que él quiera, aportando cada vez más saturación si es que así se desea. Es así como se resuelve, además, el problema de ambigüedad existente. Asimismo, la función de pérdida elegida, *smooth-L1 loss*, actúa adecuadamente en los casos en los que es necesario realizar estimaciones, evitando de esta manera que se calcule la media cuando hay ambigüedad.

Para comprender totalmente la metodología seguida en este artículo, antes es necesario definir el concepto de tensor.

Un tensor es una estructura matemática multidimensional que almacena datos para un posterior análisis. Los tensores pueden tener cualquier número de dimensiones o ejes, dependiendo de la información que se desea almacenar en ellos. Por ejemplo, un tensor que solo contiene un valor tiene 0 dimensiones, y se denomina tensor 0D; un tensor de una dimensión es lo que se conoce como vector, y un tensor de dos dimensiones se representa como una matriz.

Por otra parte, esta estructura posee otra propiedad llamada forma o *shape*, la cual es representada como un vector de enteros que indican el número de dimensiones que hay en cada eje. Es decir, una imagen de 50x50 píxeles podría interpretarse como un tensor de dos dimensiones o ejes, donde su forma es (50, 50).

A continuación, se describen los puntos clave de la Red de Pistas Locales.

La entrada al sistema como tal, se basa en pistas que el usuario proporciona, las cuales toman la forma de un tensor expresado con $U_l = \{X_{ab}, B_{ab}\} \in \mathbb{R}^{H \times w \times 3}$. En este caso, U_l es un tensor que tiene dimensión 3 y cuya forma viene representada por (H, w, 3), donde H es la altura de la imagen, w es la anchura y 3 es el número de canales de la imagen. Este tensor U_l, a su vez, está compuesto por dos tensores, X_{ab} y B_{ab}.

$X_{ab} \in \mathbb{R}^{H \times w \times 2}$ es un tensor que tiene dimensión 3 y cuya forma viene representada por (H, w, 2). En este caso, el valor 2 hace referencia a los dos canales de color, a y b, que son los valores que toman los puntos o pistas que deja el usuario.

$B_{ab} \in \mathbb{R}^{H \times w \times 1}$ representa una máscara binaria, o bien un tensor que tiene dimensión 2, es decir, una matriz. Dicho tensor denota qué píxeles ha aportado el usuario y cuáles no, y su forma viene representada por (H, w).

La tarea de agrupar datos de entrenamiento adecuados es imprescindible. En este caso, es complicado encontrar datos referidos a la interacción del usuario ya que estos dependen del propio sistema, por lo que se simulan generándolos de manera artificial. Para ello, se toman muestras de distintas zonas de la fotografía y se calcula la media de los valores que toman los píxeles en esta zona. Para elegir las ubicaciones de donde se toman las muestras, se utiliza una función Gaussiana de dos dimensiones, ya que se quiere simular el comportamiento del usuario y, normalmente, este pichará en los píxeles situados en el centro de la imagen.

Para finalizar este proceso, es necesario que la red copie las pistas de color que se han generado o que el usuario ha proporcionado en la imagen de salida. Sin embargo, como la red no está capacitada para hacerlo por sí sola, en el 1% de los casos, se copia el color verdadero en la salida.

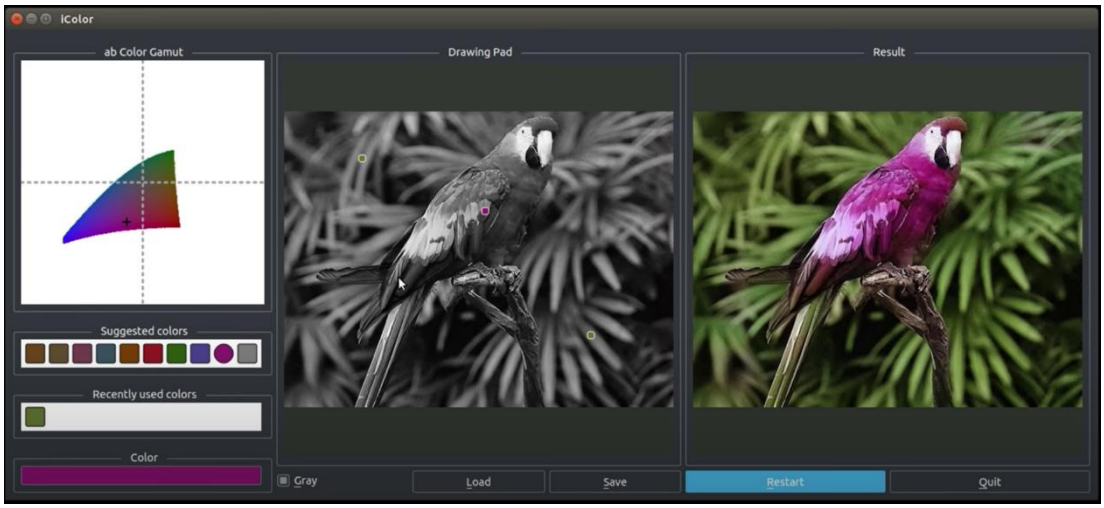

Figura 16: Interfaz de usuario de la aplicación desarrollada en el estudio que utiliza Local Hints Network [42].

Como se muestra en la Figura 16, la interfaz de usuario de la aplicación desarrollada está formada por varios componentes. Uno de ellos es la imagen de entrada en blanco y negro, en la que el usuario puede pinchar para escoger los colores de cada zona que desee colorear. Al lado de esta, se encuentra la fotografía de salida que se colorea en tiempo real según las pistas que esté aportando el usuario. A la izquierda, está el panel en el que se muestra la paleta de colores sugeridos y la gama *ab* basada en la luminosidad, ambos elementos relacionados con el punto en el que el usuario está haciendo clic en ese instante.

Para finalizar, se explica la arquitectura de red que se usa en el desarrollo, la cual se divide en dos ramas: una rama principal y una rama secundaria.

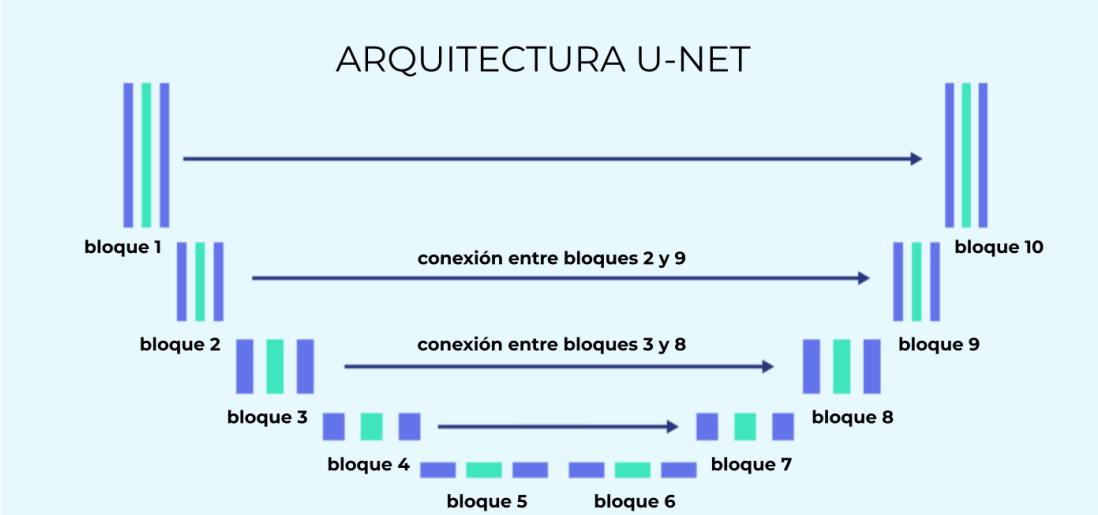

Figura 17: Arquitectura U-Net con 10 bloques de capas conv-relu [43].

Por una parte, la rama principal presenta una arquitectura *U-Net*, como se visualiza en la Figura 17, y está compuesta por 10 bloques de capas convolucionales. En cada uno de los 4 primeros bloques, se duplica la dimensión de las características de los tensores de características y se divide a la mitad la resolución espacial. Además, cada bloque contiene 2 o 3 parejas de *conv-relu*. Por el contrario, en cada uno de los 4 últimos bloques, se reduce a la mitad la dimensión de las características y se duplica espacialmente de manera progresiva, recuperando así las dimensiones del inicio. En los bloques 5 y 6, se usan redes convolucionales dilatadas con factor 2 en vez de dividir la resolución espacial. Por otro lado, para poder recuperar correctamente las dimensiones espaciales del principio, se establecen conexiones entre

bloques, como las existentes entre el bloque 2 y el bloque 9 o entre el bloque 3 y el bloque 8, que se pueden observar en la Figura 17Figura 17.

En cuanto a la técnica, para poder cambiar las resoluciones espaciales, se realizan operaciones de submuestreo y muestreo ascendente, usando un núcleo de 3x3 en las capas convolucionales. Además, tras cada bloque se añade una capa *BatchNorm*, caracterizada por aportar rapidez y estabilidad a la red neuronal.

Por otra parte, en la rama secundaria, las pistas que proporciona el usuario se concatenan con la imagen de entrada en blanco y negro, además de realizarse predicciones para mostrar al usuario la paleta de posibles colores (basada en los puntos del usuario y la imagen de entrada) que puede adoptar cada píxel. Como se puede deducir, en esta parte de la red se aprovechan los parámetros especificados en la rama principal, ya que la tarea de predecir los colores de una imagen completa y la tarea de predecir la paleta de colores que se recomienda al usuario para colorear cada píxel tienen vastas similitudes. Es importante destacar que el comportamiento de la rama secundaria no interfiere en la labor ejercida por la rama principal, por lo tanto, no se utiliza el algoritmo de *backpropagation* dentro de la rama principal con los datos inferidos de la rama secundaria.

Figura 18: Puntos de color aportados por el usuario, resultados de la coloración a partir de estos puntos e imagen original a color [41].

En base a lo expuesto anteriormente, se puede deducir que, aunque el modelo propuesto tiene ventajas, como la de incorporar la interacción del usuario aprovechándola para realizar recomendaciones a este y colorear la fotografía,

existen casos en los que el comportamiento del sistema es inadecuado. Es decir, como se puede observar en la Figura 18, la elección del color realizada por parte del usuario en uno de los puntos del primer plano de la imagen puede inducir a que el color del fondo cambie. Asimismo, puede ocurrir que la propagación de la información dada a través de las pistas del usuario no se efectúe correctamente y, por lo tanto, el coloreado de la imagen no se aproxime lo suficiente al resultado que se desea obtener, como se muestra en la Figura 19Figura 19.

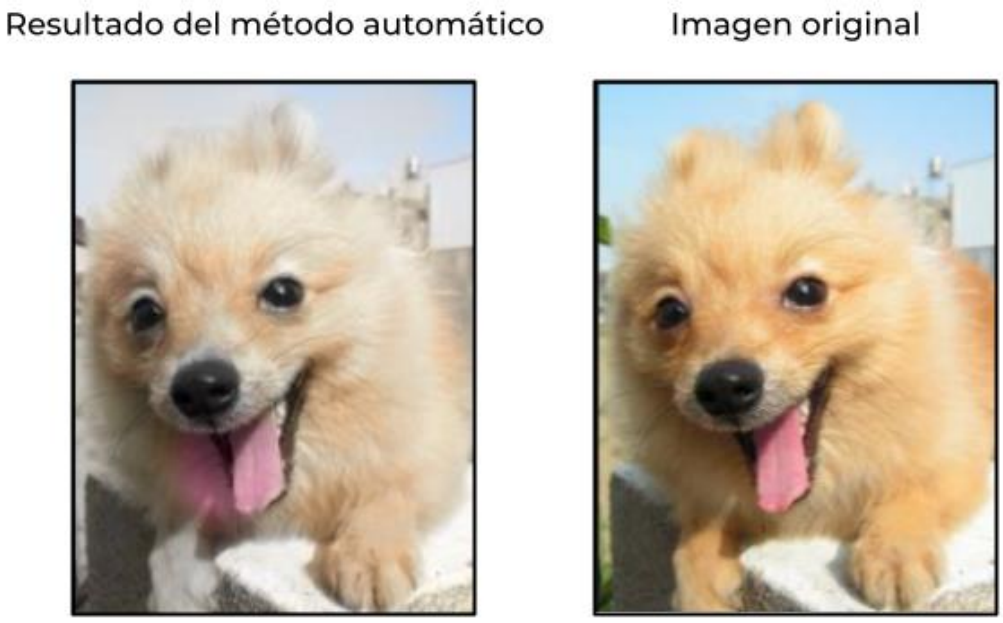

Figura 19: Resultado de ejecutar el método automático de coloración e imagen original a color [41].

Por otro lado, hay casos en los que es realmente complicado definir los límites de los elementos de una imagen, por lo que el usuario debe ser el que determine cada parte de forma concreta. Sin embargo, si se busca gran precisión a la hora de colorear la fotografía es recomendable decantarse por utilizar *Photoshop*, ya que la aplicación desarrollada está pensada para una corta interacción del usuario y, para resolver problemas como los mencionados anteriormente, se necesita tanto precisión como tiempo.

Por último, es importante recalcar que con la solución propuesta se obtienen resultados favorables. No obstante, una de las ampliaciones que es considerada en el artículo es imitar mejor al usuario a la hora de simular las pistas.

4.5.2. *U-Net* y *GAN* condicional sin preentrenamiento

En el siguiente artículo [12], se propone un modelo basado en combinar una *GAN* condicional con otra función de pérdida o *loss function* llamada *L1 loss*, también conocida como *mean absolute error*.

Si se utilizase la función *L1 loss* por sí sola, el modelo colorearía de marrón o gris muchos de los píxeles cuando no tuviese claro qué color se acerca más al verdadero. Esto se debe a que, al calcular la media, el modelo elegiría este color para reducir la *L1 loss*.

En primer lugar, se escoge el *dataset COCO* de la web, el cual contiene 20000 imágenes, aunque solo se utilizan 8000, y se convierten al espacio de color *Lab*. En este caso, la *GAN* está compuesta por un generador, el cual toma una imagen en blanco y negro (canal *L*) como entrada y da lugar a una imagen con dos canales en color (*a* y *b*), y un discriminador, que se encarga de unir los tres canales en una imagen y decidir si ésta es falsa o real.

La condición que convierte a esta red neuronal en condicional es la imagen en blanco y negro proporcionada tanto al generador como al discriminador.

Por una parte, el generador desarrollado es una *U-Net*, algoritmo de segmentación semántica cuya estructura en forma de U viene dada por añadir módulos de muestreo descendente y ascendente a partir de un módulo central.

Por otra parte, el modelo propuesto como discriminador se basa en una arquitectura sencilla formada por bloques apilados de capas *Convolutional-BatchNormalization-LeakyReLU*.

Figura 20: Resultado del primer estudio [44].

Como se puede ver en la Figura 20Figura 20, este modelo es capaz de colorear correctamente los elementos más comunes de las imágenes como el cielo, la hierba, etc. Sin embargo, no ocurre lo mismo al colorear elementos no tan comunes.

Por otra parte, en algunos de los resultados aparecen manchas de color, como en primera imagen de la segunda fila. La obtención de resultados tan malos como estos se debe a que el *dataset* con el que se ha entrenado el modelo no es lo suficientemente grande [44].

4.5.3. *U-Net* y *GAN* condicional con preentrenamiento

Como solución al problema descrito anteriormente, en este mismo artículo [12], se presenta una nueva estrategia que consiste en utilizar una red previamente entrenada en el generador haciendo uso del aprendizaje supervisado, con el fin de dotar al generador de conocimiento antes del entrenamiento.

Este método de preentrenamiento será utilizado dos veces:

- En primer lugar, la red *ResNet18*, que es un modelo de clasificación preentrenado, servirá como pilar de la *U-Net* implementada en el generador.

- En segundo lugar, el generador completo será preentrenado usando *L1 Loss*.

Una de las ventajas encontradas respecto al estudio del apartado anterior es que, al haber preentrenado el generador, el número de *epochs* necesarias para entrenar el modelo disminuye notablemente. Mientras que en el primer ejemplo eran necesarias 100 *epochs*, en este modelo solo se utilizan 20, lo cual hace que se agilice el proceso de aprendizaje de la red.

El autor se percata de que, tras el preentrenamiento con la función *L1 Loss*, antes del entrenamiento *adversarial*, el modelo ya es capaz de producir muy buenos resultados. Sin embargo, éste funciona mejor en imágenes con objetos comunes, ya que, al intentar predecir colores de objetos raros, tiende a usar tonos grises (Figura 21).

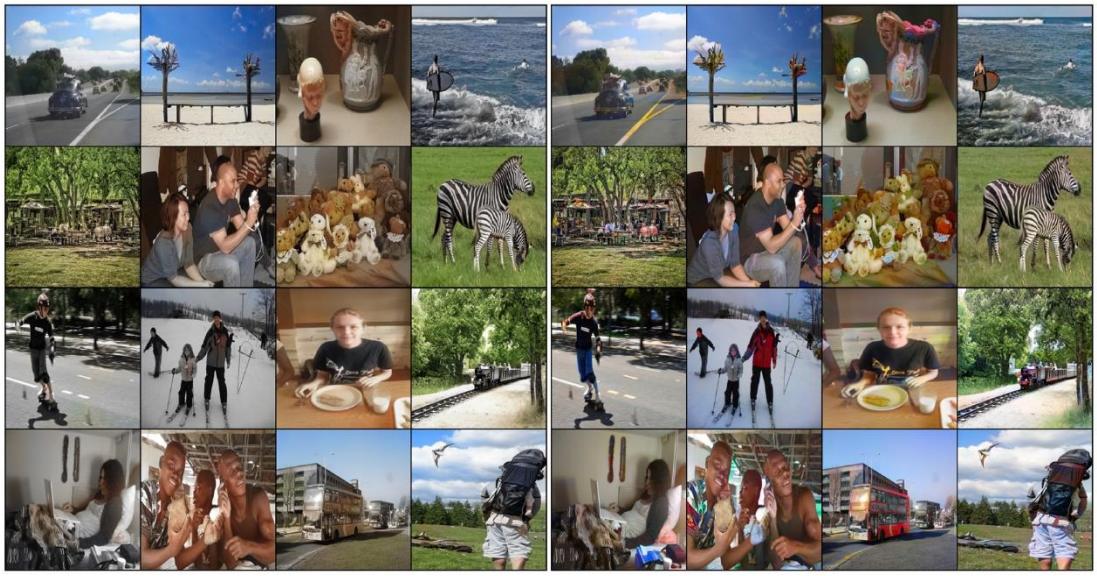

Figura 21: Diferencias entre los resultados obtenidos utilizando la U-Net sin entrenamiento antagónico (izquierda) y la U-Net con entrenamiento antagónico (derecha) [44].

Tal y como se ha podido comprobar, se puede deducir que el preentrenamiento de un generador junto a un entrenamiento antagónico posterior es una forma sólida y factible de abordar la tarea de coloración de imágenes obteniendo buenos resultados.

4.5.4. Conclusiones

En la mayor parte de los artículos que explican cómo resolver la tarea de coloración de imágenes, los desarrolladores convierten el contenido del

conjunto de datos que hayan elegido al espacio de color *Lab*. Aunque los conjuntos de datos elegidos son distintos en cada caso, todos tienen en común que incluyen imágenes a color.

Una vez se han convertido estas imágenes al espacio de color *Lab*, el canal *L*, que representa la luminosidad y, por tanto, es una imagen en blanco y negro, se utiliza como entrada del modelo. Los canales *a* y *b*, sin embargo, sirven para entrenar el modelo, siendo éstos las salidas que se desean obtener.

Por lo tanto, tras haber concluido la fase de entrenamiento del modelo, la salida que éste genera está formada por los canales *a* y *b*, relativos a los colores.

El último paso es unir la imagen en escala de grises que se usa como entrada con los canales *a* y *b* obtenidos, generando así la imagen a color que se predice.

La lectura de estos ejemplos junto a los conocimientos previos sobre redes neuronales facilita saber cuáles son las arquitecturas que mejor solventan el problema del coloreado de fotografías. Tanto las *CNN* como *GAN* son redes construidas para resolver cuestiones relacionadas con imágenes, por lo tanto, se puede intuir que son las indicadas para actuar en este caso.

Además de las soluciones explicadas en estos casos, la mayoría de los estudios que resuelven este problema utilizan arquitecturas con bloques convolucionales como los definidos en el primer ejemplo. Es decir, bloques de dos tipos: unos que operan con submuestreo y otros que utilizan el muestreo ascendente. Los primeros están formados por capas de convolución seguidas de capas *ReLU,* para aumentar la dimensión de las características, a los que se añade también una capa de *BatchNorm*, que estabiliza la red. El otro tipo de bloque trata de reducir la dimensión de características mediante operaciones de muestreo ascendente, recuperando así el tamaño que se desea obtener en la salida.

Por otra parte, tras leer la tercera forma de resolver el problema, se puede deducir que utilizar redes neuronales preentrenadas agiliza el proceso de

aprendizaje, ya que no es necesario utilizar tantas *epochs* para que el modelo adquiera suficiente conocimiento.

5. METODOLOGÍA

5.1. Fases de creación de un modelo de *Deep Learning*

Antes de presentar la forma en que se ha abordado este problema, es necesario tener claro los pasos a seguir para crear y utilizar un modelo de aprendizaje profundo. En el libro [45], se proponen las fases que se deben tener en cuenta durante este proceso, que son las enumeradas y descritas a continuación:

1. Decidir un objetivo
2. Recoger un conjunto de datos
3. Crear un modelo
4. Entrenar el modelo
5. Convertir el modelo
6. Hacer una inferencia
7. Resolver problemas

El primer paso que se debe dar es, sin duda, aclarar cuáles son los objetivos del proyecto. Hay que decidir qué se quiere prever, qué datos recoger y qué arquitectura utilizar. Una vez aclarado esto, se puede empezar a crear el conjunto de datos.

La recopilación de un conjunto de datos consta de tres pasos básicos: la selección de datos, la recogida de estos y el etiquetado.

En primer lugar, es de considerable importancia seleccionar los datos, ya que es conveniente entrenar el modelo utilizando sólo la información relevante para la resolución del problema.

Una vez seleccionados los datos, se tiene como objetivo recoger una multiplicidad de datos que puedan representar toda la gama de condiciones y eventos que pueden existir en el sistema. La diversidad de estos, por lo tanto, ayudará a representar todos los escenarios posibles.

Después de la recogida de los datos, se puede proceder al etiquetado, el proceso que asocia cada dato con una clase.

Llegados a este punto, ya se ha creado el conjunto de datos y, por tanto, se pasa a la creación de la arquitectura del modelo.

Para decidir de qué modo estructurar la arquitectura, es necesario tener en cuenta varios factores: qué problema se desea resolver, de qué tipo de datos se dispone y cómo pueden ser utilizados para alimentar el modelo. Además, es necesario tener en cuenta las características de las que dispone el dispositivo con el que se está tratando: los modelos muy grandes, por ejemplo, requieren más memoria, que no todos los dispositivos tienen.

Posteriormente, el modelo será alimentado con datos en un proceso conocido como entrenamiento o *training*. El entrenamiento es un proceso mediante el cual, dada una determinada entrada, un modelo aprende a producir una salida correcta. Este proceso se consigue alimentando el modelo con los datos y haciendo pequeñas modificaciones para que el modelo sea lo más preciso posible.

Una vez obtenido un modelo que funcione bien, es posible convertirlo en el formato que se desee y, a continuación, hacer inferencias y evaluar cualquier problema del sistema.

5.2. Herramientas

5.2.1. Lenguaje de programación – *Python*

El lenguaje de programación que ha sido utilizado para llevar a cabo la implementación es *Python*. Esta decisión viene dada tanto por la sencillez del lenguaje como por lo ligado que está al desarrollo de arquitecturas de IA [46].

Se listan a continuación algunas de las características que ofrece *Python*:

- Es un lenguaje multiplataforma.
- Permite la programación orientada a objetos.
- Cuenta con una sintaxis clara y sencilla, incluso se podría decir que parece pseudocódigo.

- Soporta el tipado dinámico, es decir, no es necesaria la asignación de tipo a la hora de declarar las variables.
- Proporciona una amplia cantidad de librerías, entre las que destacan las relacionadas con la Inteligencia Artificial.

5.2.2. Entorno de desarrollo – *Google Colab*

El entorno de desarrollo que se ha elegido para la implementación es G*oogle Colab*, que es una herramienta ampliamente usada para resolver problemas relacionados con el *Machine Learning*.

Este entorno de trabajo proporciona la posibilidad de crear cuadernos o *notebooks* divididos en celdas que se pueden ejecutar por separado, lo cual agiliza y facilita la ejecución.

Una cualidad destacable de la herramienta es que permite al usuario beneficiarse del *hardware* de *Google*, sin tener en cuenta el *hardware* del que dispone el equipo. Esto se debe a que el código del *notebook* se ejecuta en los servidores de la nube de *Google*, por lo que se utilizan la *GPU* y la *TPU* de la plataforma a través de la *web*.

Otra de las propiedades que proporciona *Colab* es que incluye muchas de las librerías que se emplean en el desarrollo de redes neuronales sin que sea necesaria una instalación previa. La librería más utilizada en el ámbito del aprendizaje automático, la cual viene incluida por defecto en *Colab*, es *Tensorflow* [47].

6. IMPLEMENTACIÓN Y DESARROLLO

6.1. Obtención del conjunto de datos

El *dataset* escogido se extrae de *Kaggle* [48], plataforma que ofrece una extensa variedad de conjuntos de datos destinados a ser utilizados en tareas de aprendizaje automático.

Concretamente, el conjunto elegido [49] se divide en dos directorios: uno de entrenamiento (*training_set*) y uno de validación (*validation_set*), aunque este último se usará como conjunto de prueba. Cada uno de ellos incluye imágenes de 5 estilos de arte distintos separadas en carpetas: *drawings* (dibujos), *engraving* (grabados), *iconography* (iconografía), *painting* (pintura) y *sculpture* (escultura). Una de las razones por las que se elige este conjunto es que ha sido utilizado en varios casos para resolver el mismo problema, por lo que se pueden realizar comparaciones con lo implementado por otros desarrolladores o estudiar la forma en la que otros han desarrollado otras arquitecturas.

En este caso, como los recursos son limitados, solo se utilizan los datos contenidos en *painting*. Esta decisión viene dada por dos motivos:

Por una parte, las imágenes de este tipo presentan homogeneidad entre ellas, lo cual ayudará a que los resultados que se obtengan al realizar las pruebas sean mejores.

Por otra parte, el número de imágenes que incluye, 2128 archivos en el conjunto de entrenamiento y 236 en el de prueba, es adecuado: es suficiente para la fase de entrenamiento y no es excesivo para los recursos de los que se dispone.

Algunos ejemplos de las imágenes contenidas en el conjunto de datos del entrenamiento se muestran en la Figura 22.

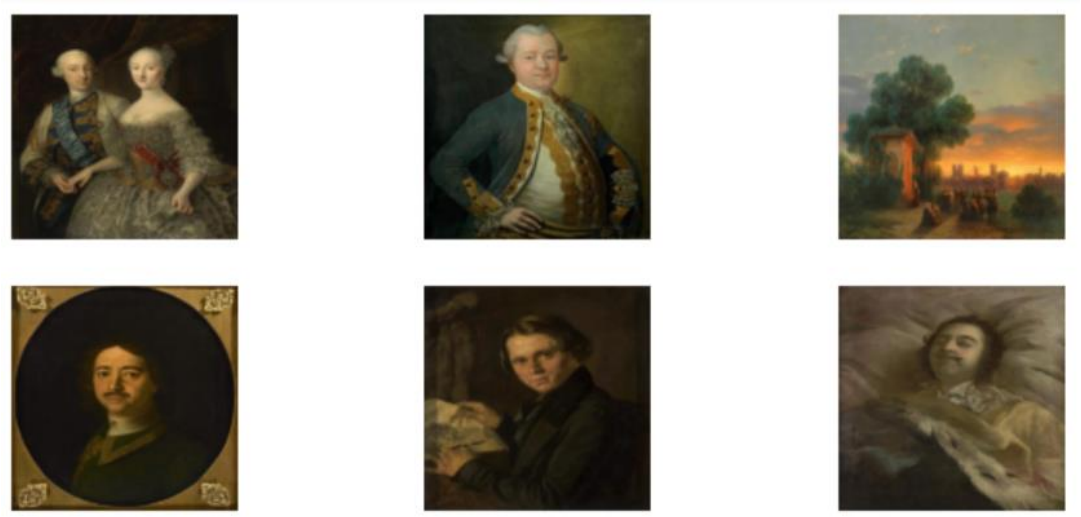

Figura 22: Ejemplos de imágenes contenidas en la carpeta painting del dataset.

6.2. Preprocesamiento del conjunto de datos

Antes de entrenar la red, es necesario un procesamiento previo de los datos escogidos. En este caso, este preprocesamiento se realiza tanto durante la obtención de los datos como tal, como tras haberlos extraído, y consta de los siguientes pasos.

Es importante mencionar que dentro del *dataset* existen archivos que no son imágenes como tal y, por lo tanto, no resultan de utilidad para el posterior entrenamiento del modelo. Por esta razón, el primer paso del preprocesamiento consiste en recorrer todos los archivos contenidos en el conjunto de datos y almacenar en un tensor solo aquellos que pueden ser leídos como imágenes.

Además de esto, antes de guardar cada elemento en el tensor, es necesario redimensionarlo. Es decir, cada una de las imágenes es convertida a unas dimensiones especificadas, que en este caso son 64x64, haciendo que el conjunto de datos sea lo más homogéneo posible. Este paso es imprescindible para entrenar la red ya que esta tomará una entrada con unas dimensiones concretas, por lo que todos los datos de entrada deben tener el mismo tamaño.

Las imágenes digitales que han sido almacenadas vienen representadas por píxeles. Cada uno de ellos está formado por tres canales (*R*, *G* y *B*). Por lo tanto, cada píxel contiene tres valores que oscilan entre 0 y 255, uno para

cada canal. Para convertir estos valores en cifras que estén entre 0 y 1, el contenido del tensor se divide entre 255, asignándole el tipo *float32* a cada uno de sus elementos.

Como en la mayoría de los casos en los que se resuelve la tarea de *image colorization*, las fotografías son convertidas de *RGB* al espacio de color *Lab*. La razón de esta conversión es que, en este espacio de color, las imágenes son representadas con tres canales: uno para el grado de luminosidad y dos para los colores. Esto es realmente útil para introducir el canal de luminosidad, que es una imagen en blanco y negro, como entrada de la red, y tener que predecir solamente dos canales, en vez de tres, como ocurriría si se usase *RGB*.

En primer lugar, se utiliza la función *rgb2lab* de la librería *skimage* [50] para realizar la conversión del tensor que contiene las imágenes.

Los valores contenidos en los canales *a* y *b* están dentro del intervalo que va de -128 a 128. Como se explicará en la sección x, los valores de los píxeles de los canales *a* y *b* que hayan sido predichos por el modelo estarán dentro del intervalo [-1,1]. Para facilitar la comparación entre los valores reales de los píxeles con los valores que se predicen y evaluar cómo funciona la red es necesario que los píxeles de los canales de la imagen real estén también dentro del intervalo [-1,1]. Por esta razón, se divide entre 128 el tensor *ab* que resulta como salida [51].

Tras esto, se generan tres tensores como salida: el primero contiene el canal *L* (luminosidad), el segundo contiene los canales *ab* (colores) y el último incluye los tres canales.

En la Figura 23 se muestran los tres canales (*L*, *a* y *b*) de algunas de las imágenes del conjunto de datos.

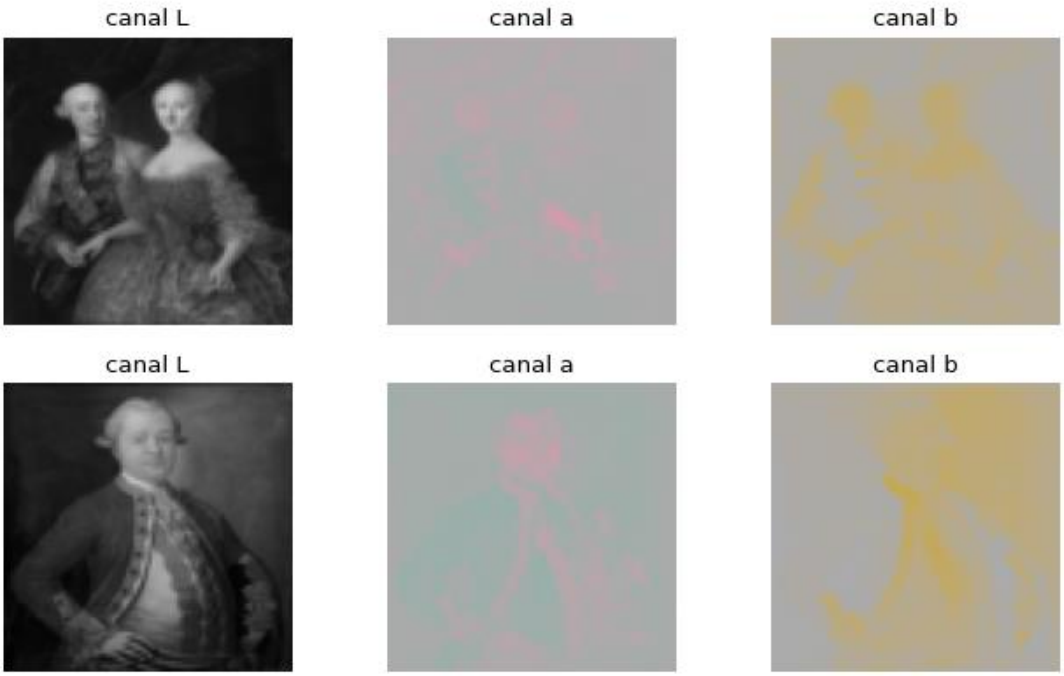

Figura 23: Canales L, a y b de dos de las imágenes de entrenamiento convertidas a Lab.

6.3. Entrenamiento de la red

6.3.1. Arquitectura base

Tras la investigación realizada del estado de la cuestión, se decide abordar el problema utilizando una Red Neuronal Convolucional. Para ello, se parte de la arquitectura de *autoencoder* propuesta en [52].

Los autocodificadores o *autoencoders* (Figura 24) son un tipo de red neuronal que está formada por dos partes: un *encoder* y un *decoder*. El *encoder* es una red neuronal que comprime la entrada para obligar al modelo a aprender a extraer las características más relevantes de los datos de entrada dando lugar a datos con una dimensión eficiente. La salida obtenida se introduce como entrada del *decoder* para regenerar la entrada original [53].

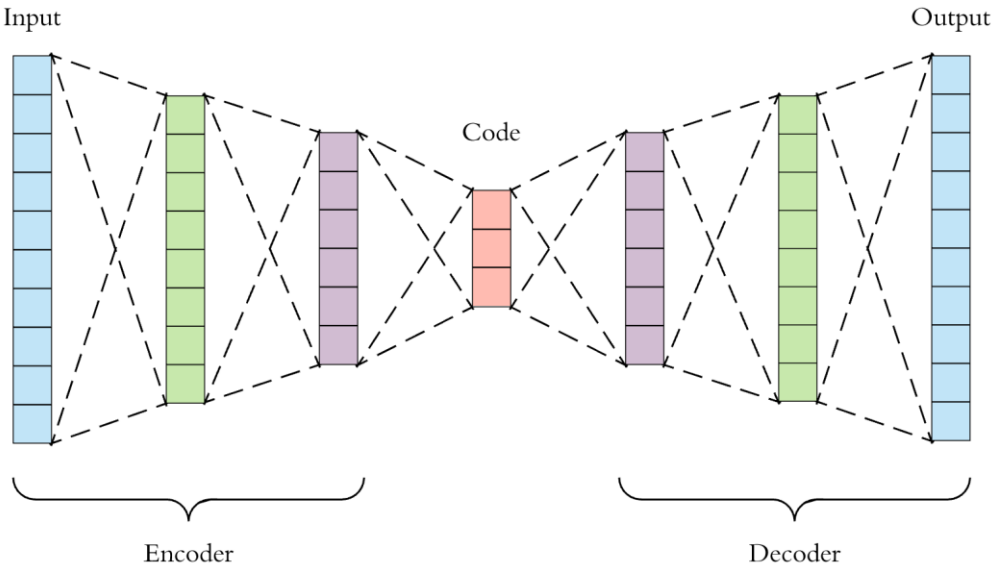

Figura 24: Arquitectura de un autoencoder [53].

En nuestro caso, se construye un modelo secuencial al que se le añade una primera capa *Input*, cuya dimensión de entrada (64x64x1) coincide con la del tensor *L*, es decir, viene dada por las dimensiones de la imagen y el único canal que contiene.

A continuación, se agregan los distintos bloques convolucionales que forman el *encoder*.

El primero de ellos está compuesto por una capa convolucional a la que se le asignan distintos parámetros, entre los que se encuentra el número de filtros, y una capa *LeakyReLU*, que actúa como función de activación no lineal de la capa anterior.

Una de las funciones de activación más conocidas es la *ReLU*, la cual puede tomar valores entre 0 e infinito, según se supere o no un umbral.

LeakyReLU es una variante de la función *ReLU* que proporciona más estabilidad, por lo que el proceso de aprendizaje se agiliza. Esta función, sin embargo, puede tomar valores por debajo de 0, haciendo que, cuando no se supere el umbral, la función tenga una pequeña pendiente para los valores negativos en lugar de una pendiente plana como ocurre en *ReLU*.

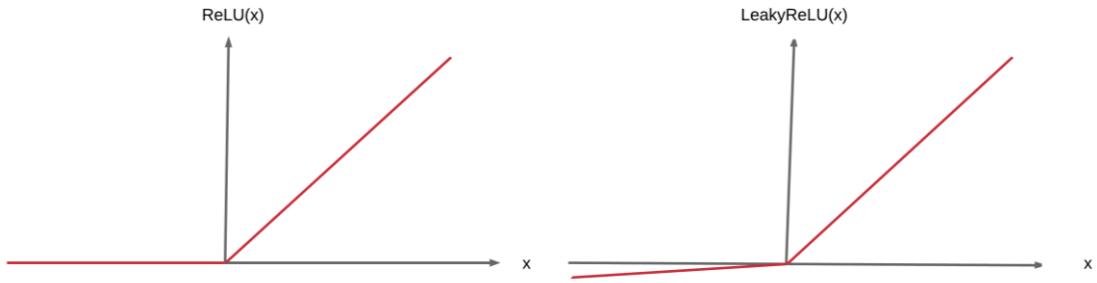

Figura 25: Comparación entre las gráficas que representan las funciones de activación ReLU y LeakyReLU [54].

Según lo expuesto en [32], [34] y [55], la función de activación que se suele aplicar sobre el mapa de características resultante de una capa de convolución es *ReLU*, excepto en las últimas capas, donde se aplica la Tangente Hiperbólica (*TanH*) para proporcionar valores de píxeles apropiados de acuerdo con los valores reales de deformación. Es decir, *TanH* siempre devuelve un resultado que está dentro del intervalo que va de -1 a 1. Como se comentó previamente en la sección 6.2, es posible comparar los resultados que han sido predichos con los colores de la imagen real (Figura 26), porque los píxeles contenidos en los canales a y b de esta imagen se encuentran en el intervalo [-1,1].

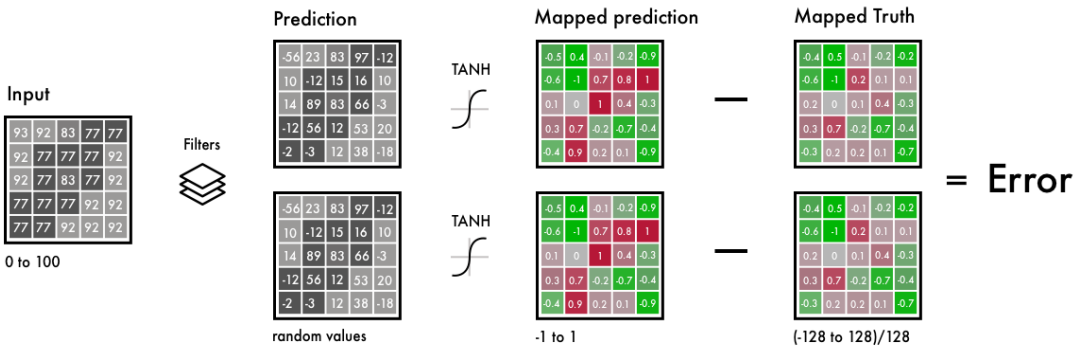

Figura 26: Comparación entre la salida del modelo (se le aplicó la función TanH) y la imagen real (fue dividida entre 128) [51].

Por otro lado, en [56] se explica que *LeakyReLU* genera los mismos o mejores resultados que *ReLU*. Además, *LeakyReLU* puede actuar en arquitecturas en las que *ReLU* falla. Un claro ejemplo sería un modelo que incluye "cuellos de botella", es decir, capas muy estrechas con un número reducido de neuronas.

En la arquitectura que se va a elaborar, la reducción de la dimensionalidad tras las operaciones de convolución y *pooling* provoca cuellos de botella.

Además, una de las características fundamentales de la función *LeakyReLU* que hace que sea de gran utilidad en las *CNN* es que no activa todas las neuronas al mismo tiempo, por lo tanto, durante la fase de *backpropagation*, solo cambian los pesos y sesgos de algunas neuronas.

Por todo esto, *LeakyReLU* es la función de activación que se aplica tras cada capa convolucional, exceptuando la última, en la que se aplica *TanH*.

Los siguientes bloques, además de contener estas mismas dos capas (convolucional y *LeakyReLU*), pueden incluir una capa de normalización por lotes o *Batch Normalization* y una capa de *pooling*. Es importante añadir que, a medida que se incorporan más bloques, se incrementa el número de filtros de la capa convolucional del bloque y se reduce la dimensión de las salidas correspondientes.

Para dar estabilidad al modelo y estandarizar la entrada de las capas en cada lote, es necesario que los pesos de la red se mantengan en un rango de valores razonables. Una forma de conseguirlo es incluir una capa de normalización, la cual se encarga de calcula la media y desviación estándar de cada lote de entrada y lo normaliza realizando distintas operaciones.

La capa de *pooling* no es fija en todos los modelos, ya que se elaboran arquitecturas tanto con *max pooling* como con *average pooling* para, posteriormente, comparar los distintos resultados.

Por otra parte, el *decoder* adquiere como entrada los datos de salida del *encoder* y aplica distintas operaciones sobre ellos. Los primeros bloques del *decoder* están compuestos por una capa de *Upsampling*, una convolucional, una *LeakyReLU* y una de *Batch Normalization*. En este caso, el número de filtros en cada bloque se reduce, mientras que el tamaño de la salida de cada capa aumenta.

Una manera de aumentar la dimensión de los datos de entrada es utilizar una capa de *Upsampling*. Para duplicar el tamaño, esta capa repite las filas y columnas de la matriz de entrada, como se puede observar en la [38].

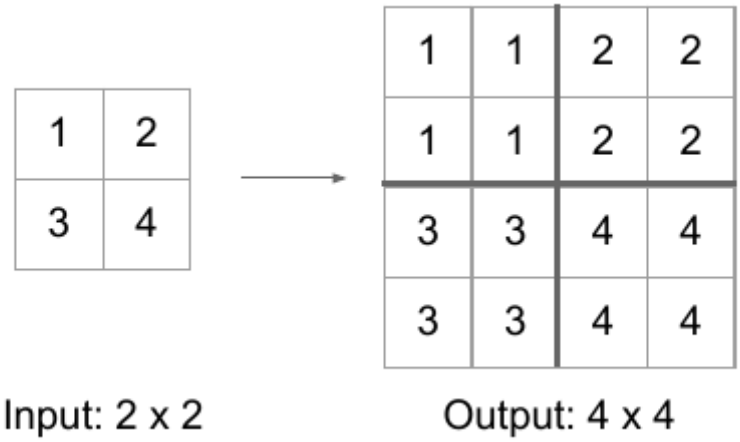

Figura 27: Funcionamiento de la capa de Upsampling [57].

Finalmente, el penúltimo bloque solo incluye las capas de convolución y *LeakyReLU*, y el último bloque, que da lugar a la salida final de la arquitectura, solo contiene una capa de convolución a la que se le aplica la función *TanH*.

6.3.2. Optimización de hiperparámetros

Los hiperparámetros son parámetros definidos antes del entrenamiento para especificar de qué forma se desea entrenar el modelo. Es posible configurar cada uno de los hiperparámetros. Es importante no confundir los hiperparámetros con los parámetros, ya que estos últimos son los que cambian durante el proceso de aprendizaje de la red y no se pueden controlar desde el exterior.

La optimización de hiperparámetros o *Hyperparameter Tuning* consiste en elegir los valores ideales del conjunto de hiperparámetros de un algoritmo de *Machine Learning*. La selección apropiada de los hiperparámetros de las distintas de la red es un paso crucial para predecir resultados con el mayor grado de precisión posible.

Existen distintos algoritmos para realizar esta optimización, como *Grid Search* o *Random Search*. La principal diferencia entre ambos es que mientras que *Grid Search* prueba todas las combinaciones posibles de valores, *Random Search* elige aleatoriamente solo algunos valores con los que realizar las pruebas. Por lo tanto, aunque *Random Search* no siempre devuelve los valores de mejor rendimiento para el modelo, estos sí son lo suficientemente

buenos y se obtienen en un tiempo mucho menor agilizando así el proceso [58].

En el informe [59], se resuelve el problema del coloreado de imágenes utilizando una *CNN*. Para ello, al compilar la red, se emplea el optimizador *Adam* (valores por defecto de β1=0.9 y β2=0.999) con una tasa de aprendizaje o *learning rate* de 0.001 que desciende a 0.0001 cuando la pérdida se aplana (a partir de la *epoch* 150, aproximadamente). El modelo se entrena con 200 *epochs* y un tamaño de lote de 150. Todos estos hiperparámetros se establecen tras aplicar el algoritmo *Random Search* para la optimización de parámetros.

Por otro lado, en el artículo [55], en el que también se resuelve el mismo problema empleando una *CNN*, se vuelve a utilizar el optimizador *Adam*.

En nuestro caso, atendiendo a las resoluciones anteriores y observando que se obtienen buenos resultados, se decide emplear el optimizador *Adam* en todos los modelos.

Por otra parte, se lleva a cabo la optimización de otros hiperparámetros como el *learning rate,* la función de pérdida y el número de *epochs*. Como los recursos son limitados, esta optimización se efectúa únicamente para el primer modelo propuesto y el algoritmo que se emplea para optimizar tanto la tasa de aprendizaje como la función de pérdida es *Random Search.*

La tasa de aprendizaje o *learning rate* controla el tamaño del paso para que un modelo alcance la función de pérdida mínima. Una tasa de aprendizaje demasiado alta agiliza el entrenamiento, pero la probabilidad de encontrar la función de pérdida mínima disminuye. Sin embargo, emplear una tasa de aprendizaje demasiado baja facilita la posibilidad de encontrar la función de pérdida mínima utilizando muchos más recursos e incrementando el número de *epochs* [30], como se puede observar en la Figura 28. Por esta razón, se decide probar con los valores 0.001 y 0.0001 para este hiperparámetro.

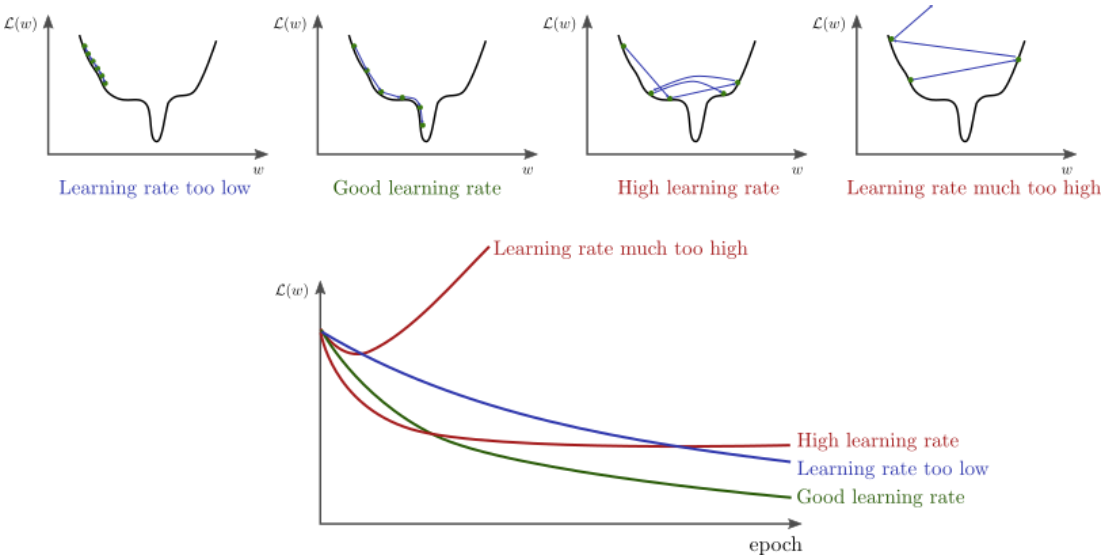

Figura 28: Distintos valores que puede tomar el learning rate y las gráficas correspondientes a este parámetro respecto al número de epochs [60]

Asimismo, se evalúa qué función de pérdida funciona mejor en este modelo entre las dos posibles que se eligen: *mean squared error* y *mean absolute error*. Como se indica en el apartado 4.5, los tres estudios extraídos de los artículos [41] y [12] utilizan la función *L1 loss*, que no es más que otra forma de llamar a la función *mean absolute error*. Además, como el problema que se aborda es de regresión, se han seleccionado estas dos funciones porque son dos de las más utilizadas para estos casos [61].

Por el contrario, el número de *epochs* se ajusta manualmente. Es decir, una vez entrenado el modelo con un número de *epochs* definido, se examina en cuál de ellas se obtiene el mejor resultado para la variable *val_accuracy* y se entrena el modelo con ese número de *epochs*. Sin embargo, para disminuir el tiempo de entrenamiento solamente el segundo modelo es entrenado con este número de *epochs*, los demás son entrenados durante 10 *epochs*. En la demostración [30], se intenta evitar tanto el *underfitting* como el *overfitting* buscando un número apropiado de *epochs* entre 20 y 100. En nuestro caso, al presentar restricciones respecto a la capacidad de procesamiento, el máximo número de *epochs* con el que se realizan pruebas es 50.

Según el libro [37] y el artículo [62], los hiperparámetros de una capa convolucional que resultan importantes a la hora de realizar la optimización son: el número de filtros (*filters*), el tamaño de filtro (*kernel_size*), el tamaño de paso (*strides*) y el relleno (*padding*).

El *padding* consiste en agregar píxeles con valor 0 alrededor de la imagen y se utiliza para que los datos de salida de la capa mantengan la misma dimensión que los datos de entrada [38]. Como en nuestro caso se desea que los primeros bloques reduzcan la dimensión de las características, el *padding* toma un valor fijo y no se ajusta mediante *Hyperparamenter Tuning*.

De nuevo, se toma la decisión de ajustar el tamaño de filtro utilizando *Random Search*. En [59] se comenta que un *kernel size* demasiado pequeño no es capaz de capturar suficiente información. Sin embargo, no tiene sentido que este valor se exceda ya que capturaría demasiada información y no se podría establecer la jerarquía de extracción de características cada vez más complejas. Por esta razón y atendiendo a lo realizado en los estudios, se decide que los valores que va a poder tomar el *kernel size* son 4, 5 o 6.

Por otra parte, por defecto, se decide que el relleno tome el valor 0, el tamaño de paso tome el valor 1 y el número de filtros se cambie manualmente en cada capa. Estos valores se modifican en las diversas arquitecturas que se van a elaborar para apreciar el comportamiento de estas.

Tras finalizar el proceso de optimización de hiperparámetros, se obtienen los siguientes valores óptimos para cada uno de ellos:

- Tasa de aprendizaje: 0.0001
- Función de pérdida: *mean_squared_error*
- Número de *epochs*: 46
- Tamaño de filtro: 5

Es importante recalcar que todos los valores obtenidos a partir de la optimización se obtienen del segundo modelo, pero se aplican a cada una de las arquitecturas construidas, excepto a la primera. Esto es así porque en este proyecto no es posible dedicar el tiempo suficiente a realizar la búsqueda de hiperparámetros adecuados para cada modelo, ya que los recursos se ven limitados. Sin embargo, es evidente que estos valores no serán los óptimos en todas las arquitecturas.

6.3.3. Primer modelo

Es importante destacar que, previamente, para aprender cómo construir un modelo correctamente, se han realizado diversas pruebas con distintos conjuntos de datos, arquitecturas y parámetros, entre otros aspectos. Sin embargo, se ha decidido incluir solo los modelos y *notebooks* más importantes.

El código referente a esta arquitectura se encuentra en el Anexo 1.

El primer modelo consiste en la arquitectura base explicada en el apartado 6.3.1. En este caso, para poder realizar comparaciones posteriores, se decide no llevar a cabo la optimización de hiperparámetros. Por lo tanto, los valores que toman algunos de los parámetros se ajustan manualmente y son los siguientes:

- Tasa de aprendizaje: 0.01
- Función de pérdida: *mean_absolute_error*
- Número de *epochs*: 10
- Tamaño de filtro: 3

Por un lado, el *encoder* de este modelo está compuesto por 5 bloques convolucionales. El primero de ellos contiene una capa convolucional de 16 filtros. A medida que se avanza en los bloques, el número de filtros de la capa de convolución de cada uno de ellos se multiplica por 2. Por lo tanto, las siguientes capas convolucionales cuentan con 32, 64, 128 y 256 filtros.

Asimismo, a cada una de las capas convolucionales de los bloques le sigue una *LeakyReL,* y en el segundo y tercer bloque del *encoder* se añade una capa *BatchNorm* y una de *MaxPooling*. Esta última reduce a la mitad el tamaño del dato de entrada, ya que se detalla que el parámetro *pool size* tome el valor (2, 2).

Por otro lado, el *decoder* está formado por:

- Dos bloques que contienen una capa *UpSampling*, en la que se especifica que debe duplicar la dimensión del dato de entrada, una

convolucional, una *LeakyReLU* y una de normalización por lotes. Las capas de convolución tienen 128 y 64 filtros respectivamente.

- Un bloque que solo contiene una capa convolucional con 32 filtros y una *LeakyReLU*.
- Una última capa convolucional con dos filtros a la que se le aplica la función de activación *TanH*.

En el proceso de entrenamiento, este modelo se entrena con un conjunto de datos de 2042 imágenes, de las que un 0.2 se utiliza para validar el resultado, durante 50 *epochs* sin usar *batches.*

6.3.4. Segundo modelo

El código referente a esta arquitectura se encuentra en el Anexo 2.

La única diferencia entre este modelo y el anterior es que en este se ha llevado a cabo el ajuste de parámetros descrito en el apartado 6.3.2.

6.3.5. Tercer modelo

El código referente a esta arquitectura se encuentra en el Anexo 3.

El tercer modelo utiliza la misma arquitectura que el segundo a excepción de las capas de *pooling*. En este caso, en vez de aplicar capas de *Max Pooling*, se aplican capas de *Average Pooling*.

Con ello, se presupone que los resultados que se van a obtener van a ser mucho menos precisos. Es decir, al aplicar *Average Pooling* se consigue reducir el tamaño de la imagen de entrada calculando la media entre varios píxeles, se entiende que el modelo colorea las salidas optando por colores neutros y poco intensos, producto del previo cálculo de la media. Las imágenes de salida predichas, por lo tanto, tienden a ser mayoritariamente marrones por lo que no se suelen corresponder con la realidad.

6.3.6. Cuarto modelo

El código referente a esta arquitectura se encuentra en el Anexo 4.

El cuarto modelo propuesto consta de una arquitectura como la del segundo modelo. Sin embargo, para reducir el tiempo de ejecución y comparar la

precisión de la red en las primeras *epochs*, el modelo es entrenado durante 10 *epochs* dividiendo el conjunto de datos de entrenamiento en lotes de 100 imágenes.

6.3.7. Quinto modelo

El código referente a esta arquitectura se encuentra en el Anexo 5.

El siguiente modelo se basa de nuevo en una red neuronal como la descrita en el segundo modelo. Por el mismo motivo por el que se construye el cuarto modelo, esta red se entrena con un conjunto de datos que contiene 200 imágenes durante 10 *epochs*.

6.3.8. Sexto modelo

El código referente a esta arquitectura se encuentra en el Anexo 6.

El sexto modelo, sin embargo, aunque parte de una estructura parecida a la del segundo modelo, difiere de ella en el número de filtros especificado en cada capa de convolución.

En esta arquitectura, se decide que las capas convolucionales del *encoder* cuenten con 8, 16, 32 y 64 filtros respectivamente. Por su parte, las capas de este tipo que se encuentran dentro del *decoder* disponen de 32, 16 y 8 filtros.

6.3.9. Interfaz de usuario

Finalmente, se decide desarrollar una sencilla interfaz interactiva incrustada dentro del propio notebook. Para ello, se importa la librería *Gradio* y se define un método que realiza predicciones utilizando un modelo entrenado, al igual que se ha hecho anteriormente.

Como se muestra en la Figura 29, la interfaz está compuesta de dos zonas. En la primera zona, se debe colocar la imagen en blanco y negro de entrada y hacer clic en el botón de enviar.

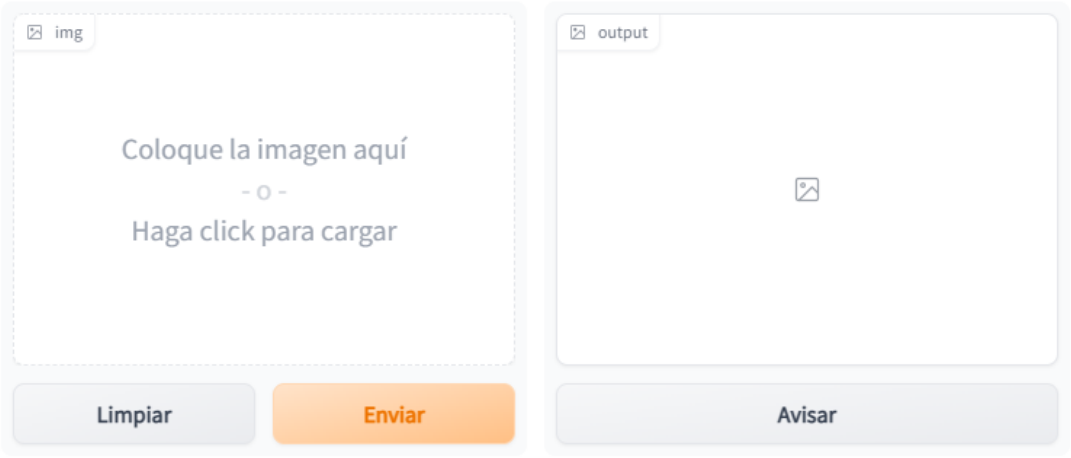

Figura 29: Inicio de la interfaz de usuario.

En la Figura 30, se muestra cómo actúa la interfaz al pinchar sobre el botón enviar. En la segunda zona aparece la imagen coloreada resultante. Además, para subir una nueva imagen solo es necesario pinchar en el botón de limpiar.

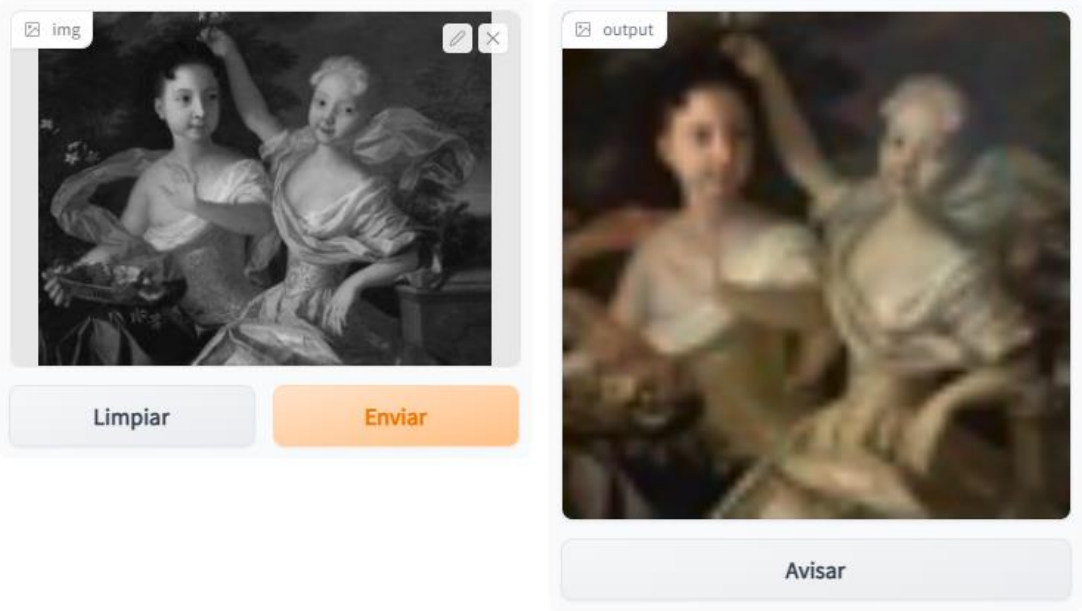

Figura 30: Se muestra el resultado de colorear la imagen con la interfaz de usuario.

7. RESULTADOS Y DISCUSIÓN

A continuación, se muestran las gráficas de precisión y de pérdida de cada uno de los modelos respecto al número de *epochs* durante las que han sido entrenados. En estas dos gráficas se representan los valores que toman las variables de precisión y de pérdida durante en el entrenamiento tanto con el conjunto de entrenamiento como con el de validación.

En la gráfica de precisión, un valor mayor indica un mejor aprendizaje del modelo.

En la gráfica de pérdida, un valor cero hace referencia a que no se cometieron errores en el proceso de aprendizaje por lo que, cuanto menor sea el valor que tomen las variables, mejor habrá aprendido el modelo.

Asimismo, se presentan los resultados de las predicciones en distintas etapas del entrenamiento.

Por último, se comparan todas las arquitecturas y los resultados obtenidos con cada una de ellas.

7.1.1. Primer modelo

El código referente a esta arquitectura se encuentra en el Anexo 1.

El entrenamiento del primer modelo en cada una de las *epochs* tiene una duración aproximada de 200 segundos.

Por una parte, en la Figura 31 se observa que, en las cuatro primeras *epochs*, la precisión de ambos conjuntos aumenta hasta 0.85, aproximadamente. Sin embargo, a partir de esta *epoch*, los valores que toma la precisión del conjunto de entrenamiento tienden a disminuir según se sigue entrenando el modelo.

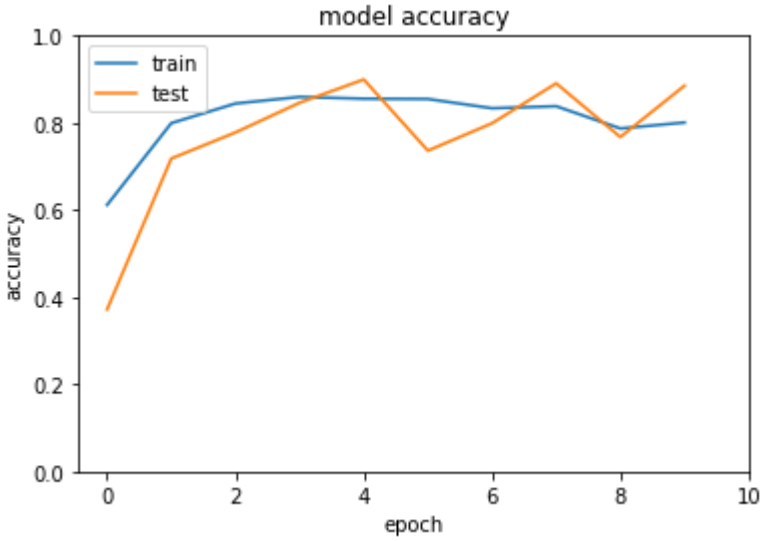

Figura 31: Gráfica de precisión del primer modelo durante 10 epochs.

Por otra parte, en la Figura 32 se muestra que la tendencia de la variable de pérdida del conjunto de entrenamiento a partir de la *epoch* 6 es aumentar.

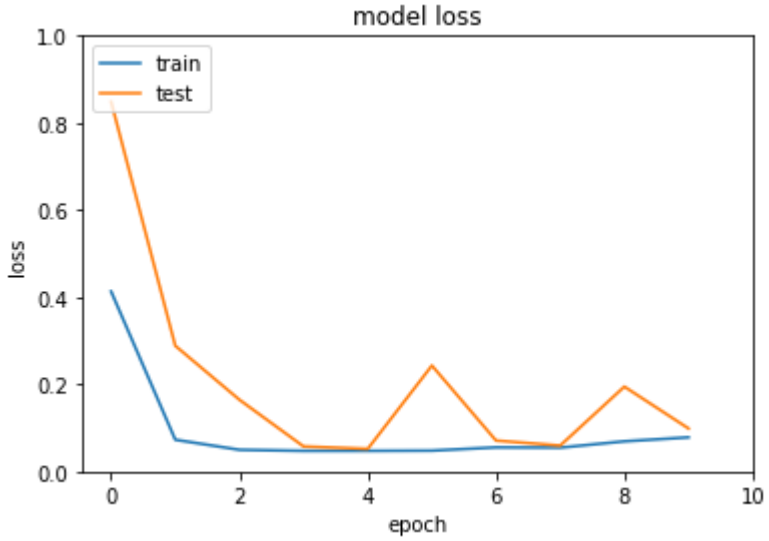

Figura 32: Gráfica de pérdida del primer modelo durante 10 epochs.

Por lo tanto, se puede suponer que este modelo no actuará correctamente, aunque se siga entrenando durante más *epochs*.

Por otro lado, en la Figura 33 se visualiza de qué forma va aprendiendo el modelo, mostrando las predicciones que realiza en distintas *epochs*.

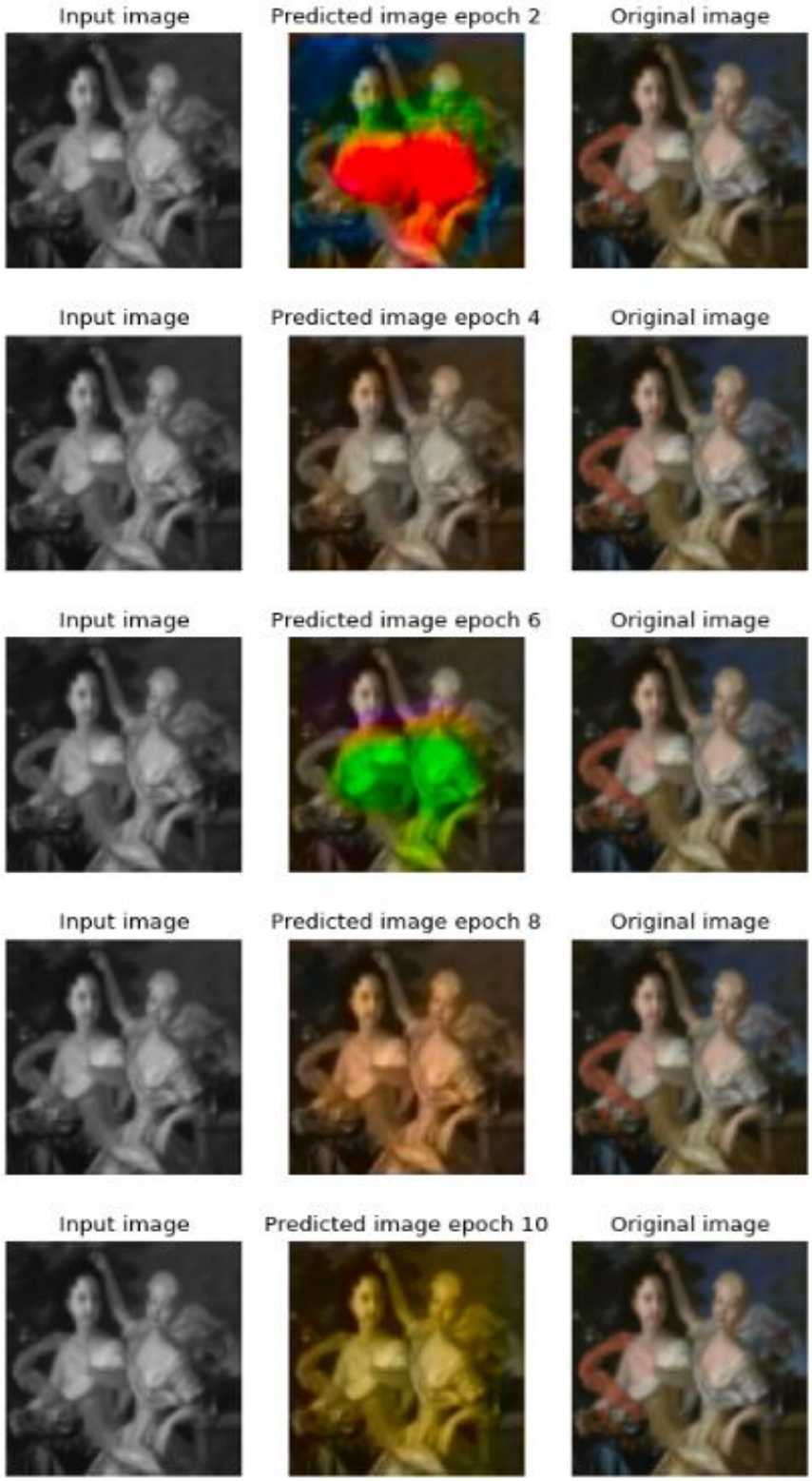

Figura 33: Predicciones del primer modelo en distintas epochs del entrenamiento.

En la Figura 34, se visualizan dos imágenes de entrada junto a las predicciones realizadas con el modelo entrenado durante 10 *epochs* y sus correspondientes imágenes originales.

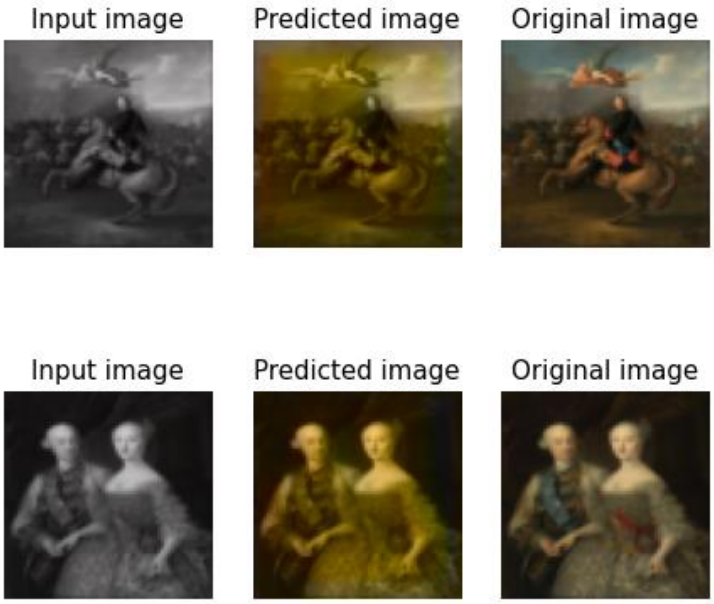

Figura 34: Ejemplo de dos imágenes de entrada junto a las predicciones realizadas con el primer modelo entrenado durante 10 epochs y sus correspondientes imágenes originales.

7.1.2. Segundo modelo

El código referente a esta arquitectura se encuentra en el Anexo 2.

El entrenamiento del segundo modelo en cada una de las *epochs* tiene una duración aproximada de 620 segundos.

Después de entrenar el modelo durante 50 *epochs*, se obtiene que la red que mejor predice el resultado se corresponde con la de la *epoch* 46. Sin embargo, tras visualizar las gráficas de precisión (Figura 35) y de pérdida (Figura 36), se puede observar que existe un problema de sobreentrenamiento. Es decir, aunque se siga entrenando el modelo durante más y más *epochs*, los valores de las distintas variables apenas cambian.

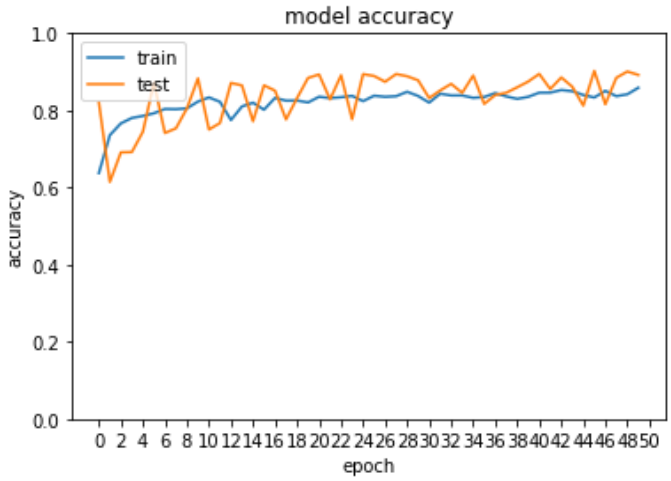

Figura 35: Gráfica de precisión del segundo modelo durante 50 epochs.

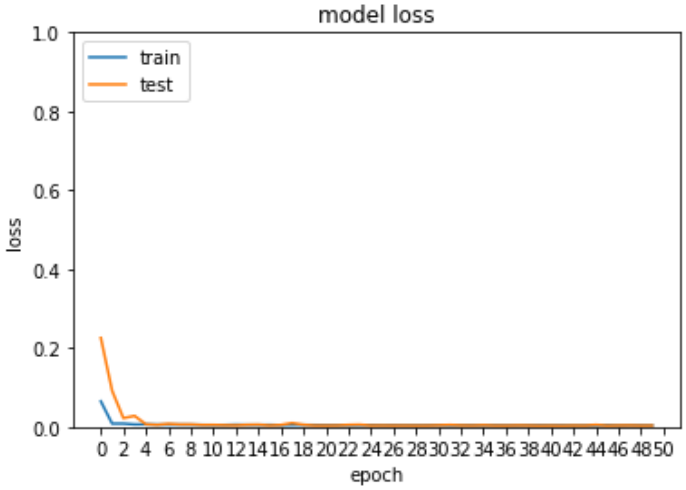

Figura 36: Gráfica de pérdida del segundo modelo durante 50 epochs.

Por lo tanto, estas gráficas junto a la visualización de las imágenes que se obtienen de las predicciones en distintas *epochs* hacen que se deduzca que uno de los modelos que mejor resultados genera es el entrenado durante 35 *epochs*.

Por otra parte, en la Figura 37Figura 33 se visualiza de qué forma va aprendiendo el modelo, mostrando las predicciones que realiza en distintas *epochs*.

Figura 37: Predicciones del segundo modelo en distintas epochs del entrenamiento.

Un ejemplo del resultado de las predicciones de este modelo se observa en la Figura 38.

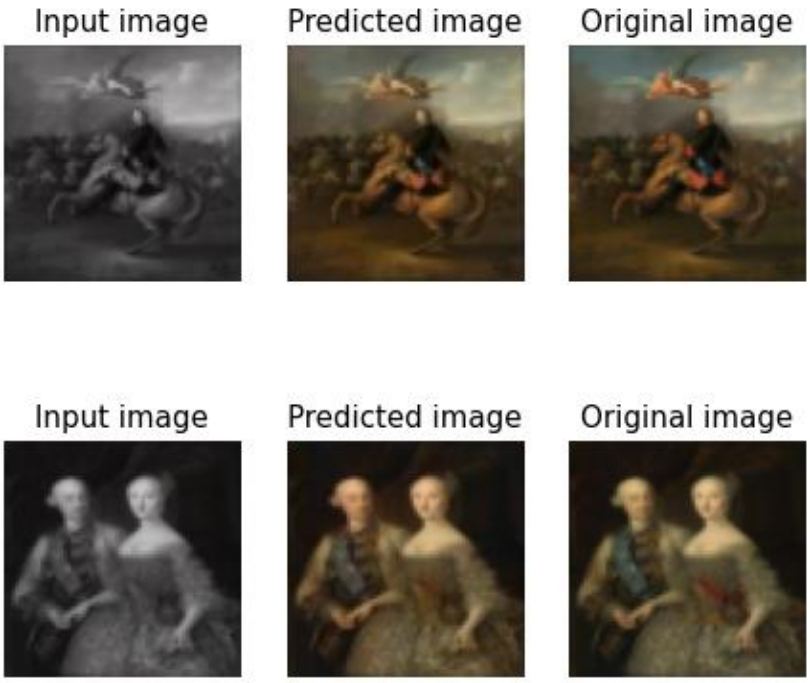

Figura 38: Ejemplo de dos imágenes de entrada junto a las predicciones realizadas con el segundo modelo entrenado durante 35 epochs y sus correspondientes imágenes originales.

7.1.3. Tercer modelo

El código referente a esta arquitectura se encuentra en el Anexo 3.

El entrenamiento del tercer modelo en cada una de las *epochs* tiene una duración aproximada de 513 segundos.

Como se puede observar en la Figura 39 y en la Figura 40, este modelo parece seguir un proceso de aprendizaje normal. Las gráficas representan valores que aumentan en el caso de la precisión y que disminuyen en el caso de la pérdida. Además, tanto el valor de *accuracy* como el de *loss* para el conjunto de entrenamiento se considera lo suficientemente bueno en la *epoch* 10.

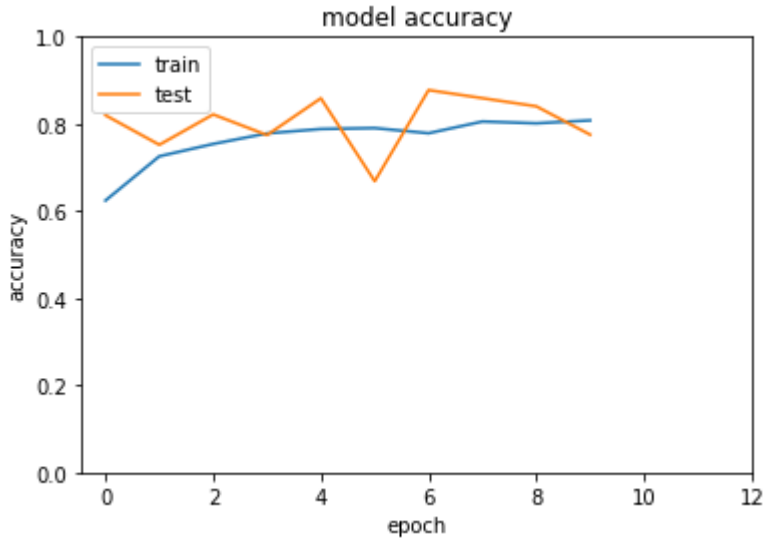

Figura 39: Gráfica de precisión del tercer modelo durante 10 epochs.

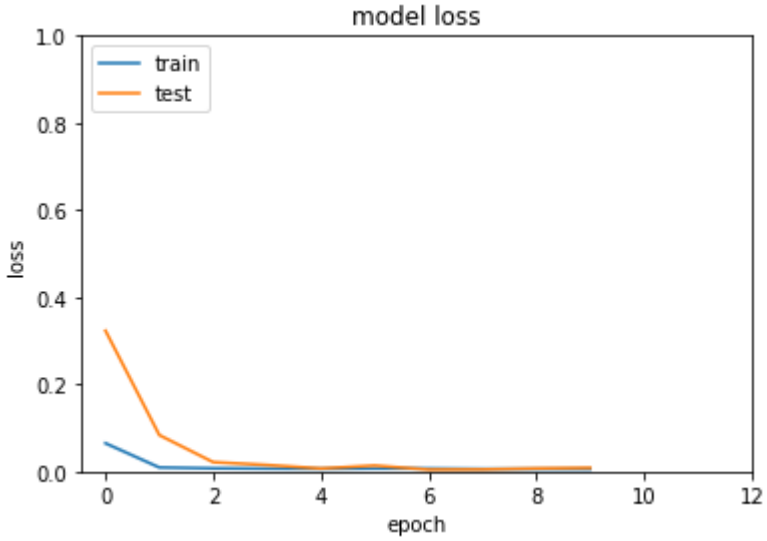

Figura 40: Gráfica de pérdida del tercer modelo durante 10 epochs.

Sin embargo, al observar las predicciones realizadas (Figura 41), se percibe gran homogeneidad en la forma de colorear la imagen. Es decir, las imágenes generadas a partir de las predicciones del modelo se colorean de colores neutros como el marrón, producto de la media calculada con las capas de *Average Pooling*. Esto hace que, visualmente, los resultados no sean los esperados.

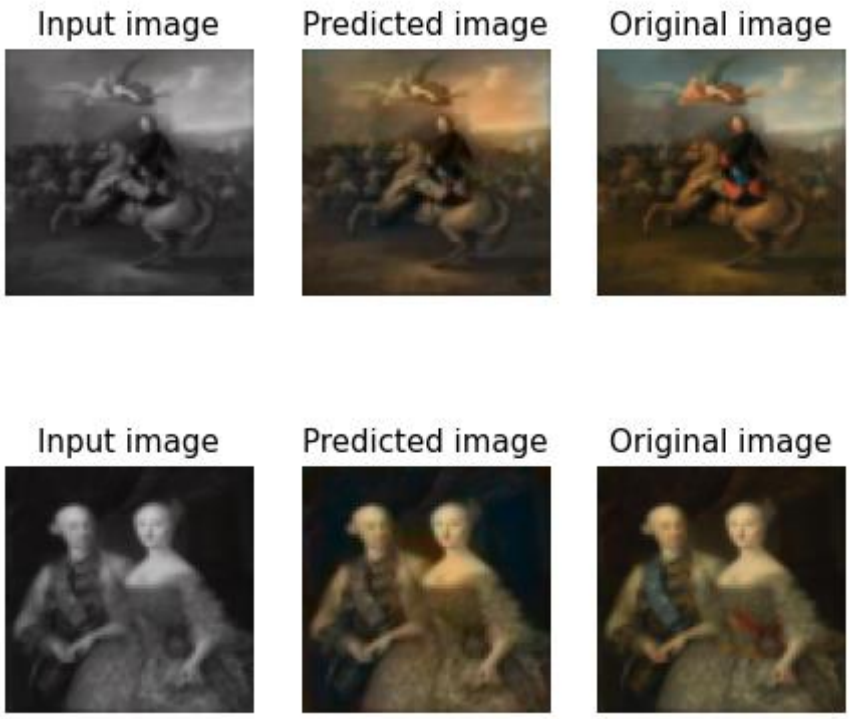

Figura 41: Ejemplo de dos imágenes de entrada junto a las predicciones realizadas con el tercer modelo entrenado durante 10 epochs y sus correspondientes imágenes originales.

Por otra parte, en la Figura 42 se visualiza de qué forma va aprendiendo el modelo, mostrando las predicciones que realiza en distintas *epochs*.

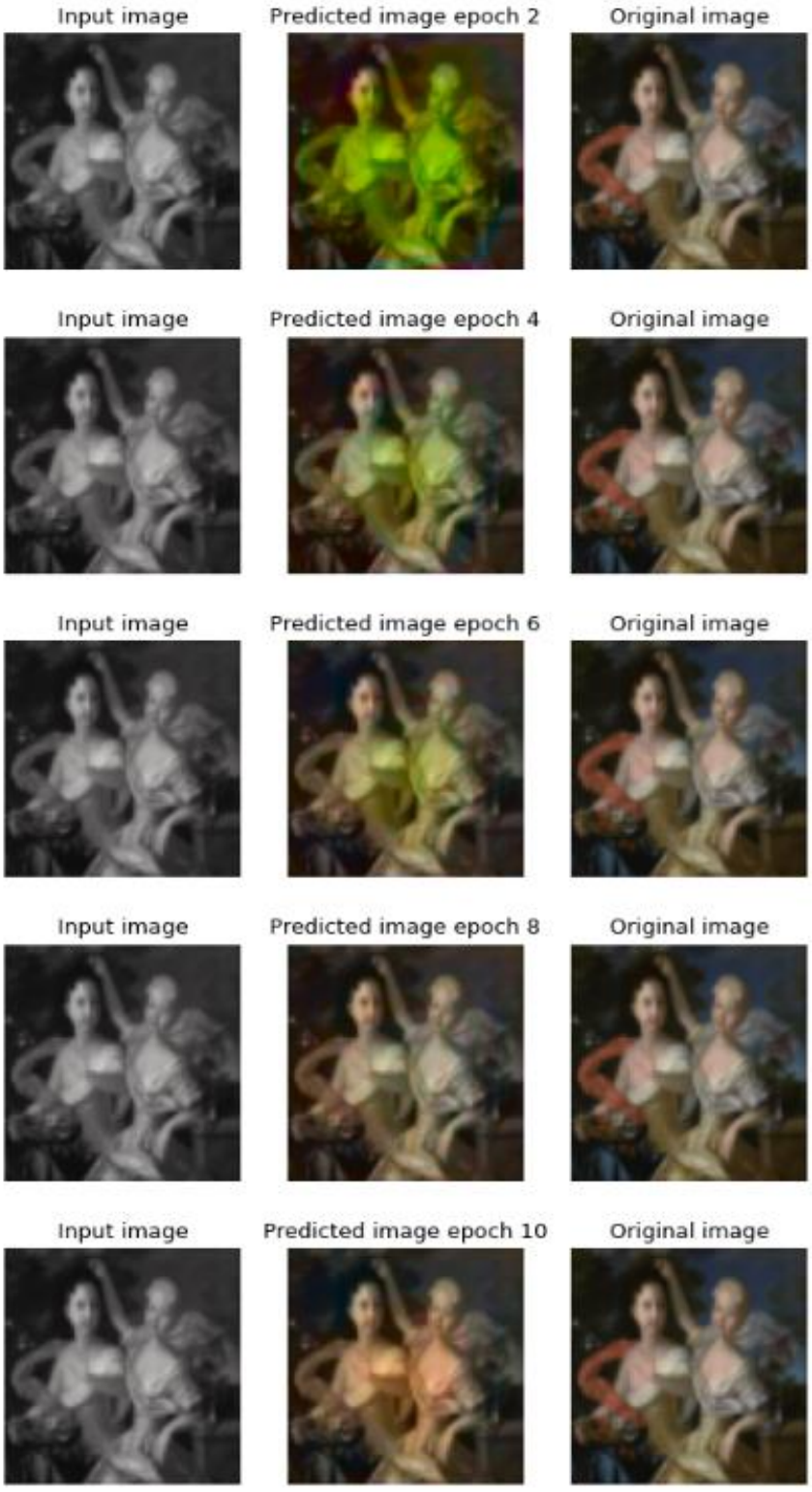

Figura 42: Predicciones del tercer modelo en distintas epochs del entrenamiento.

7.1.4. Cuarto modelo

El código referente a esta arquitectura se encuentra en el Anexo 4.

El entrenamiento del primer modelo en cada una de las *epochs* tiene una duración aproximada de 522 segundos.

La separación por lotes del conjunto de entrenamiento reduce el tiempo de entrenamiento porque los pesos se ajustan en cada uno de los lotes. Sin embargo, los resultados obtenidos empeoran respecto a los modelos anteriores.

Tanto en la gráfica de precisión (Figura 43) como en la de pérdida (Figura 44) se puede observar que los valores de las variables podrían ser mejores.

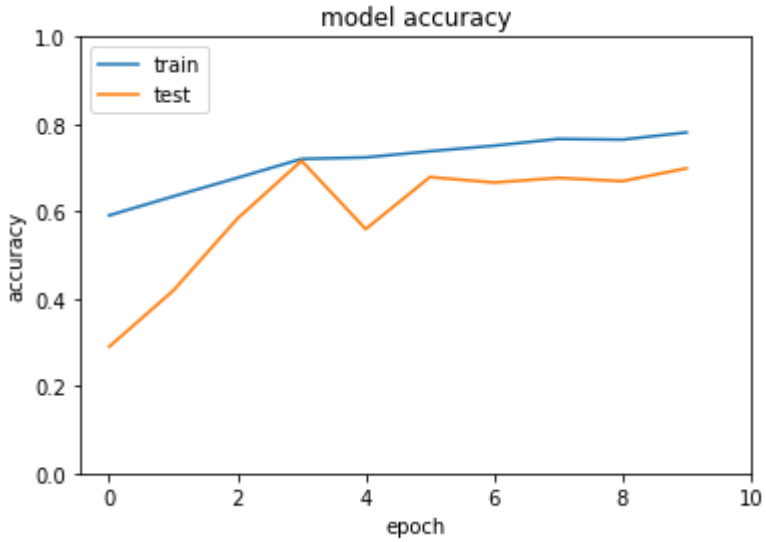

Figura 43: Gráfica de precisión del cuarto modelo durante 10 epochs.

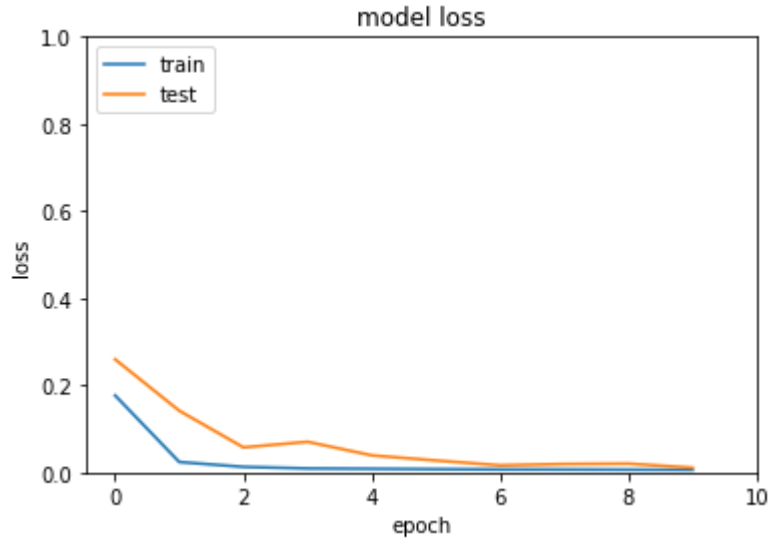

Figura 44: Gráfica de pérdida del cuarto modelo durante 10 epochs.

Por otra parte, en la Figura 45 se visualiza de qué forma va aprendiendo el modelo, mostrando las predicciones que realiza en distintas *epochs*.

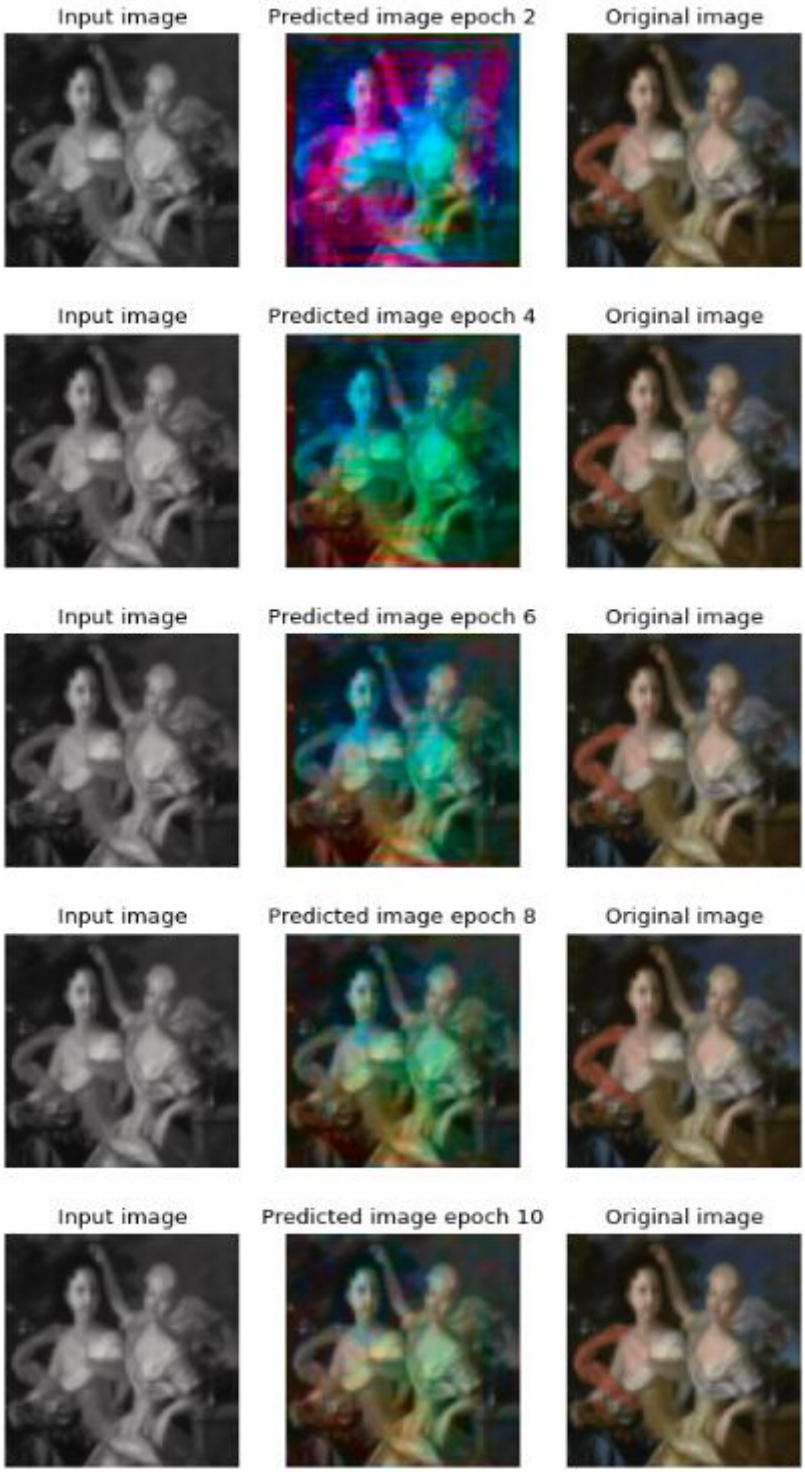

Figura 45: Predicciones del cuarto modelo en distintas epochs del entrenamiento.

Asimismo, en la Figura 46 se muestra que los colores predichos con esta red no se corresponden, en gran medida, con los colores de las imágenes reales de las que provienen.

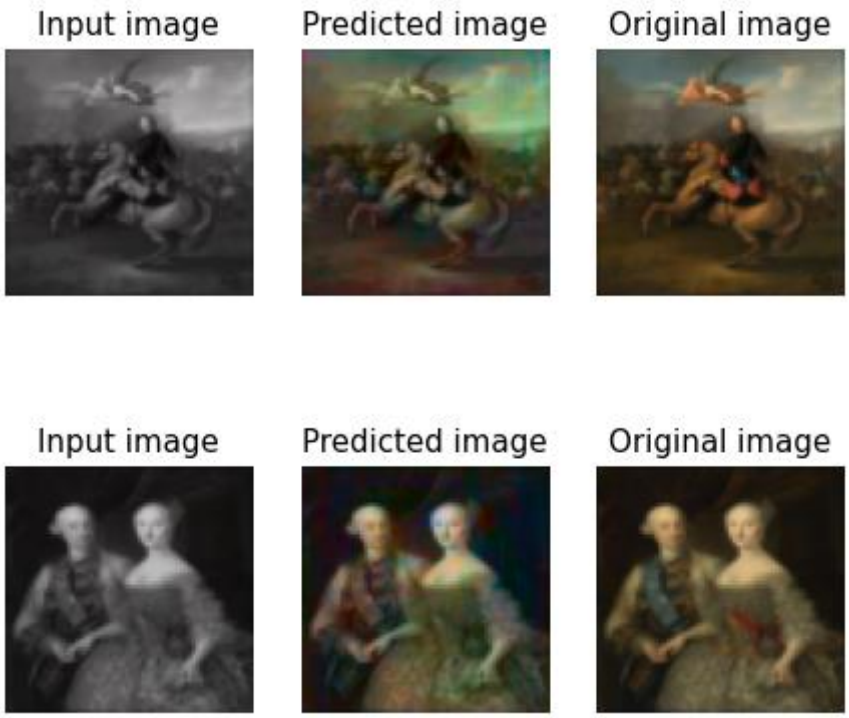

Figura 46: Ejemplo de dos imágenes de entrada junto a las predicciones realizadas con el cuarto modelo entrenado durante 10 epochs y sus correspondientes imágenes originales.

7.1.5. Quinto modelo

El código referente a esta arquitectura se encuentra en el Anexo 5.

El entrenamiento del quinto modelo en cada una de las *epochs* tiene una duración aproximada de 50 segundos.

Al verse reducido el conjunto de entrenamiento, la reducción del tiempo de entrenamiento es evidente.

Por un lado, en la Figura 47 y en la Figura 48 se visualiza que tanto los valores que toma *accuracy* como los que toma *loss* tienen un comportamiento normal. Es decir, en el caso de la *accuracy*, aumenta según el número de *epochs* y, en el caso de la *loss*, disminuye hasta un límite.

Por el contrario, los valores de precisión y pérdida en el conjunto de validación son extremadamente malos en comparación a otras arquitecturas. Esto es así porque, al haber entrenado el modelo con tan pocos datos, este no es capaz de realizar buenas predicciones cuando no conoce las imágenes de entrada.

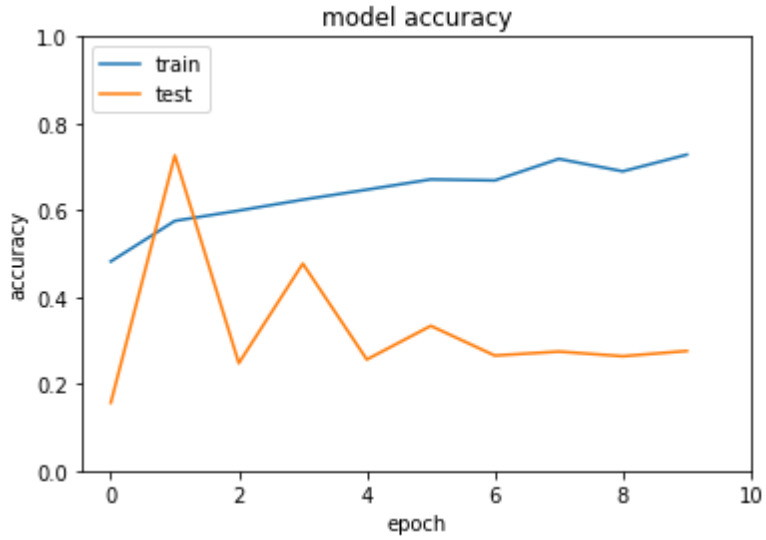

Figura 47: Gráfica de precisión del quinto modelo durante 10 epochs.

Tras observar la gráfica de precisión se puede deducir que la red puede seguir entrenándose, ya que la tendencia de la precisión en el conjunto de datos de entrenamiento en las últimas *epochs* no se ha estancado y sigue aumentando.

Asimismo, se puede visualizar que no se ha dado el problema de *overfitting*

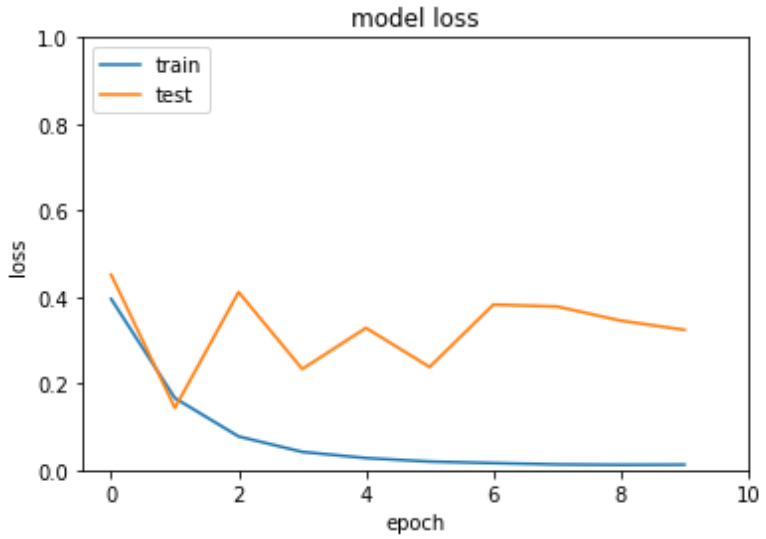

Figura 48: Gráfica de pérdida del quinto modelo durante 10 epochs.

Es obvio, por lo tanto, que los colores predichos de las imágenes que predice esta red no se corresponden para nada con los originales, como se ve en la Figura 49.

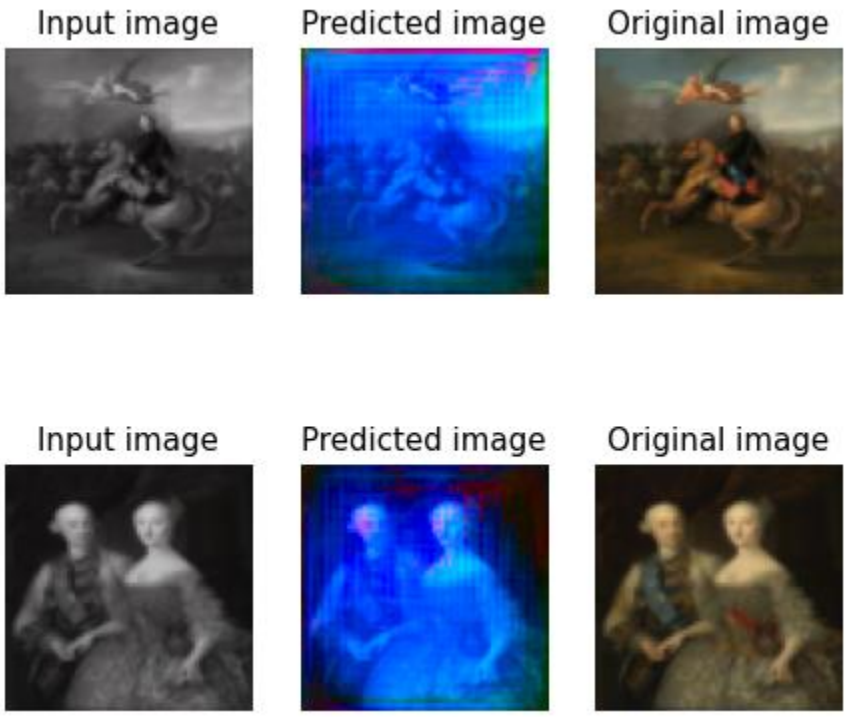

Figura 49: Ejemplo de dos imágenes de entrada junto a las predicciones realizadas con el quinto modelo entrenado durante 10 epochs y sus correspondientes imágenes originales.

Por último, en la Figura 50 se visualiza de qué forma va aprendiendo el modelo, mostrando las predicciones que realiza en distintas *epochs*.

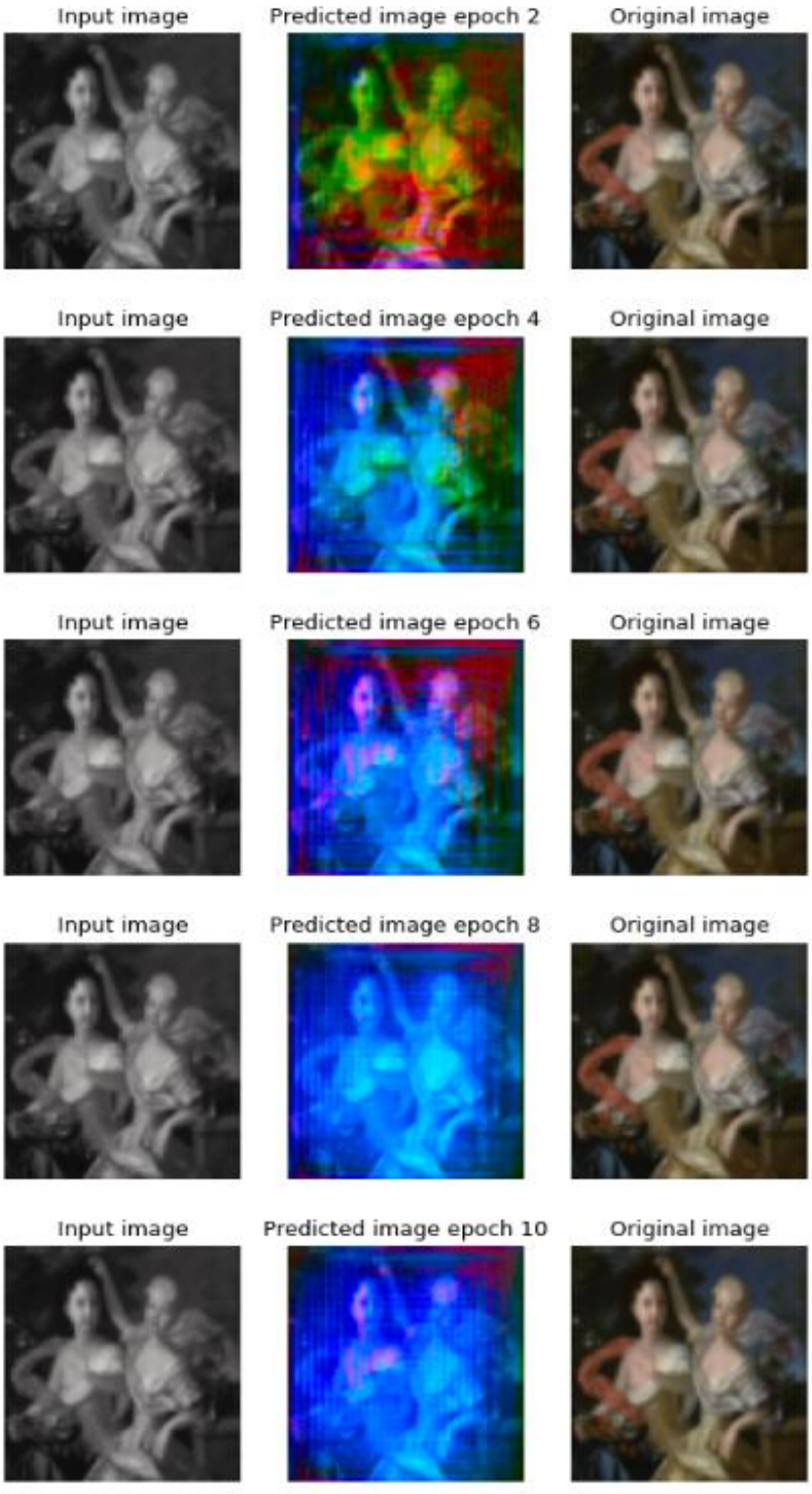

Figura 50: Predicciones del quinto modelo en distintas epochs del entrenamiento.

7.1.6. Sexto modelo (10 *epochs*)

El código referente a esta arquitectura se encuentra en el Anexo 6.

El entrenamiento del sexto modelo en cada una de las *epochs* tiene una duración aproximada de 91 segundos.

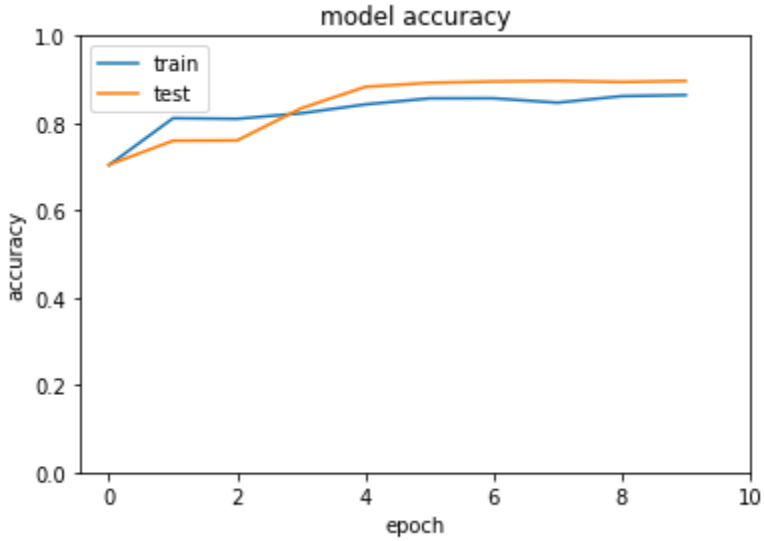

Figura 51: Gráfica de precisión del sexto modelo durante 10 epochs.

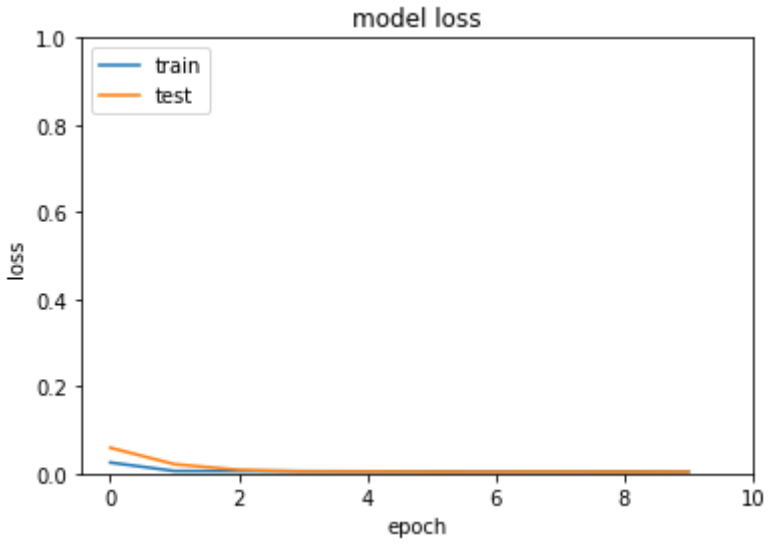

Figura 52: Gráfica de pérdida del sexto modelo durante 10 epochs.

Como se puede comprobar en ambas gráficas (Figura 51 y Figura 52), los valores que toman la precisión y la pérdida con ambos conjuntos de datos son bastante buenos y siguen una tendencia normal.

Por otra parte, en la Figura 53 se visualiza de qué forma va aprendiendo el modelo, mostrando las predicciones que realiza en distintas *epochs*.

Figura 53: Predicciones del sexto modelo en distintas epochs del entrenamiento.

Viendo los resultados que se obtienen (Figura 54), se considera que el modelo aún no ha aprendido lo suficiente, ya que hay zonas que no tienen color.

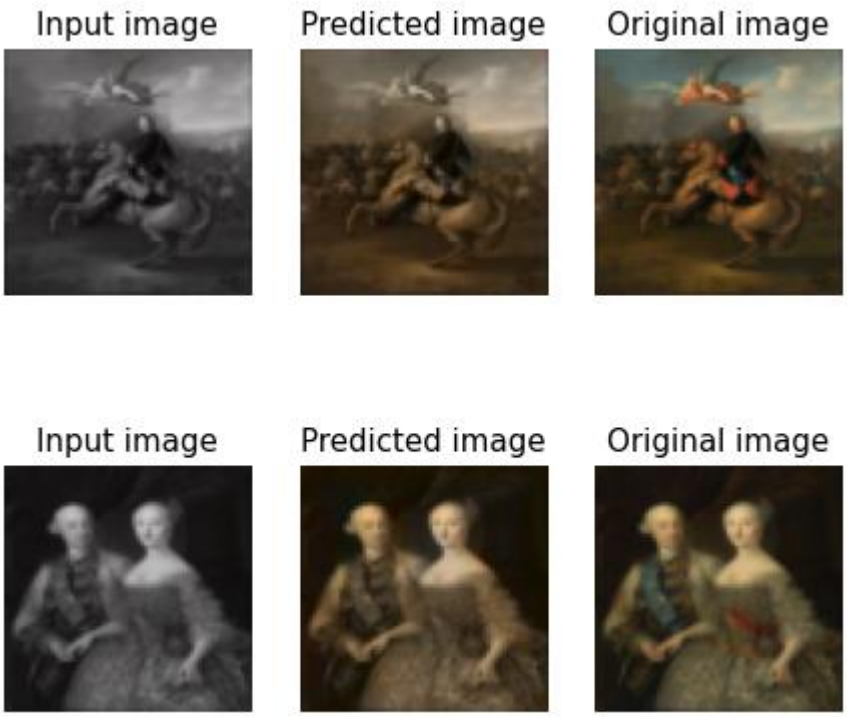

Figura 54: Ejemplo de dos imágenes de entrada junto a las predicciones realizadas con el sexto modelo entrenado durante 10 epochs y sus correspondientes imágenes originales.

7.1.7. Sexto modelo (30 *epochs*)

Al percibir que los resultados para las variables de precisión y pérdida eran muy buenos en esta arquitectura, se decide entrenarla durante 20 *epochs* más. Sin embargo, tras este entrenamiento, en la Figura 55 y en la Figura 56 se observa que la red sufre un problema de *overfitting*, por lo tanto, aunque la red siga siendo entrenada, los resultados que se obtienen no van a mejorar.

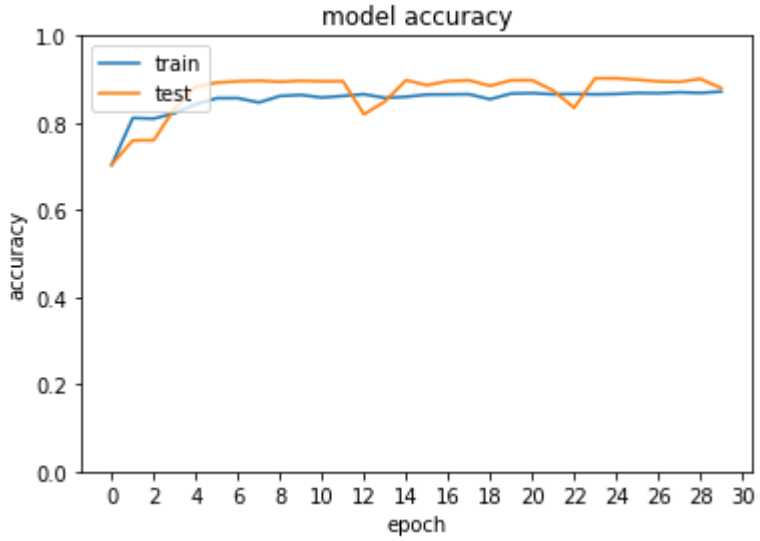

Figura 55: Gráfica de precisión del sexto modelo durante 30 epochs.

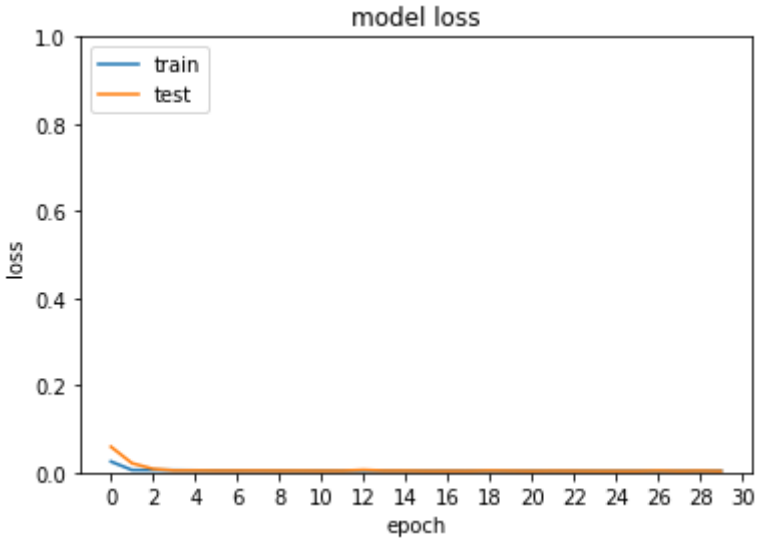

Figura 56: Gráfica de pérdida del sexto modelo durante 30 epochs.

Por otro lado, visualmente los resultados (Figura 57) son mejores que los obtenidos con la red entrenada durante 10 *epochs*. Sin embargo, siguen sin ser lo suficientemente buenos comparándolos con las imágenes originales y teniendo en cuenta que el proceso de entrenamiento ha durado 30 *epochs*.

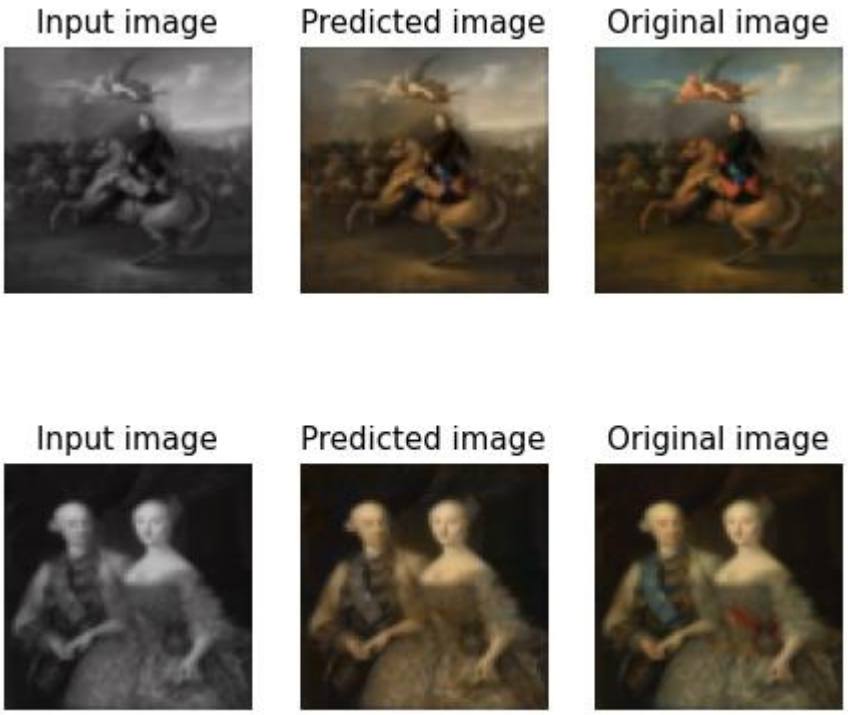

Figura 57: Ejemplo de dos imágenes de entrada junto a las predicciones realizadas con el sexto modelo entrenado durante 30 epochs y sus correspondientes imágenes originales.

7.1.8. Comparación

En este problema concreto es complicado evaluar los resultados de cada arquitectura, ya que no solo hay que basarse en obtener valores favorables en las variables de pérdida y precisión de los conjuntos de datos de entrenamiento y validación. Es decir, es imprescindible que el usuario observe las imágenes resultantes y analice qué modelo genera las que se corresponden más con las originales, aunque esta tarea sea totalmente subjetiva.

Por eso, tras estudiar los resultados de todas las arquitecturas, tanto la precisión y la pérdida (Tabla 2) como las imágenes predichas, se decide que el modelo que mejor actúa es el segundo.

	accuracy	***val_accuracy***	***loss***	***val_loss***
Primer modelo	0.8009	0.8855	0.0782	0.0986
Segundo modelo	0.8230	0.8829	0.00506	0.0048
Tercer modelo	0.8081	0.7754	0.0053	0.0083
Cuarto modelo	0.7810	0.6988	0.0059	0.0108
Quinto modelo	0.7281	0.2759	0.0123	0.3241
Sexto modelo 10 *epochs*	0.8642	0.8967	0.0042	0.0040
Sexto modelo 30 *epochs*	0.8719	0.8800	0.0037	0.0031

Tabla 2: Comparación de los valores de precisión y pérdida en los conjuntos de entrenamiento y validación de todas las arquitecturas.

Asimismo, se muestra la comparación de todas las variables de cada una de las arquitecturas en la Figura 58.

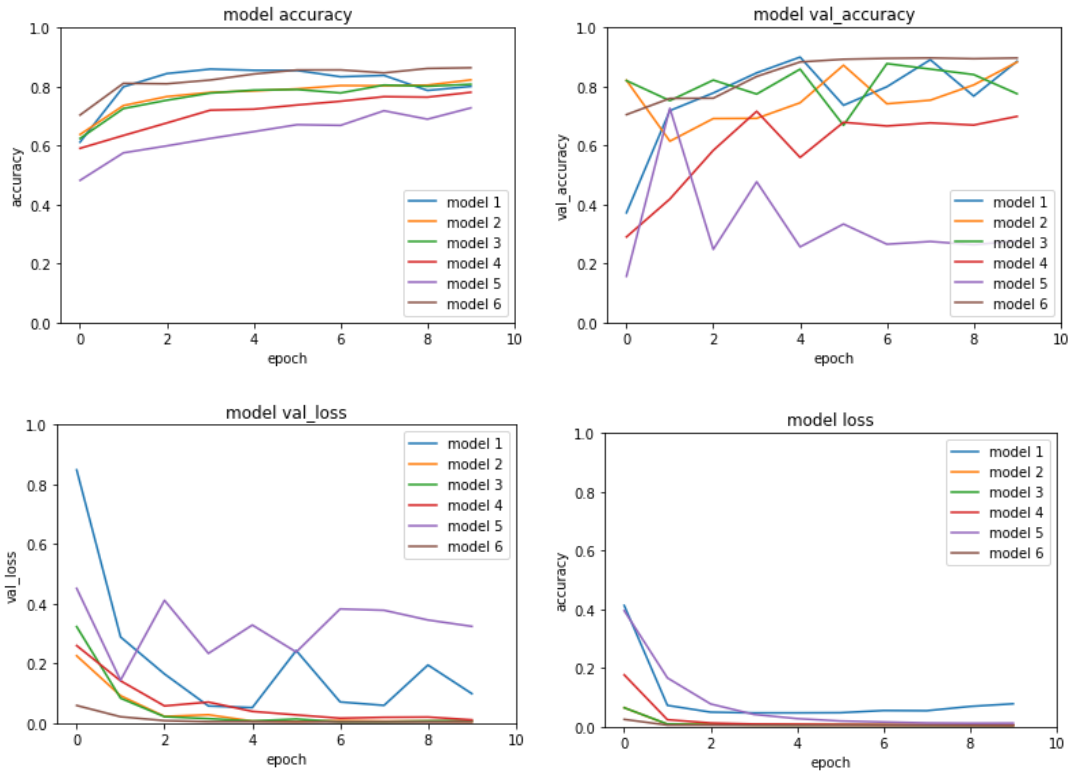

Figura 58: Gráficas de precisión y pérdida de todas las arquitecturas entrenadas durante 10 epochs.

8. CONCLUSIONES

Como ya se comentó, al inicio del proyecto se marcaron diferentes objetivos que se deseaban alcanzar.

El objetivo principal era elaborar una herramienta que diese la posibilidad de dar color a fotografías que estaban en blanco y negro. Sin embargo, para conseguir esto era necesario establecer otros objetivos.

Uno de ellos fue analizar lo que ya existía. Dicho de otra forma, se debía indagar tanto en las arquitecturas de redes neuronales actuales como en los estudios que resolvían el problema del coloreado automático utilizando técnicas de aprendizaje automático.

Otro de los objetivos que se quería conseguir era construir un modelo y realizar una optimización de hiperparámetros de este para obtener los valores con los que mejores resultados se obtenían.

Asimismo, se consideraba que la elaboración de distintos modelos y la comparación entre ellos aportaría gran valor al proyecto.

Por último, algo que facilitaría la interacción del usuario sería desarrollar una interfaz sencilla que incorporase el funcionamiento del coloreado de imágenes.

En primer lugar, se puede afirmar que se ha conseguido implementar un modelo que genera imágenes a color que se corresponden lo suficiente con las fotografías reales de las que proceden.

Para construir la arquitectura, fue imprescindible realizar una vasta investigación del estado de la cuestión, por lo que se alcanzó el segundo objetivo.

Conseguir obtener buenos resultados a partir del modelo fue posible gracias al ajuste de hiperparámetros de la red realizado ya que, antes de aplicar la optimización, las imágenes generadas no se acercaban tanto a la realidad. Este hecho refleja que también se ha logrado este propósito.

Finalmente, se implementan distintas arquitecturas cambiando los parámetros de sus capas o la forma en la que se realiza el proceso de entrenamiento, y

se desarrolla una interfaz de usuario muy simple. Asimismo, se comparan aspectos como las salidas que producen o el tiempo que dura su entrenamiento.

Con todo lo comentado se puede concluir que se han alcanzado todos los objetivos propuestos adecuadamente. Aun así, el trabajo realizado se ha visto limitado por los recursos escasos con los que se contaba, ya que solo se ha utilizado el tiempo de proceso de *Colab*. Por lo tanto, con mayores recursos se habría podido usar un conjunto de imágenes de mayor calidad, realizar un mejor ajuste de hiperparámetros para cada uno de los modelos y probar con otros tipos de redes neuronales.

8.1. Trabajo futuro

En primera instancia, se podría utilizar un conjunto de datos de entrenamiento mucho más extenso y variado que el que se ha utilizado. En general, el *dataset* que se ha empleado contiene imágenes de colores mayoritariamente neutros como el marrón. Por lo tanto, es relativamente sencillo que los colores que prediga la red se asemejen a la realidad.

Sin embargo, si la red hubiese sido entrenada con imágenes que contienen colores vivos, sería posible obtener mejores resultados con distintos tipos de fotografías, independientemente de que fuesen de colores neutros o no.

Otra mejora a la hora de entrenar el modelo sería utilizar una red preentrenada como las usadas en algunos de los estudios que se han analizado durante el desarrollo del proyecto.

Por último, un avance muy atractivo para este proyecto, pero de mayor complejidad sería resolver el problema de coloreado de vídeos.

REFERENCIAS BIBLIOGRÁFICAS

[1] S. Camilo and C. Aguirre, “Very Deep Convolutional Neural Networks for Image Colorization”.

[2] “color | Definición | Diccionario de la lengua española | RAE - ASALE.” https://dle.rae.es/color (accessed Mar. 12, 2022).

[3] “Síntesis aditiva de color - Wikipedia, la enciclopedia libre.” https://es.wikipedia.org/wiki/S%C3%ADntesis_aditiva_de_color (accessed Mar. 15, 2022).

[4] “Síntesis sustractiva de color - Wikipedia, la enciclopedia libre.” https://es.wikipedia.org/wiki/S%C3%ADntesis_sustractiva_de_color#Colores_primarios_y_secundarios (accessed Mar. 15, 2022).

[5] “Síntesis Aditiva y Sustractiva, dos formas de ver el color.” https://laprestampa.com/el-proceso-grafico/diseno/sintesis-aditiva-y-sustractiva/ (accessed Mar. 15, 2022).

[6] “Modelo de colores - Wikipedia, la enciclopedia libre.” https://es.wikipedia.org/wiki/Modelo_de_colores#Lista_de_modelos_de_colores (accessed Apr. 20, 2022).

[7] “Espacio de color - Wikipedia, la enciclopedia libre.” https://es.wikipedia.org/wiki/Espacio_de_color#Lista_parcial_de_espacios_de_color (accessed Apr. 20, 2022).

[8] “Documento sin título.” http://pfc.upnfm.edu.hn/cursos/gimp/unidad02/txt_0006.htm (accessed Apr. 20, 2022).

[9] “Los colores en Informática: modelos RGB y HSV 1 Los colores en informática”, Accessed: Apr. 20, 2022. [Online]. Available: www.gimp.org

[10] “Item 1009/362 | Repositorio INAOE.” https://inaoe.repositorioinstitucional.mx/jspui/handle/1009/362 (accessed May 19, 2022).

[11] “RGB y CMYK: Qué son y cuándo usar cada modo de color - Imborrable.” https://imborrable.com/blog/rgb-y-cmyk/ (accessed Apr. 20, 2022).

[12] “Colorizing black & white images with U-Net and conditional GAN—A Tutorial | Towards Data Science.” https://towardsdatascience.com/colorizing-black-white-images-with-u-net-and-conditional-gan-a-tutorial-81b2df111cd8 (accessed May 23, 2022).

[13] “RGB - Wikipedia, la enciclopedia libre.” https://es.wikipedia.org/wiki/RGB (accessed Apr. 20, 2022).

[14] “Espacio de color Lab - Wikipedia, la enciclopedia libre.” https://es.wikipedia.org/wiki/Espacio_de_color_Lab#cite_note-versus-1 (accessed Apr. 20, 2022).

[15] “¿Conoce el Espacio de Color CIE L*A*B*? - AQ instrumentsAQinstruments.” https://www.aquateknica.com/conoce-el-espacio-de-color-cie-lab/ (accessed Apr. 20, 2022).

[16] “Truco express: esto es lo que no debes olvidar acerca de los espacios de color RGB, CMYK y Color Lab.” https://www.xatakafoto.com/trucos-y-consejos/truco-express-esto-es-lo-que-no-debes-olvidar-acerca-de-los-espacios-de-color-rgb-cmyk-y-color-lab (accessed Apr. 20, 2022).

[17] “Modo de color Lab.” https://sobrecolores.blogspot.com/2010/03/modo-de-color-lab.html (accessed Apr. 20, 2022).

[18] J. Jesús Romero Carlos Dafonte Ángel Gómez Fernando Jorge Penousal and F. Alfredo Brañas, “Inteligencia Artificial y Computación Avanzada”.

[19] J. F. Avila-Tomás, M. A. Mayer-Pujadas, and V. J. Quesada-Varela, “La inteligencia artificial y sus aplicaciones en medicina I: introducción antecedentes a la IA y robótica,” *Atención Primaria*,

vol. 52, no. 10, pp. 778–784, Dec. 2020, doi: 10.1016/J.APRIM.2020.04.013.

[20] A. Kaplan, M. Haenlein, and A. Kaplan, *Siri, Siri in my Hand, who's the Fairest in the Land? On the Interpretations, Illustrations and Implications of Artificial Intelligence*. 2018. Accessed: Apr. 26, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0007681318301393

[21] "Aprendizaje automático en acción: Un libro para el lego, guía paso a paso ... - Alan T. Norman - Google Libros." https://books.google.es/books?hl=es&lr=&id=XX29DwAAQBAJ&oi=fnd&pg=PT3&dq=tipos+aprendizaje+automatico&ots=rGknuMO1yI&sig=xg2oISdMTeyGonH48kNEC-MyXUc#v=onepage&q=tipos%20aprendizaje%20automatico&f=false (accessed Apr. 25, 2022).

[22] "'Machine Learning': definición, tipos y aplicaciones prácticas - Iberdrola." https://www.iberdrola.com/innovacion/machine-learning-aprendizaje-automatico (accessed Apr. 25, 2022).

[23] "▷¿Qué es el Machine Learning? | Aprendizaje Automático | UCM." https://www.masterdatascienceucm.com/que-es-machine-learning/ (accessed May 02, 2022).

[24] "1 What is deep learning? - Deep Learning with Python, Second Edition." https://livebook.manning.com/book/deep-learning-with-python-second-edition/chapter-1/v-4/49 (accessed May 31, 2022).

[25] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets *".

[26] Jordi. Casas Roma, Anna. Bosch Rué, and Toni. Lozano Bagén, "Deep learning : principios y fundamentos," p. 260, 2019.

[27] J. P. Cerrone, "Generación de niveles de juego de plataformas utilizando redes neuronales LSTM." Accessed: Jun. 13, 2022. [Online]. Available:

https://www.ridaa.unicen.edu.ar/xmlui/bitstream/handle/123456789/2358/Trabajo%20Final%20-%20Juan%20Pedro%20Cerrone.pdf?sequence=1&isAllowed=y

[28] "¿Qué es una Red Neuronal? Parte 1 : La Neurona | DotCSV - YouTube." https://www.youtube.com/watch?v=MRIv2IwFTPg&ab_channel=DotCSV (accessed Jun. 13, 2022).

[29] "Deep Learning – Introducción práctica con Keras - Jordi TORRES.AI." https://torres.ai/deep-learning-inteligencia-artificial-keras/#3_REDES_NEURONALES_DENSAMENTE_CONECTADAS (accessed Jun. 13, 2022).

[30] "Tuning the Hyperparameters and Layers of Neural Network Deep Learning." https://www.analyticsvidhya.com/blog/2021/05/tuning-the-hyperparameters-and-layers-of-neural-network-deep-learning/ (accessed Jun. 26, 2022).

[31] "Conceptos básicos sobre redes neuronales." https://vincentblog.xyz/posts/conceptos-basicos-sobre-redes-neuronales (accessed Jun. 26, 2022).

[32] R. A. Vasco Carofilis, "Optimización Automática de Hiperparámetros en Modelos de Aprendizaje Automático mediante PBIL," Universidad de Sevilla, Sevilla, 2019. Accessed: Jun. 28, 2022. [Online]. Available: http://darkmatter.ciemat.es/documents/585242/809389/AndresVasco-TFM.pdf/b53043be-2860-4dbc-b861-4aabf573dd95

[33] A. González Muñiz, "Aplicaciones de técnicas de inteligencia artificial basadas en aprendizaje profundo (deep learning) al análisis y mejora de la eficiencia de procesos industriales," Universidad de Oviedo, Oviedo, 2018. Accessed: Jun. 26, 2022. [Online]. Available: https://www.researchgate.net/profile/Ana-Gonzalez-Muniz/publication/334328255_Aplicaciones_de_tecnicas_de_inteligencia_artificial_basadas_en_aprendizaje_profundo_deep_lear

ning_al_analisis_y_mejora_de_la_eficiencia_de_procesos_industriales/links/5d24c78ca6fdcc2462d04f8a/Aplicaciones-de-tecnicas-de-inteligencia-artificial-basadas-en-aprendizaje-profundo-deep-learning-al-analisis-y-mejora-de-la-eficiencia-de-procesos-industriales.pdf

[34] "Convolutional Neural Networks: La Teoría explicada en Español | Aprende Machine Learning." https://www.aprendemachinelearning.com/como-funcionan-las-convolutional-neural-networks-vision-por-ordenador/ (accessed Jun. 29, 2022).

[35] V. Dumoulin, F. Visin, and G. E. P. Box, "A guide to convolution arithmetic for deep learning," 2018, Accessed: Jun. 26, 2022. [Online]. Available: http://ethanschoonover.com/solarized

[36] M. Yani, B. Irawan, and C. Setiningsih, "Application of Transfer Learning Using Convolutional Neural Network Method for Early Detection of Terry's Nail," *Journal of Physics: Conference Series*, vol. 1201, no. 1, May 2019, doi: 10.1088/1742-6596/1201/1/012052.

[37] "Leggi Online. Titolo: "Deep learning : fundamentos, teoría y aplicación. Categoria: E-book - Torrossa." https://www.torrossa.com/it/catalog/preview/4947314 (accessed Jun. 26, 2022).

[38] L. R. Calcagni, "Redes Generativas Antagónicas y sus aplicaciones," La Plata, 2020. Accessed: Jun. 30, 2022. [Online]. Available: http://sedici.unlp.edu.ar/bitstream/handle/10915/101507/Documento_completo.pdf-PDFA.pdf?sequence=1&isAllowed=y

[39] A. Levin, D. Lischinski, and Y. Weiss, "Colorization using Optimization", Accessed: Jun. 14, 2022. [Online]. Available: http://www.museum.tv/archives/etv/index.html.

[40] A. Hertzmann, C. E. Jacobs, N. Oliver, B. Curless, and D. H. Salesin, "Image analogies," *Proceedings of the 28th Annual*

Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 2001, pp. 327–340, 2001, doi: 10.1145/383259.383295.

[41] R. Zhang, J.-Y. Zhu, A. S. Lin, T. Yu, and A. A. Efros, “Real-Time User-Guided Image Colorization with Learned Deep Priors”, doi: 10.1145/3072959.3073703.

[42] “Real-Time User-Guided Image Colorization with Learned Deep Priors - YouTube.” https://www.youtube.com/watch?v=eL5ilZgM89Q&t=234s&ab_channel=RichardZhang (accessed Jun. 03, 2022).

[43] “U-NET : todo lo que tienes que saber sobre la red neuronal de Computer Vision.” https://datascientest.com/es/u-net-lo-que-tienes-que-saber (accessed Jun. 04, 2022).

[44] “Image Colorization with U-Net and GAN Tutorial.ipynb - Colaboratory.” https://colab.research.google.com/github/moein-shariatnia/Deep-Learning/blob/main/Image%20Colorization%20Tutorial/Image%20Colorization%20with%20U-Net%20and%20GAN%20Tutorial.ipynb#scrollTo=J9D43UdIMFCl (accessed May 24, 2022).

[45] P. Warden and D. Situnayake, “TinyML Tensorflow Lite”.

[46] R. G. Duque, “Python PARA TODOS”, Accessed: Jul. 04, 2022. [Online]. Available: http://mundogeek.net/tutorial-python/

[47] “Te damos la bienvenida a Colaboratory - Colaboratory.” https://colab.research.google.com/?hl=es#scrollTo=-Rh3-Vt9Nev9 (accessed Jul. 04, 2022).

[48] “Kaggle: Your Machine Learning and Data Science Community.” https://www.kaggle.com/ (accessed Jun. 24, 2022).

[49] “Art Images: Drawing/Painting/Sculptures/Engravings | Kaggle.” https://www.kaggle.com/datasets/thedownhill/art-images-drawings-painting-sculpture-engraving (accessed Jun. 24, 2022).

[50] “Module: color — skimage v0.20.0.dev0 docs.” https://scikit-image.org/docs/dev/api/skimage.color.html (accessed Jun. 25, 2022).

[51] “Colorizing B&W Photos with Neural Networks.” https://blog.floydhub.com/colorizing-b-w-photos-with-neural-networks/ (accessed Jun. 30, 2022).

[52] “Image Colorization basic implementation with CNN | Kaggle.” https://www.kaggle.com/code/pythonerski/image-colorization-basic-implementation-with-cnn (accessed Jul. 01, 2022).

[53] “Variational AutoEncoder - Fernando Sancho Caparrini.” http://www.cs.us.es/~fsancho/?e=232 (accessed Jul. 01, 2022).

[54] “Weight Initialization and Activation Functions - Deep Learning Wizard.” https://www.deeplearningwizard.com/deep_learning/boosting_models_pytorch/weight_initialization_activation_functions/ (accessed Jun. 27, 2022).

[55] P. M. Khanolkar, C. C. McComb, and S. Basu, “Predicting elastic strain fields in defective microstructures using image colorization algorithms,” *Computational Materials Science*, vol. 186, p. 110068, Jan. 2021, doi: 10.1016/J.COMMATSCI.2020.110068.

[56] “What is the difference between ReLU and LeakyReLU? – Quick-Advisors.com.” https://quick-advisors.com/what-is-the-difference-between-relu-and-leakyrelu/ (accessed Jun. 27, 2022).

[57] “Autoencoder: Downsampling and Upsampling.” https://kharshit.github.io/blog/2019/02/15/autoencoder-downsampling-and-upsampling (accessed Jul. 01, 2022).

[58] “Hyperparameter Tuning with Grid Search and Random Search | by Idil Ismiguzel | Towards Data Science.” https://towardsdatascience.com/hyperparameter-tuning-with-grid-search-and-random-search-6e1b5e175144 (accessed Jun. 27, 2022).

[59] M. Agrawal and K. Sawhney, “Exploring Convolutional Neural Networks for Automatic Image Colorization.” Accessed: Jun. 29, 2022. [Online]. Available: https://web.stanford.edu/~kartiks2/cnnColorization.pdf

[60] “Todo lo que Necesitas Saber sobre el Descenso del Gradiente Aplicado a Redes Neuronales | by Jaime Durán | MetaDatos | Medium.” https://medium.com/metadatos/todo-lo-que-necesitas-saber-sobre-el-descenso-del-gradiente-aplicado-a-redes-neuronales-19bdbb706a78 (accessed Jun. 29, 2022).

[61] “Losses.” https://keras.io/api/losses/ (accessed Jun. 29, 2022).

[62] W. Y. Lee, S. M. Park, and K. B. Sim, “Optimal hyperparameter tuning of convolutional neural networks based on the parameter-setting-free harmony search algorithm,” *Optik (Stuttg)*, vol. 172, pp. 359–367, Nov. 2018, doi: 10.1016/J.IJLEO.2018.07.044.

ANEXOS

Anexo 1. Notebook referente al primer modelo. https://colab.research.google.com/drive/1McVV--73LZPTFuLz_6hnSesvgBo70iwD?usp=sharing

Anexo 2. Notebook referente al segundo modelo. https://colab.research.google.com/drive/1aPycJ-e2JvUBadfoWrrpm8QX68Y5qV3a?usp=sharing

Anexo 3. Notebook referente al tercer modelo. https://colab. https://colab.research.google.com/drive/1LNH2MTaGd6qF6dIvRMpRHovurV6BS6LM?usp=sharing

Anexo 4. Notebook referente al cuarto modelo. https://colab.research.google.com/drive/1gnao1mc_pGOY0XKyALwxpWpQ0oBx81Bg?usp=sharing

Anexo 5. Notebook referente al quinto modelo. https://colab.research.google.com/drive/1W4XM1dBudpyCfywCx9hFYZ0ylYysLyKy?usp=sharing

Anexo 6. Notebook referente al sexto modelo. https://colab.research.google.com/drive/1DRmESVibhzHvUITEGOlxhhgZGwF6OF4S?usp=sharing