Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN COMPUTACIÓN

Desarrollo de una herramienta basada en Deep Learning para asistir en el coloreado de imágenes antiguas

Estudante: David Maseda Neira
Dirección: Enrique Fernández Blanco

A Coruña, febreiro de 2020.

A mis padres, que me han ayudado a lo largo de estos años, mucho más allá de lo que creen.

Agradecimientos

Dedicado a toda mi familia, que me ha dado los medios y el la fuerza para llegar hasta aquí. A mis amigos, que en todo momento me han brindado su apoyo y ayuda, en especial a Eugenia y a Carmela.

Especial agradecimiento a mi tutor Enrique, que me ha ayudado enormemente en la realización de este proyecto.

Resumen

Dentro de la restauración y conservación del patrimonio cultural, un área que está comenzando a recibir mucha atención es el sector audiovisual. En concreto el coloreado de fotografías es un uno de los principales campos de trabajo ya que permite revivir tanto fotografías como vídeo de archivo. Esto permite a su vez ofrecer nuevas perspectivas y puntos de vista a documentos antiguos. Tradicionalmente, este coloreado de imágenes es llevado a cabo por artistas especializados mediante un proceso manual con herramientas de edición fotográfica. En este trabajo se propone una alternativa basada en las *Generative Adversarial Networks*, un nuevo modelo de *deep learning* con resultados muy prometedores. Las *Generative Adversarial Networks* son un campo de investigación muy activo, que evoluciona constantemente, y durante el desarrollo de este trabajo se explorará este área del aprendizaje automático, con la intención de mostrar el avance que supone respecto a las alternativas tradicionales, ya sea de cara al usuario final como para el alumno, que desarrollará competencias en un campo de reciente aparición. En paralelo a este modelo, se desarrollará una herramienta accesible al usuario para asistir al coloreado de imágenes de manera sencilla, utilizando el estándar de industria del desarrollo web.

Abstract

Within the restoration and conservation of cultural heritage, an area that is beginning to receive a lot of attention is the audiovisual sector. Specifically, the coloring of photographs is one of the main fields of work since it allows to revive both photographs and video files. This allows us to offer new perspectives and points of view to old documents. Traditionally, this coloring of images is carried out by specialized artists through a manual process with photo editing tools. This work proposes an alternative based on Generative Adversarial Networks, a new model of deep learning with very promising results. Generative Adversarial Networks are a very active field of research, which is constantly evolving, and during the development of this work this area of automatic learning will be explored, with the intention of showing the progress that it entails versus traditional alternatives, either for the end user as for the student, who will develop skills in a field of recent appearance. In parallel to this model, a tool accessible to the user will be developed to assist in the coloring of images in a simple way, using web development industry standards.

Palabras clave:

- Aprendizaje automático
- Redes Neuronales Artificiales
- Aprendizaje profundo
- Modelos generativos
- Visión artificial
- Tratamiento de imágenes

Keywords:

- Machine Learning
- Artificial Neural Network
- Deep Learning
- Generative Models
- Articial Vision
- Image Processing

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1 Descripción del trabajo

Dentro de la restauración y conservación del patrimonio cultural, un área que está comenzando a recibir mucha atención es el sector audiovisual. Esto es así tanto en archivos públicos como privados; ejemplo de estos últimos es la explotación comercial de imágenes documentales actualizadas a color por numerosos canales de televisión. En concreto, el coloreado de fotografías es uno de los principales campos de trabajo, ya que permite revivir tanto fotografías como cine y vídeo de archivo. Esto permite a su vez ofrecer nuevas perspectivas y puntos de vista a documentos antiguos. Tradicionalmente, este coloreado de imágenes es llevado a cabo por artistas especializados mediante un proceso meramente manual, con herramientas de edición como pueden ser los archiconocidos *Photoshop* o *GIMP*. Cabe señalar que también existen otras alternativas, como aproximaciones con métodos basados en etiquetas. En estas últimas, los artistas se liberan de una parte del trabajo, pero es necesario que refinen los resultados finales dentro del flujo de restauración personalmente.

En la última década ha surgido un conjunto de técnicas que, de contemplarse, podría convertirse en la solución para atajar los problemas que plantea el campo de la restauración de imágenes. Estas técnicas, conocidas bajo el nombre genérico de *deep learning*, presentan una serie de características que las hacen idóneas para la aproximación de herramientas de apoyo a la restauración.

Si bien las primeras aproximaciones al coloreado de imágenes mediante *deep learning* utilizaron redes neuronales convolucionales (*Convolutional Neural Networks*, CNN) [3], estas suponen perder la naturaleza multimodal del coloreado de imágenes. La principal razón es que dichas redes se basan, como su nombre indica, en la aplicación de filtros de convolución junto con la minimización de una función clásica de pérdida (*loss*), por ejemplo, la entropía cruzada binaria combinada con funciones de regulación como las conocidas como L1 y L2 [4]. Esta combinación de procesados trata de minimizar la diferencia entre la salida esperada y la

salida obtenida, para conseguir un error mínimo lo que, en el ámbito de la imagen, lleva a la elección de colores "neutros" e imágenes desaturadas con tonos marrones [4].

Para solucionar este problema, en 2014 Ian Goodfellow et. al presentan las *Generative Adversarial Networks* (*GAN*) [2]. El uso de estas redes se popularizó para la generación de imágenes, ya que obtenían resultados muy superiores a los métodos tradicionales tanto de *deep learning* como de técnicas más clásicas. Una red *GAN* se basa en la existencia de un Generador y un Discriminador. El generador es entrenado para obtener imágenes cada vez más fidedignas, mientras la tarea del discriminador es distinguir las imágenes reales de las generadas. A través de los gradientes calculados en el entrenamiento del discriminador, se actualizan los pesos en el generador. Esto está directamente relacionado con el equilibrio de Nash [5] en el cual dos sistemas independientes tienen que cooperar para mejorar en su conjunto. Por lo tanto, no existe una función de *loss* predefinida, sino que el objetivo del entrenamiento es crear una función artificial que el generador aprenda a minimizar, mientras el discriminador mejora en sus prestaciones.

Esta arquitectura presenta una alternativa muy viable a las redes tradicionales, ya que se puede generar una función de *loss* que fomente la saturación y penalice las alternativas conservadoras, generando así imágenes más vivas.

El entrenamiento de una *GAN* sigue siendo a día de hoy un campo muy activo de investigación, ya que todavía se trata de un proceso extremadamente costoso en tiempo de cómputo con resultados un tanto impredecibles en el resultado final [6]. Por este motivo sigue siendo un punto de gran interés, y se experimenta con diferentes arquitecturas y funciones de entrenamiento con el fin de reducir el coste computacional y conseguir resultados más estables.

1.2 Descripción del problema

El coloreado de imágenes es un problema intrínsecamente no determinista, debido a que una fotografía en blanco y negro puede tener varios equivalentes fidedignos a color. Como se puede observar en la figura 1.1, la misma imagen en blanco y negro puede tener varios coloreados realistas, y sin más información, no es posible distinguir cuál de ellos es o no válido.

Para obtener resultados viables y diversos, se requiere de arquitecturas más potentes y adaptadas a la generación de distribuciones multimodales. La arquitectura elegida ha sido la *GAN*, ya que aporta muy buenos resultados en otros campos de la imagen, y supone una nueva aproximación, que ofrece un mayor abanico de posibilidades a la solución del problema. Además de esto, al ser una arquitectura y paradigma relativamente reciente, la investigación del mismo aporta un conocimiento muy útil a nivel académico.

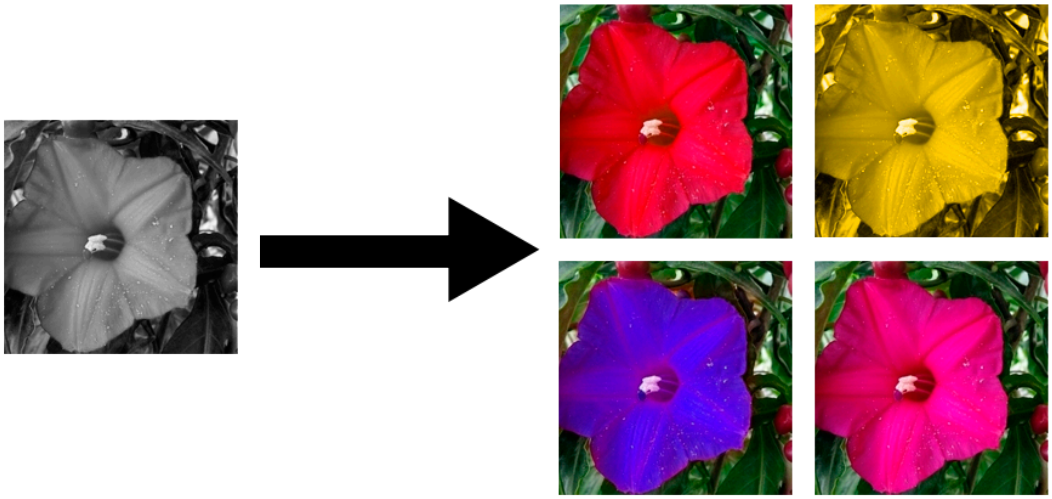


Figura 1.1: Cuatro posibles coloreados de la misma imagen original.

1.3 Objetivos

El objetivo principal que se marca en este proyecto es la creación de una herramienta que permita el coloreado preliminar de fotos antiguas. Con este punto en mente, se establecen una serie de subtareas más concretas como son:

- Investigar modelos actuales de *deep learning* para el coloreado de imágenes.
- Estudiar los fundamentos, estructura y funcionamiento de una *GAN*.
- Desarrollar un modelo que permita el coloreado de imágenes en blanco y negro y estudiar sus resultados.
- Introducir el uso de herramientas de desarrollo web, tanto en *frontend* como en *backend* para hacer accesible el modelo al usuario final.

1.4 Estructura de la memoria

Como se puede intuir por la descripción de los objetivos, este trabajo presenta dos grandes bloques de desarrollo (Modelo de Inteligencia Artificial y Herramienta Web), que correrán en paralelo. Con el fin de facilitar la lectura de este documento, cada uno de esos dos grandes bloques está contenido dentro de un capítulo siendo precedido por las definiciones generales y un capítulo de conclusiones que servirá de puente de unión entre ambas fases de tal manera que la estructura de la memoria es la siguiente:

- **Capítulo 1: Introducción.** Presentación del problema y los objetivos de este trabajo.
- **Capítulo 2: Fundamentos.** Introducción al lector a los fundamentos teóricos básicos para tratar el problema
- **Capítulo 3: Desarrollo del modelo.** Descripción del proceso de creación del modelo de *deep learning*, y guía al lector en las diversas aproximaciones acontecidas y los pasos metodológicos del desarrollo
- **Capítulo 4: Desarrollo de la herramienta.** Compendio de todos los aspectos del desarrollo de la herramienta que sirve como nexo de comunicación entre el usuario y el modelo resultante del capítulo anterior. Así mismo, se hará una descripción arquitectural y de diseño de la herramienta.
- **Capítulo 5: Conclusiones y Futuros Desarrollos.** Exposición de las conclusiones del proyecto, así como problemas encontrados, fallos del sistema y posibles desarrollos futuros.

Capítulo 2

Fundamentos

En este capítulo se presentan, por un lado, todos los fundamentos teóricos tanto matemáticos como relativos a las redes de neuronas necesarios para entender en profundidad el presente trabajo. Por el otro, también contiene una descripción de las herramientas tanto tecnológicas como metodológicas aplicadas en el desarrollo del proyecto.

2.1 Fundamentos teóricos

En primer lugar, para entender en profundidad el desarrollo de este trabajo es necesario establecer algunas definiciones. Éstas se focalizarán mayoritariamente en los conceptos necesarios dentro de las técnicas de aprendizaje por refuerzo (*Reinforcement Learning*), como las denominadas técnicas de *Deep Learning*. Ambos puntos convergen en los conceptos y definiciones necesarias para la creación de las redes conocidas como *Generative Adversarial Networks* (*GAN*), núcleo de los desarrollos llevados a cabo en este trabajo.

2.1.1 Equilibrio de Nash

La primera de las definiciones necesarias para entender los desarrollos llevados a cabo es el equilibrio de Nash [7]. Este concepto, que está ligado a la teoría de juegos, es aplicable a juegos de dos o más jugadores si se cumplen dos requisitos:

- El jugador conoce su estrategia óptima.
- El jugador conoce la estrategia de todos los demás jugadores.

Teniendo en cuenta esto, la definición del equilibrio de Nash establece que:
"Un jugador no modifica su estrategia mientras sus rivales mantengan la suya, por lo que cada jugador está utilizando la estrategia más óptima en cada momento".

Este fundamento es la base teórica del aprendizaje de las *GAN* (definición que se verá en la página 12 en al sección dedicada a tal concepto), ya que estas pueden modelarse como juegos de dos jugadores (generador y discriminador).

2.1.2 La operación de convolución

Una operación de convolución bidimensional se basa en la aplicación de un *kernel* a una imagen, que en el fondo no es más que una matriz de dos dimensiones (para imágenes de 1 canal).

El *kernel* es simplemente una matriz bidimensional de pesos, que se aplica de manera deslizante sobre la imagen, calculando una multiplicación "elemento-a-elemento" respecto a la ventana de la imagen sobre la que se encuentra. El resultado de esta operación es reducido mediante una suma, y el resultado de esta será aplicado al píxel sobre el que se encuentra el centro del *kernel* [1].

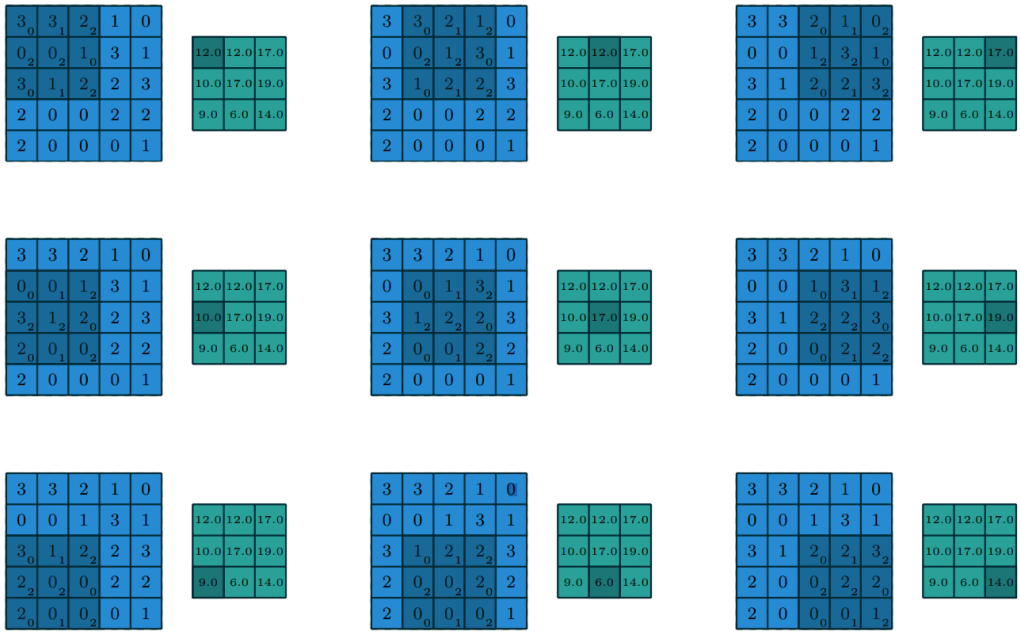


Figura 2.1: Aplicación de una convolución[1].

Esta operación se aplica sobre cada elemento de la matriz, produciendo a su vez otra matriz de características.

Cuando la imagen de entrada tiene más de un canal, se utiliza una colección de *kernels* llamada **filtro**. Un filtro tiene un *kernel* único para cada canal. Cada uno de estos *kernels* se aplica a su respectivo canal, y el resultado de todos ellos se suma para producir la matriz de características.

2.1.3 Métricas utilizadas

Con el fin de medir el correcto desempeño de cada una de las aproximaciones que se van a efectuar en este trabajo para el problema del coloreado de imágenes, es necesario definir un par de medidas que permitan cuantificar la similitud para permitir "premiar" o "castigar" a los modelos según su desempeño.

Mean Square Error (MSE)

El Error Cuadrático Medio o más conocido por sus siglas en inglés, *MSE* [8], es un estimador estadístico muy usado en problemas de optimización. Este estimador trata de medir la diferencia entre los resultados de la función objetivo y la definida por un modelo. La función resultante recibe el nombre de función de *loss*. Con ello, el resultado de la función de *loss* usando el MSE devuelve el cálculo de la media de los cuadrados de los errores, que se describe a continuación:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2 \tag{2.1}$$

Según la función descrita en 2.1, dado un vector Y de datos, con n elementos,y un vector de predicciones $\hat{Y}$, el MSE sería la suma de las diferencias al cuadrado de los elementos Y_i y X_i, para cada $i \in [1, n]$, dividido por el número de elementos en los vectores.

Structural Similarity Index (SSIM)

El *Structural Similarity Index* [9], o *SSIM*, es una medida para cuantificar la similitud percibida entre dos imágenes. El objetivo en el diseño de dicha medida es mejorar el deficiente comportamiento observado en imágenes por otras medidas como puede ser, por ejemplo, el MSE.

En el *SSIM*, sea $N \times N$ un conjunto de submatrices definidas sobre una imagen, dadas dos submatrices x e $y \in N \times N$, la similitud entre ellas se calcula mediante la fórmula simplificada mostrada en la ecuación 2.2

$$SSIM(x, y) = \frac{(2\mu_x\mu_y + c1)(2\sigma_{xy} + c2)}{(\mu_x^2 + \mu_y^2 + c1)(\sigma_x^2 + \sigma_y^2 + c2)} \tag{2.2}$$

Siendo las variables:

- μ_x tμ_y: la media de la submatriz x e y, respectivamente.
- σ_x^2 y σ_y^2: la varianza de la submatriz x e y, respectivamente.
- σ_{xy}: el coeficiente de correlación de x e y.

- $c1$ y $c2$: dos variables de estabilización basadas en el rango dinámico de las ventanas.

La ecuación 2.2 está simplificada, ya que normalmente se utilizan factores de ponderado para la luminancia, crominancia y estructura. Estos parámetros habitualmente reciben los nombres de α, β y γ respectivamente. En este trabajo se han usado los valores simplificados recomendados en el artículo original, fijándolos todos a 1.

2.2 Redes Neuronales Artificiales

A continuación se establecerán aquellas definiciones sobre redes de neuronas artificiales que se consideran necesarias para comprender los desarrollos llevados a cabo en el capítulo 3 de esta memoria.

2.2.1 Funciones de activación

En las Redes Neuronales Artificiales, la función de activación de un nodo define la salida de dicho nodo respecto a sus entradas. Existen muchas funciones de activación ampliamente utilizadas en *deep learning*. En esta sección se definirán las funciones utilizadas en este trabajo, y el motivo por el que han sido elegidas.

Sigmoide

La función sigmoide transforma la entrada a una escala de salida de $(0, 1)$, donde los valores altos tienden asintóticamente a 1 y los valores bajos tienden de manera asintótica a 0.

$$f(x) = \frac{1}{1 - e^{-x}} \tag{2.3}$$

Tangente hiperbólica

La función sigmoide transforma la entrada a una escala de salida de $(-1, 1)$, donde los valores altos tienden asintóticamente a 1 y los valores bajos tienden de manera asintótica a -1.

$$f(x) = \frac{2}{1 - e^{-2x}} - 1 \tag{2.4}$$

ReLU

La ReLU, introducida inicialmente en [10], es una función de activación lineal, que se define como ”la parte positiva de su argumento”. El problema principal de funciones como la sigmoide o la tangente hiperbólica es que saturan para valores de entrada muy altos o muy bajos, manteniendo su salida casi constante. En las redes neuronales profundas con muchas

capas intermedias el uso de funciones no lineales como las mencionadas anteriormente hace que en cada capa los gradientes se hagan cada vez menores, fenómeno conocido como *desvanecimiento de gradientes* [11]. El uso de una función que no sature, como la ReLU, ayuda a paliar este problema en redes muy profundas, y se ha establecido como estándar *de facto* en las redes neuronales con muchas capas ocultas. Sin embargo, las funciones no lineales siguen utilizándose fundamentalmente en las capas de salida, ya que acotan su resultado, mientras que el conjunto imagen de la ReLU es $[0, \inf)$

$$f(x) = x^+ = max(0, x) \tag{2.5}$$

LeakyReLU

Si los valores de entrada de la ReLU son mayoritariamente menores a 0, la derivada de esta función (utilizada para calcular los gradientes) será 0. En este caso, el entrenamiento quedará bloqueado, y el modelo no será capaz de continuar el aprendizaje. Para paliar este problema, se introduce la **LeakyReLU**, una modificación de la ReLU original, que añade un factor a, normalmente 0.1, que evita que la derivada sea igual a 0 en valores negativos, y permite evitar el desvanecimiento de gradiente de manera más eficiente.

$$f(x) = \begin{cases} x & \text{si} \quad x > 0 \\ ax & \text{si} \quad x \leq 0 \end{cases} \tag{2.6}$$

2.2.2 Redes Neuronales Convolucionales

Las Redes Neuronales Convolucionales (*Convolutional Neural Network, CNN*) [3, 12, 13] son un modelo de red análogo a las redes tradicionales que han sido ampliamente utilizadas para el tratamiento de imágenes en los últimos años. Estas redes tienen una primera parte que se denomina extractor en el que las neuronas tratan de generalizar las características que serán utilizadas por la segunda parte de la red, que es la que realizará la función de clasificación o regresión. Para ello, el extractor se centrará en tres elementos principales: La anchura, la altura y el número de canales. La principal diferencia de las redes convolucionales con respecto a las tradicionales, por lo tanto, es ese denominado extractor de características, el cual se basa, en primer lugar, en lo que se denominan capas convolucionales. Dichas capas se basan en la aplicación de la operación de convolución como la vista en la sección 2.1.2 de la página 6, que incluye un número de filtros entrenables. Tras la aplicación de cada uno de los filtros, el resultado es un conjunto de características que se ordenan en una matriz resultante de la concatenación de las salidas de las convoluciones, produciendo una salida multicanal. Tras la operación de convolución, a cada elemento de las matriz de características se le aplica una operación no lineal (usualmente una *ReLU* [14]) de manera individual.

Junto a las operaciones de convolución, las operaciones de *pooling* son el otro bloque fundamental en una *CNN*. Una operación de *pooling* es similar a una convolución convencional, pero tiene dos diferencias fundamentales: en primer lugar, la operación de *pooling* es una combinación no lineal para lo que se utilizan funciones como la media (*average pooling*) o el máximo (*max pooling*); en segundo lugar, el *striding* o distancia entre dos puntos consecutivos de aplicación del filtro. Mientras que en las capas convoluciones este suele ser de distancia 1 en las capas de *pooling* se suele establecer al tamaño del filtro.

Al utilizar un *stride*, la salida de una capa de *pooling* tiene su dimensionalidad reducida. Por ejemplo, la aplicación de una capa de *max pooling* de dimensiones 2×2 con *stride* 2 en una imagen de dimensión $N \times N$ tendrá como salida una imagen con tamaño $(N/2) \times (N/2)$, en la que cada uno de los píxeles será el máximo de su vecindario.

2.2.3 Autoencoders

Los *autoencoders* [15] son redes neuronales para aprendizaje no supervisado que extraen características comprimiendo la entrada en un espacio latente, y reconstruyendo la salida para que sea igual a la entrada. Esto es equivalente a hacer que el modelo mimetice la función identidad $y = f(x)$.

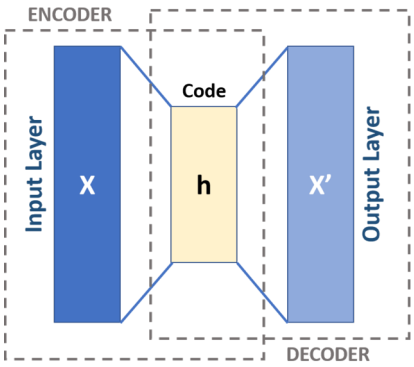


Figura 2.2: Estructura de un *autoencoder*.

Una vez el modelo aprenda a generar una salida igual a la entrada, la capa oculta contendrá toda la información necesaria para reconstruir la entrada, pero comprimida. Esto resulta de gran utilidad para reducir la dimensionalidad de la entrada de manera automática, y es una de las bases teóricas para arquitecturas avanzadas de *deep learning* como los *Stacked Autoencoders (SAE)* [16] o los *Variational Autoencoders (VAE)* [17] . En la figura 2.2 se muestra un esquema de la arquitectura de un autoencoder[1].

[1] https://commons.wikimedia.org/wiki/File:Autoencoder_schema.png

2.2.4 U-Net

La arquitectura **U-Net** [18] surge de las llamadas *Fully Convolutional Networks* [12]. Se conforma fundamentalmente de dos segmentos: El segmento de *downsampling* y el segmento de *upsampling*, que se alinean simétricamente, lo que da a la red una estructura de **U**, similar a la que se puede ver en el figura 2.3. El segmento de *downsampling* es el encargado de reducir la dimensionalidad de la entrada y abstraer la información relevante, y el segmento de *upsampling* reconstruye la imagen de salida en base a las características obtenidas en el anterior segmento. Esta estructura se basa fundamentalmente en la de un *autoencoder*, vistos anteriormente en la sección 2.2.3. En el tratamiento de imágenes, la localidad espacial es fundamental. Esto se consigue en la **U-Net** mediante conexiones entre los bloques simétricos de ambos segmentos, que proporcionan a la rama de *upsampling* la información de contexto necesaria para generar la imagen de salida.

Cada bloque del segmento de *downsampling* consiste en la aplicación de dos convoluciones, cada una seguida por una función de activación ReLU y una operación de *max-pooling* 2×2 con *stride* 2. El objetivo es reducir la dimensionalidad espacial de la imagen a la mitad. Así mismo, cada bloque subsiguiente duplica el número de filtros, lo que provoca un aumento de canales y, por tanto, de características. .

Para el segmento de *upsampling*, se utiliza inicialmente una capa de convolución transpuesta [19], seguida por la aplicación de dos convoluciones que reducen el número de canales. Finalmente, se aplica una última capa de convolución que reduce los filtros al número de canales necesarios para la salida.

Como podemos ver en la figura 2.3, cada capa del segmento de *downsampling* está unida a la capa simétrica del segmento de *upsampling*. En la imagen, cada cuadro azul representa un mapa de características, salido de un proceso de convolución. El número de canales aumenta en cada capa del segmento de *downsampling* y se reduce en cada capa del segmento de *upsampling*.

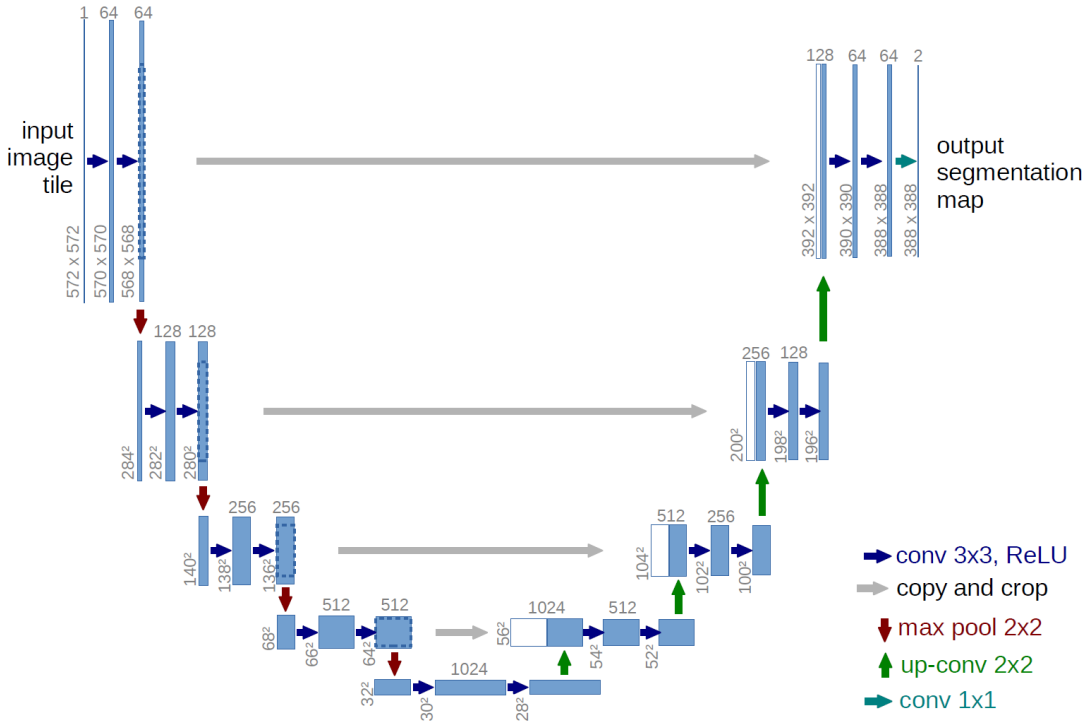


Figura 2.3: Arquitectura original U-Net.

2.2.5 *Generative Adversarial Networks*

Las *Generative Adversarial Networks* [2], de aquí en adelante *GAN*, son un modelo generativo de reciente aparición, que toma una aproximación basada en la teoría de juegos. La red aprende a generar una muestra de una distribución estadística desconocida mediante un juego de 2 jugadores. La arquitectura básica de una *GAN* se compone de dos modelos, el **generador** y el **discriminador**. El generador construye muestras basadas en una distribución aleatoria (espacio latente). El discriminador es un clasificador clásico, que recibe muestras reales (*ground truth*) y muestras generadas, y se entrena para distinguir entre ellas y poder distinguir una muestra falsa de una real. La tarea del generador es construir muestras que ”engañen” al discriminador, y la del discriminador es evitar ser ”engañado”.

El objetivo del entrenamiento de una *GAN* es obtener un estado de Equilibrio de Nash[5] entre el generador y el discriminador. Para esto, se modela la función objetivo como una función *min-max*. El discriminador trata de maximizar la función objetivo, mientras el generador trata de minimizarla.

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} log D_{\theta_d}(x) + \mathbb{E}_{z \sim p_z} log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right] \tag{2.7}$$

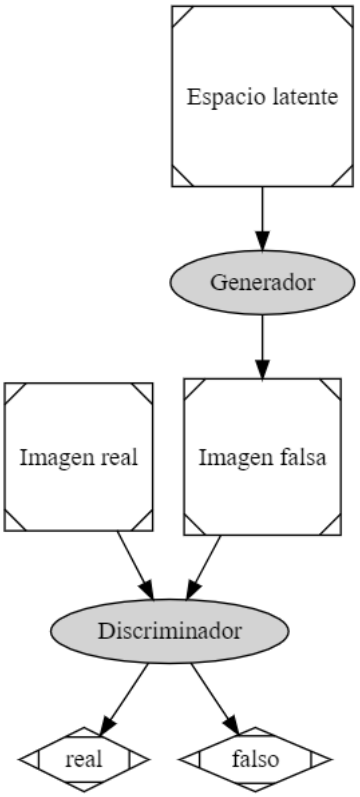


Figura 2.4: Arquitectura simplificada de una *GAN*.

Si se separa la ecuación 2.7, se pueden obtener las funciones para el generador (Eq. 2.8) y el discriminador(Eq. 2.9).

$$\min_{\theta_g} \left[\mathbb{E}_{z\sim p_z} log(1 - D_{\theta_d}(G_{\theta_g}(z)))\right] \tag{2.8}$$

$$\max_{\theta_d} \left[\mathbb{E}_{x\sim p_{data}} log D_{\theta_d}(x) + \mathbb{E}_{z\sim p_z} log(1 - D_{\theta_d}(G_{\theta_g}(z)))\right] \tag{2.9}$$

Como se puede ver en las ecuaciones 2.8 y 2.9, el discriminador debe maximizar la función para reconocer las muestras generadas como falsas y las muestras del *ground truth* como verdaderas. El generador debe minimizar la posibilidad de que el discriminador clasifique las muestras generadas como falsas. En la práctica, la minimización de la ecuación 2.8 no da buenos resultados, ya que cuando la muestra es clasificada como falsa, los gradientes del discriminador son muy planos. Por lo tanto, se invierte esta función para que el generador maximice la probabilidad de que el generador se equivoque, en lugar de minimizar la probabilidad de que obtenga el resultado correcto. Esto da lugar a la ecuación 2.10.

$$\max_{\theta_g} \left[\mathbb{E}_{z\sim p_z} log(D_{\theta_d}(G_{\theta_g}(z)))\right] \tag{2.10}$$

Debido a la complejidad de alcanzar un equilibro de Nash entre los dos jugadores, el entrenamiento de una *GAN* es normalmente muy inestable, y llegar a un punto de convergencia es difícil. Por esto, se han desarrollado diversas extensiones de las *GAN*, que se orientan a obtener una mayor estabilidad y facilidad de entrenamiento.

DCGAN

Las *DCGAN* (***D**eep **C**onvolutional* **GAN**) [20] son una extensión a las *GAN* tradicionales, en la que se utilizan modelos convolucionales profundos tanto para el generador como para el discriminador. Estas redes son ideales para la generación de imágenes, ya que combinan la potencia de las *GAN* como modelo generativo con las operaciones convolucionales, ideales para el tratamiento de imágenes como las que se describen en los siguientes apartados.

CGAN

Las *CGAN* [4, 21, 22] (***C**onditional **GAN***) añaden un condicionante a la entrada tanto del generador como del discriminador. Normalmente, una *GAN* utiliza como entrada un espacio latente. En una *GAN* tradicional, no existe ninguna manera de controlar la moda de las muestras generadas. Sin embargo, añadiendo información condicionante (por ejemplo, una *class label*) se puede dirigir el proceso de generación de muestras, si tanto el generador como el discriminador se condicionan con una información extra y, concatenando la misma a la

entrada original.
De esta manera, la función objetivo de la red pasa a ser la indicada en la ecuación 2.11

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} log D_{\theta_d}(x|y) + \mathbb{E}_{z \sim p_z} log(1 - D_{\theta_d}(G_{\theta_g}(z|y))) \right] \tag{2.11}$$

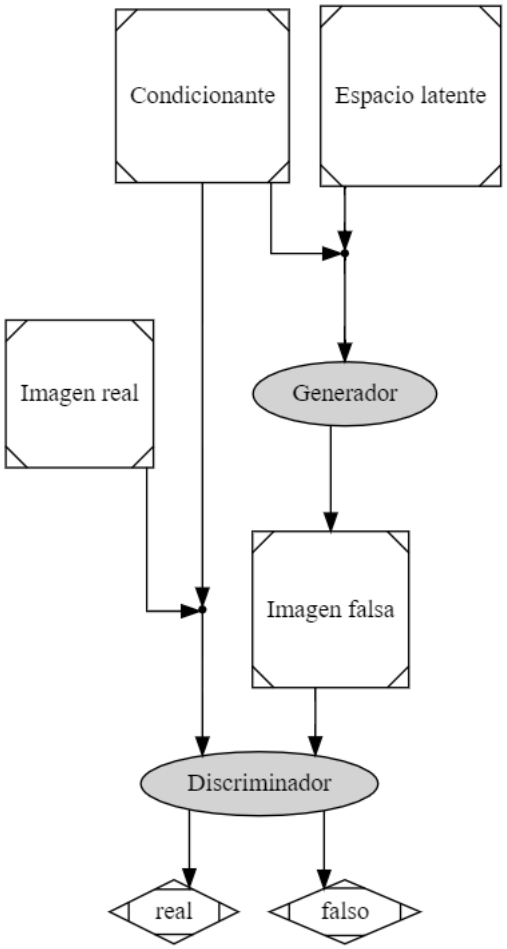


Figura 2.5: Arquitectura en alto nivel de una *CGAN*.

En la figura 2.5 se muestra un esquema en alto nivel de la arquitectura de una CGAN.

PatchGAN

La arquitectura **PatchGAN**, inicialmente introducida para el modelo **pix2pix** [4] implica una modificación en el discriminador, en la cual, en lugar de clasificar una imagen como verdadera/falsa, divide esta imagen en parches de $N \times N$ píxeles y clasifica cada uno de ellos como verdadero/falso. Con esto se consigue un mayor detalle en las áreas de mayor frecuen-

cia, dando lugar a imágenes más definidas y detalles más finos, mientras se mantienen los detalles de baja frecuencia mediante una función de regularización como la L1.

2.2.6 Problemas del entrenamiento de *GAN*

Las *GAN* son modelos muy complejos de entrenar, que pueden cambiar enormemente su comportamiento con pequeños cambios en los hiperparámetros. Además, sufren de diversos problemas durante el entrenamiento, que pueden manifestarse tanto al principio del mismo como después de varias iteraciones de entrenamiento (*epochs*), lo cual hace su evaluación extremadamente costosa. Si bien en las siguientes subsecciones se explicarán en detalle, algunos de los problemas más habituales de las *GAN* son: el colapso modal, la no convergencia o la disminución de gradientes

Colapso modal

La base del aprendizaje en una *GAN* es el equilibrio entre el generador y el discriminador. El generador debe generar muestras indistinguibles de una muestra "real" para el discriminador. Las distribuciones de datos reales son generalmente multimodales, y el generador debe poder generalizar dichas distribuciones. Sin embargo, este comportamiento teórico puede verse alterado por un desequilibrio en el que el generador converge generando una única muestra que pueda engañar al discriminador, es entonces cuando se habla del **colapso modal**.

En este caso, da igual el espacio latente que se introduzca al generador, este siempre tendrá como salida la misma muestra, que será catalogada como real. El resultado es que los gradientes de entrenamiento del discriminador tienden a cero no proveyendo tampoco de retroalimentación al generador, el cual proveerá siempre la misma salida. Existen colapsos parciales, pero un colapso total implica que debe pararse el entrenamiento, ya que no existe posibilidad para ninguna de las redes de continuar entrenándose.

Por este motivo, en el entrenamiento de una *GAN*, es vital ajustar la potencia del discriminador y el generador. Así, ninguno avasallará al otro y permitirá que se produzca una mejora en los resultados a medida que ambos modelos son entrenados.

No convergencia

Como ya se explica en la sección 2.2.5, las redes *GAN* basan su funcionamiento en alcanzar un equilibrio de Nash. Sin embargo, esto no siempre es posible, por lo que estos modelos pueden no converger hasta que se utilice el conjunto de hiperparámetros adecuado.

Para demostrar esto, se utilizará un algoritmo *minimax* muy sencillo para observar sus propiedades. Defínase la función V como la divergencia entre dos funciones, la cual determina

la situación del algoritmo según la ecuación 2.12

$$\min_{P2} \max_{P1} V(x, y) = xy \tag{2.12}$$

Según dicha función, el jugador P1 desea minimizar el valor de V, mientras que el jugador P2 desea maximizarlo. Para conseguirlo actualizarán los valores de x e y utilizando el descenso de gradiente, siguiendo las ecuaciones 2.13 y 2.14, respectivamente.

$$\Delta x = \alpha * \frac{\partial(xy)}{\partial(x)} \tag{2.13}$$

$$\Delta x = \alpha * \frac{\partial(xy)}{\partial(x)} \tag{2.14}$$

En las mencionadas ecuaciones α es el *learning rate* que, en este caso particular, se fijará en 0.1. Si se itera sobre el juego definido por V, se observa cómo no llega a converger, tal y como se puede aprecia en la figura 2.6. En cambio, si se aumenta el *learning rate* hasta 0.2, en cada iteración se observa una mayor divergencia, y el equilibrio de Nash se hace inalcanzable (figura 2.7).

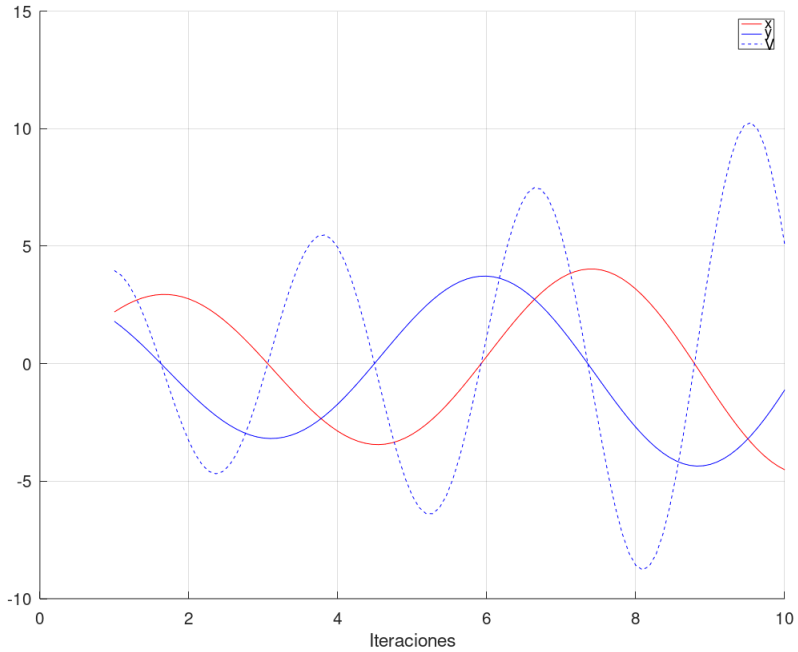


Figura 2.6: Evolución de las variables x,y y la función V durante el entrenamiento, con $alpha$=0.1

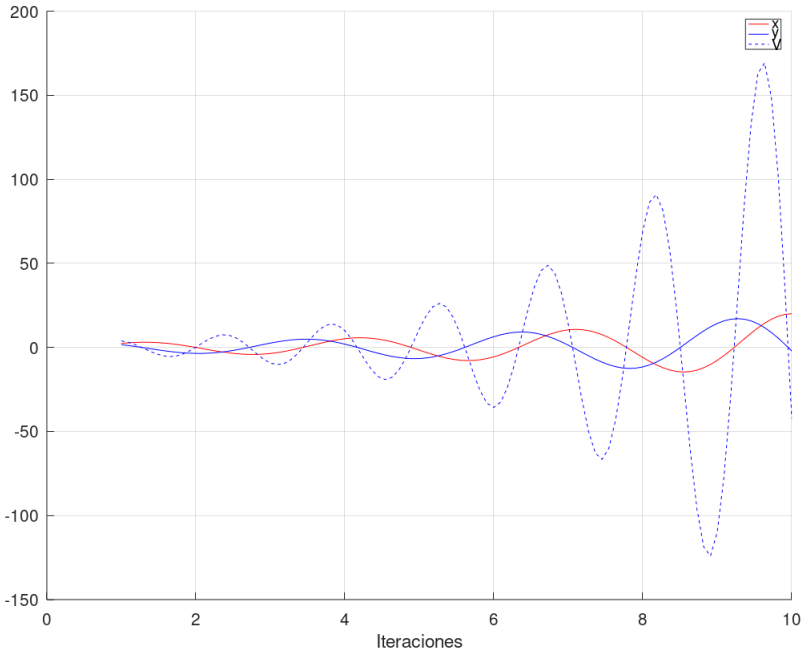


Figura 2.7: Evolución de las variables x,y y la función V durante el entrenamiento, con $alpha$=0.2

Esto sirve como demostración de que algunas funciones de *loss* no convergen con el descenso de gradiente, por lo que la convergencia de una *GAN* no está garantizada.

Disminución de gradientes

La disminución de gradientes, también conocida como *Vanishing Gradient Problem*, puede verse como un caso inverso al colapso modal. Si el discriminador consigue distinguir perfectamente las imágenes reales de las generadas, los gradientes que transmitirá al generador se desvanecerán. En este caso, el generador casi no actualiza sus pesos y no se entrena provocando otro colapso.

2.3 Herramientas y librerías utilizadas

Los fundamentos matemáticos y teóricos del aprendizaje automático son profundamente complejos, y por ello se han desarrollado varias herramientas y librerías que proveen de una abstracción sobre los mecanismos como la asignación de pesos, los pases *feedforward*, o el mecanismo de *backpropagation*. Dichas herramientas permiten definir modelos de una manera abstracta, y son de vital importancia para el desarrollo de este proyecto.

De igual manera, para el desarrollo de la aplicación web, se han utilizado librerías modernas para abstraer conceptos como el desarrollo de una API REST, o de las peticiones asíncronas al *backend*.

2.3.1 TensorFlow

Desarrollada por Google, **TensorFlow** [23] es una librería de código abierto para el **cálculo tensorial**, la cual ha sido ampliamente utilizada en desarrollos de **aprendizaje automático** (*machine learning*).

Para este trabajo se ha elegido esta herramienta en lugar de otras alternativas como Pytorch [24], MxNet [25] o Theano [26] ya que cuenta con gran adopción y soporte por parte de la comunidad. Además, está ampliamente documentada y existe una gran cantidad de documentación, ya sea oficial o en base a proyectos de código abierto.

Recientemente Google ha liberado la versión 2.0 de TensorFlow, que trae enormes cambios en el *workflow* de la herramienta, permitiendo un desarrollo más sencillo. Así mismo, TensorFlow dispone de *bindings* para muchos lenguajes de programación de amplia adopción, como Python, C++, JavaScript, C#...

En este trabajo se ha utilizado la versión 2.0 de TensorFlow en Python [27], que permite un prototipado rápido e introduce un API de alto nivel (Keras) para el diseño de modelos complejos.

2.3.2 React

Para la aplicación web se han valorado diversas alternativas como Angular o Vue. Finalmente, se ha seleccionado la librería **React** [28] en Javascript para construir las interfaces de usuario en web.

React es una librería para el diseño de aplicaciones web, que provee un modelado simple basado en el modelo **MVC**, que promueve la modularización y composición de componentes simples para construir aplicaciones complejas de una manera sostenible y escalable.

La selección de esta herramienta ha venido motivada porque se trata de una librería de código abierto y gratuita desarrollada por Facebook, que en la actualidad es la biblioteca más popular en el diseño de **SPA** (*Single Page Applications*) lo que asegura un mantenimiento a largo plazo. A mayores, la librería cuenta con una enorme cantidad de módulos y componentes que permiten prototipar rápidamente una interfaz compleja y flexible.

2.3.3 Flask

Para el desarrollo del *backend* se han valorado varias alternativas como Node.js + Express.js, Rocket o Laravel. Finalmente, se ha elegido Flask [29] debido a la facilidad de integrar

el modelo de TensorFlow (Desarrollado en Python).

Flask es un *framework* minimalista basado en Python, que permite construir **aplicaciones web** de una manera sencilla y rápida. En concreto, provee opciones para diseñar una **API REST** muy rápidamente. El uso de una herramienta en Python es de gran ayuda, ya que integrar la API con el modelo desarrollado en TensorFlow es muy sencillo, y es fácil de desplegar y probar.

En muy pocas líneas, Flask permite generar un prototipo para la API, y poder insertar el modelo congelado de TensorFlow sin salir de un entorno Python, ni usar otras opciones de despliegue como Tensorflow Serving.

2.3.4 Docker

Para el **despliegue** de la herramienta, se ha optado por utilizar un sistema basado en **microservicios** [30]. Las aplicaciones basadas en esta conceptualización permiten crear instancias de servicios modulares, comunicarlas entre ellas y agilizar tanto el despliegue como la escalabilidad de la aplicación.

Dentro de las posibilidades existentes, **Docker** [31] es un sistema de contenedores simple y potente, que permite definir contenedores con un archivo de definición sencillo. Para ello, Docker crea una imagen ligera, independiente del *host* y portable. El despliegue se hace inmediato y es independiente del sistema operativo utilizado.

Tanto los modelos preentrenados como las imágenes de Docker están disponibles públicamente como recurso, por lo que se puede hacer el despliegue en cualquier máquina. En este caso se han utilizado dos imágenes de docker, una para el *frontend* de la herramienta y otra para el *backend*. En el anexo A se mostrará la manera de desplegar la herramienta utilizando Docker de manera sencilla.

2.4 Metodología de desarrollo

Para la realización de este trabajo se adoptará una aproximación basada en prototipos [32], según la cual se construirá una primera aproximación que, tras ser evaluada, permitirá identificar los cambios pertinentes y nuevas características así como eliminar funciones innecesarias. Dichas modificaciones se implementan en un siguiente prototipo que comienza nuevamente el ciclo de evaluación, identificación y refinamiento. Dentro de cada prototipo, y arquitectura de modelo, se probará la optimización de varios hiperparámetros de las arquitecturas y se seleccionarán los que provean mejor resultado.

Para el desarrollo de la herramienta será igualmente utilizada la metodología de prototipado, en la que se identificarán requisitos y casos de uso, y se irán desarrollando y evaluando

de manera cíclica. Esto es especialmente importante en la parte de *frontend*, ya que es donde los casos de uso son más evidentes.

Al tratarse de una metodología iterativa no se puede estar al 100% seguro de la evolución del proyecto. Sin embargo, de manera orientativa, una estimación y planificación inicial de las tareas se puede ver en la figura 2.8. El resumen en costes de esta figura se observa a su vez en la tabla 2.9. Se fija el coste del trabajador como 20€/hora trabajada.

Id	Modo de tarea	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras	Nombres de los recursos
1		Estudio del estado del arte	5 días?	mar 01/10/19	vie 11/10/19		David
2		Estudio de la implementación en Tensorflow	4 días?	mar 01/10/19	mié 09/10/19		David
3		Generador	5 días?	mié 09/10/19	sáb 19/10/19	2;1CC	David
4		Discriminador	5 días?	mié 09/10/19	sáb 19/10/19	2;1CC	David
5		GAN	4 días?	sáb 19/10/19	dom 27/10/1	2;3	David
6		Modelo Creado	0 días	dom 27/10/1	dom 27/10/1	5	David
7		Entrenamiento del modelo	20 días?	dom 27/10/19	dom 03/11/19	6	Equipo de entrenamiento
8		Elección de la herramienta backend	4 horas?	mar 01/10/19	mar 01/10/19		David
9		Diseño de la API	1 día?	mié 02/10/19	jue 03/10/19	8	David
10		Integración con el modelo	1 día?	vie 04/10/19	sáb 05/10/19	9FC+50%	David
11		Backend creado	0 días	sáb 05/10/19	sáb 05/10/19	10	David
12			1 día?	mar 01/10/19	mié 02/10/19		David
13		Elección de la herramienta frontend	8 horas?	mar 01/10/19	mié 02/10/19		David
14		Diseño y análisis del software	1 día?	jue 03/10/19	vie 04/10/19	13	David
15		Implementación del frontend	4 días?	sáb 05/10/19	dom 13/10/19	14;9CC+50%	David
16		Herramienta finalizada	0 días	dom 13/10/19	dom 13/10/19	11;15	David
17			1 día?	mar 01/10/19	mié 02/10/19		David
18		Redacción de la memo	20 días?	mié 02/10/19	mar 12/11/19	1CC+20%;8CC+20	David

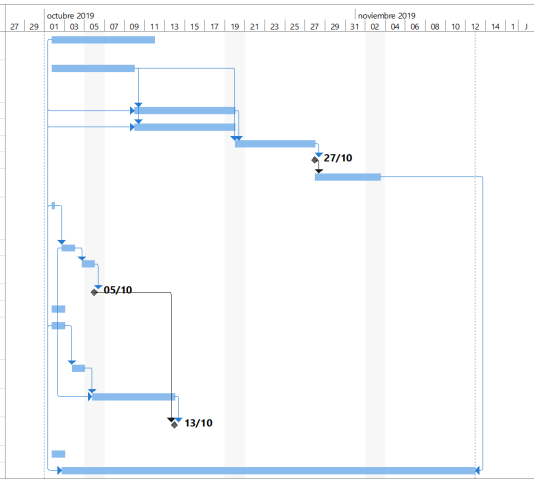


Figura 2.8: Planificación inicial del proyecto.

	Comienzo	Fin
Actual	mar 01/10/19	jue 14/11/19
Previsto	mar 01/10/19	jue 14/11/19
Real	NOD	NOD
Variación	0d	0d

	Duración	Trabajo	Costo
Actual	33d?	428h	8.560,00 €
Previsto	33d	428h	0,00 €
Real	0d	0h	0,00 €
Restante	33d?	428h	8.560,00 €

Porcentaje completado:

Duración: 0% Trabajo: 0%

Cerrar

Figura 2.9: Estimación de coste del proyecto.

Capítulo 3

Desarrollo del modelo

El presente capítulo describe el proceso de desarrollo del modelo inteligente desarrollado para solucionar el problema de coloreado de imágenes planteado en la página 2. Si bien aquí se describe de manera lineal, este desarrollo se ha solapado en el tiempo con el desarrollo descrito en el capítulo 4.

3.1 Diseño del modelo

En esta sección se describen las decisiones de diseño tomadas para completar el desarrollo del modelo. Cuestiones como la decisión del espacio de color, las entrada a tomar o la arquitectura son cruciales a la hora de resolver el problema.

3.1.1 Selección del modelo de color

Un modelo de color es una abstracción matemática que permite representar el color como n-tuplas, usualmente 3 o 4-tuplas. Es decir, asocian un valor numérico a cada color. Usualmente, al trabajar con imágenes, se utiliza el modelo **RGB**, que modela cada color como una 3-tupla, siendo sus componentes los valores de color **R**ed (Rojo), **G**reen (Verde) y **B**lue (Azul) respectivamente. El **RGB** es un modelo aditivo, es decir, se suman los componentes de la tupla para obtener el valor final de color. Para este trabajo, se ha elegido el modelo **YUV**, en el que el canal **Y** representa la luminancia (*luma*), es decir, el brillo de la imagen, y los canales **U** y **V** representan la crominancia (*croma*), es decir, los datos de color. El uso del modelo **YUV** para coloreado de imagen permite reutilizar la imagen en blanco y negro como canal **Y** de la imagen resultante, dando la posibilidad de abstraer al sistema de los datos de luminancia en la imagen resultante, y así computar solamente los datos de color. En el modelo **RGB**, un cambio en cualquiera de los componentes implica un cambio en el brillo de la imagen final. En cambio, en **YUV** el brillo y el color están desacoplados, haciendo más sencilla la inferencia del color y la fidelidad respecto a la imagen original.

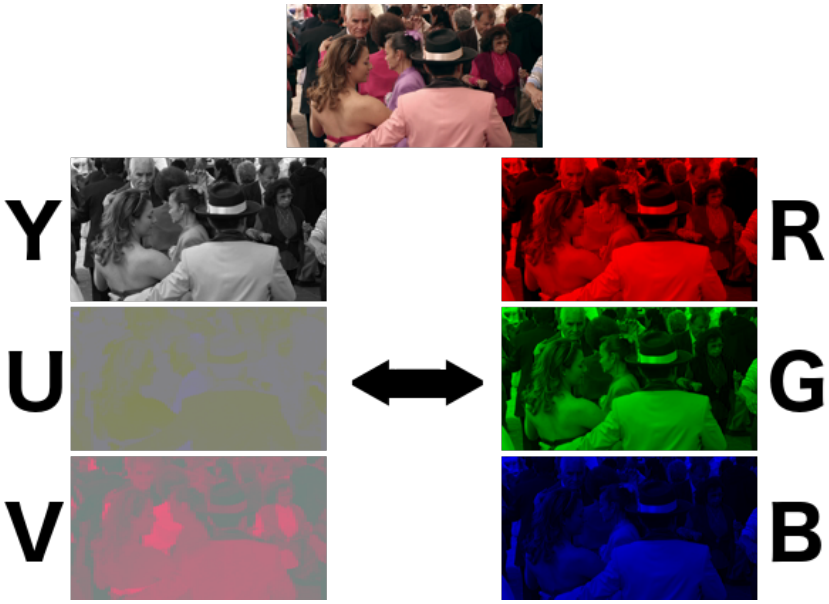


Figura 3.1: Comparativa de los canales de una imagen a color en RGB y YUV.

3.1.2 Entradas y salidas del modelo

Tras una serie de pruebas iniciales con otras aproximaciones, se decidió optar por realizar una aproximación con la arquitectura *PatchGAN* descrita en la sección 2.2.5. Para ello es necesario el describir las entradas y salidas tanto del generador como el discriminador. Como ya se especificó en la sección 3.1.1, se ha utilizado el modelo de color **YUV**. La entrada principal del generador es una distribución aleatoria de dimensiones (W,H,1), con la que el generador producirá imágenes. Así mismo, como condicionante, se utiliza la imagen a tratar, de dimensiones (W,H,1), la cual se concatena al ruido aleatorio, produciendo una entrada al modelo de dimensiones (W,H,2).

En cuanto al discriminador, la entrada es un tensor (W,H,2), correspondiente a los canales **U** y **V** de la imagen (original o generada). Esta entrada se concatena con el mismo condicionante que el generador (la imagen original en blanco y negro) de dimensiones (W,H,1), produciendo una entrada de dimensiones (W,H,3).

El generador tiene como salida un tensor (W,H,2), en el que la última dimensión corresponde a los canales **U** y **V** del modelo YUV (es decir, el color). Mientras que el discriminador tiene como salida un tensor de tamaño (30,30,1), en el que las dos primeras dimensiones son cada uno de los parches a clasificar, y la última dimensión es la probabilidad de que la entrada sea verdadera, entre 0.0 y 1.0.

3.2 Prototipo 1: Generador

3.2.1 Diseño

En el primer diseño del generador se ha utilizado un modelo clásico sin tener en cuenta las *GAN*, para poder tener rápidamente un prototipo funcional sobre el que construir el modelo. El modelo, utilizado ampliamente en la literatura asociada es el que se conoce como U-Net o modelo U, que se encuentra explicado en la página 11.

3.2.2 Desarrollo

Este tipo de modelos, consta de dos tipos de bloques los conocidos como bloques de *down-sampling* y *upsampling*, que para abreviar se suelen denominar bloques d y u respectivamente. Para los bloques de d en esta primera aproximación se ha utilizado una capa de convolución (página 6), seguida de la normalización de las salidas y una capa de activación *LeakyReLU* (página 9). Por su parte, las capas de los bloques u están compuestas por una convolución traspuesta, la normalización de las salidas de esta y la aplicación de una función ReLU (página 8). Ambos bloques pueden verse en la figura 3.3.

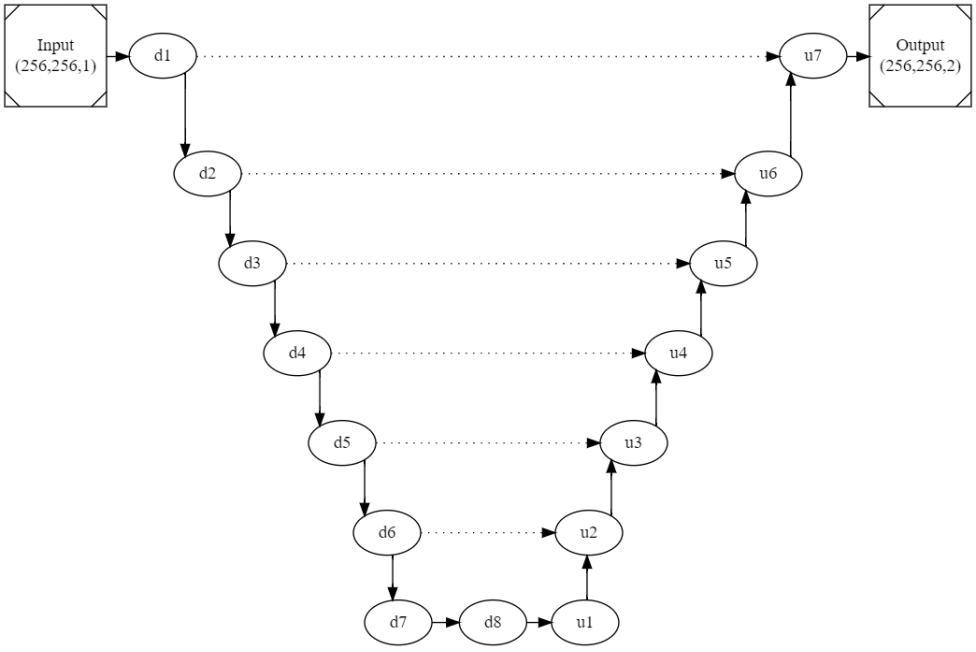


Figura 3.2: Arquitecura U-Net del generador

En la figura 3.2 se muestra la arquitectura básica del generador, según la base de la U-Net.

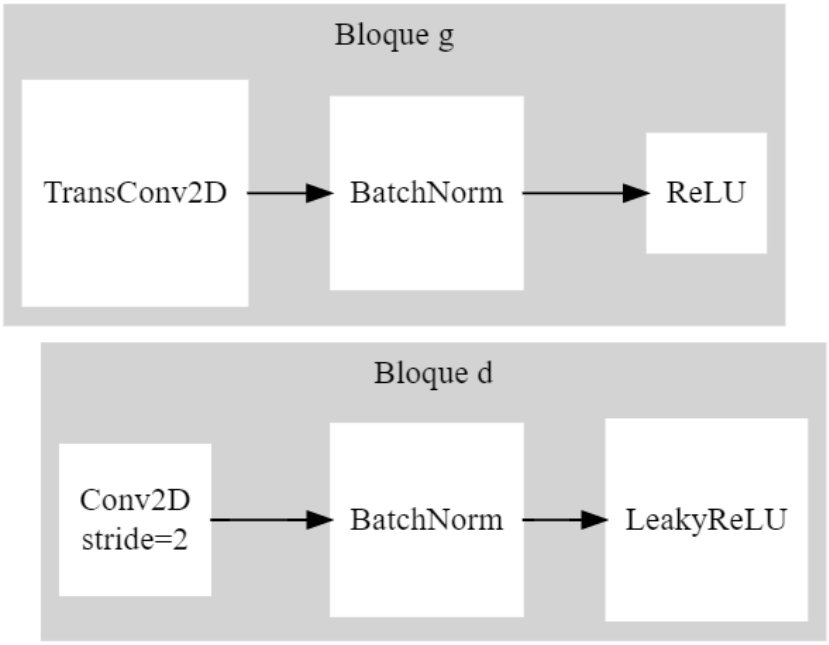


Figura 3.3: Bloques d y u de la arquitectura

Las líneas punteadas en la figura 3.2 representan las *skip-connections*, en las que se concatena la salida del bloque de *downsampling* con la de su simétrico de *upsampling*. El tamaño inicial de filtros es de 64, y se duplica en cada bloque de *downsampling*. De forma simétrica, el número de filtros se divide a la mitad en cada bloque de *upsampling*. La entrada es un ruido gaussiano de dimensiones (256,256,1), y la salida son los canales U y V que representan los colores de la imagen.

3.2.3 Pruebas

El generador en este momento tiene muy poca información para generar imágenes creíbles, ya que no dispone de momento de la imagen en blanco y negro, y tan solo utiliza un ruido aleatorio, y se basa completamente en la diferencia entre su salida y la salida esperada. Esto genera resultados poco realistas, como se puede apreciar en la figura 3.4.

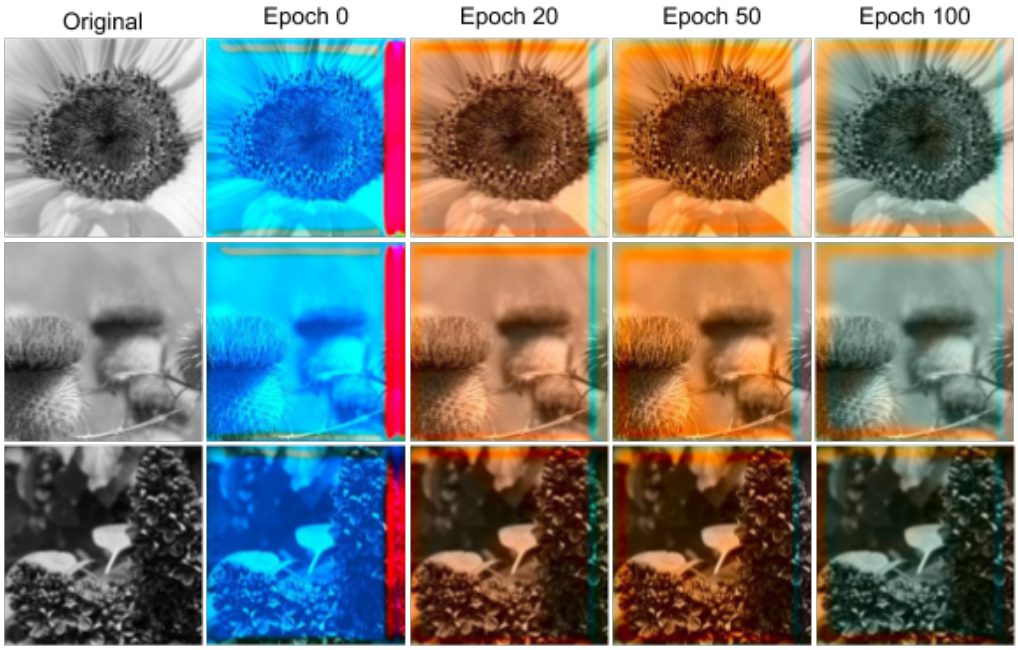


Figura 3.4: Evolución del entrenamiento del primer prototipo de generador.

3.3 Prototipo 2: Condicionante

3.3.1 Diseño

Como se ha visto en la sección 3.2, el modelo tarda en converger y no tiene suficiente información para generar imágenes realistas. Para paliar este problema, se ha utilizado la imagen original como condicionante (véase la explicación en la página 14).

3.3.2 Desarrollo

Partiendo de la arquitectura U-Net original, se ha añadido una entrada extra para actuar en forma de condicionante, en la que se cargará la imagen original en blanco y negro. Esta capa se concatena al vector latente para producir una entrada de dimensiones (W,H,2). En cada una de las capas ocultas del segmento de *downsampling* se ha utilizado la normalización de batch [33] para evitar el colapso durante el entrenamiento. Esto ha sido explicado con más detalle en la sección 3.7. Para el segmento de *upsampling* (figura 3.3), se sigue una estructura simétrica, aplicando *dropout* [34] a las primeras capas para evitar el *overfitting*. Las *skip-layers* se mantienen siguiendo la arquitectura U-Net original. La arquitectura del generador modificado, que permite el condicionamiento, se muestra en la figura 3.5

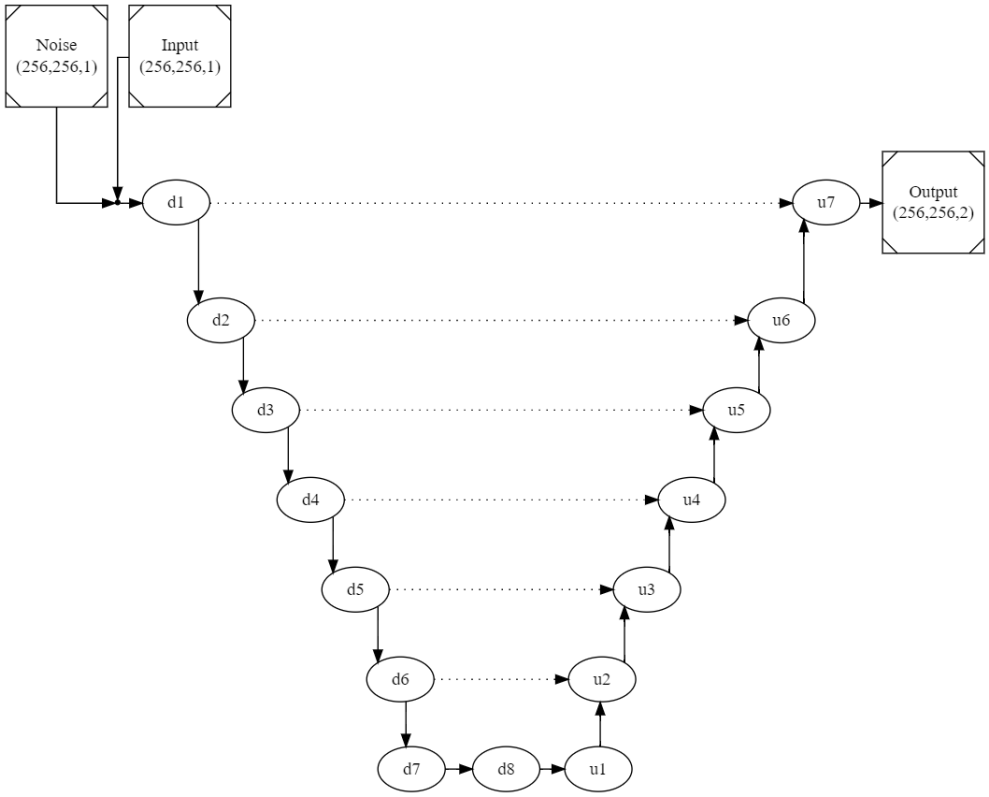


Figura 3.5: Arquitecura U-Net del generador condicionado.

Conv2D(filters=n,kernel_size=(3,3),strides=2)
BatchNormalization()
LeakyReLU()

Figura 3.6: Estructura de cada bloque **d** del discriminador.

3.3.3 Pruebas

La arquitectura es capaz de generar imágenes realistas y capturar detalles, por lo que se mantiene para ser utilizado en la estructura final. Para realizar el entrenamiento de prueba del discriminador, se generan 4000 imágenes del *dataset* procesadas con este generador, para utilizarlas como imágenes falsas. Como puede apreciarse en la figura 3.7, aunque se siguen manteniendo los tonos sepia de las arquitecturas clásicas, el ruido y los parches de color se han reducido significativamente.

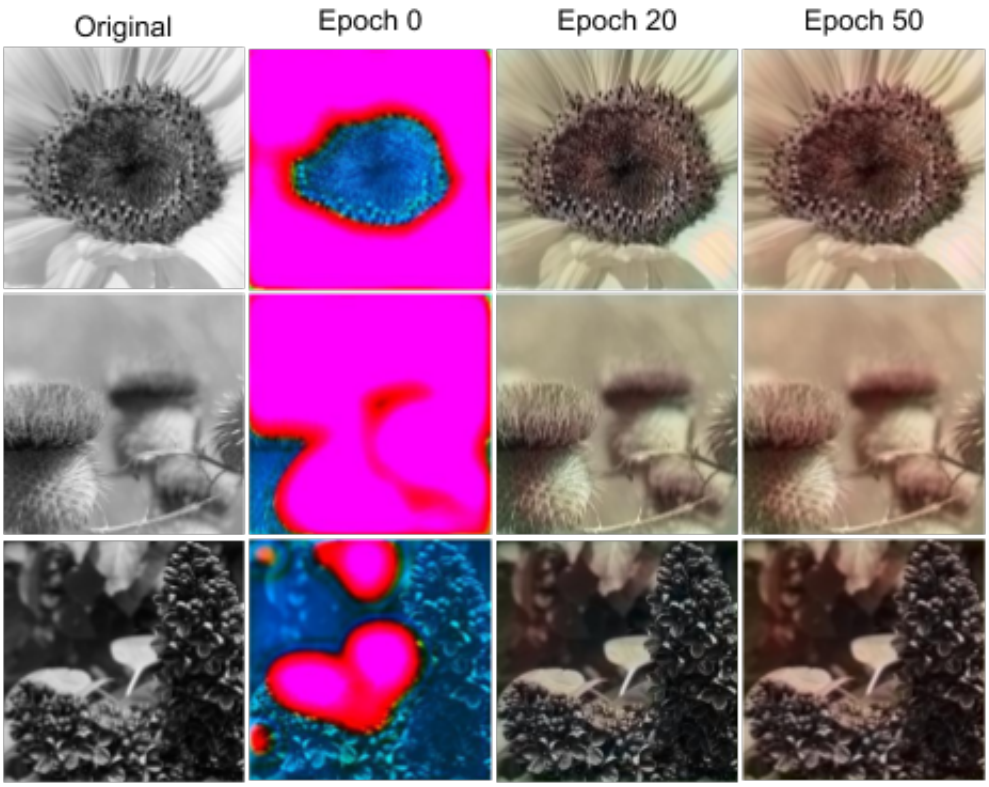


Figura 3.7: Evolución del entrenamiento del generador con condicionante.

3.4 Prototipo 3: Discriminador

Es en este caso en el que entra en juego la teoría de las redes *GAN* explicada en la sección 2.2.5. Este tercer prototipo incorpora ya el desarrollo de un clasificador, que compita con los generadores que ya han sido desarrollados. Esto permitirá clasificar si una imagen es falsa o real, donde esa competitividad hará que mejoren los resultados.

3.4.1 Diseño

Para el diseño del discriminador, se utiliza una estrategia similar al caso del generador. Viendo los resultados positivos del condicionamiento en el generador, se opta por implementar directamente esta estrategia en el discriminador.

3.4.2 Desarrollo

A la arquitectura base, la sección de discriminador de una *PatchGAN*, ya explicada en la sección 2.2.5, se le añade una nueva capa que actúa de condicionante. En esta capa se cargará

la imagen original en blanco y negro, y es concatenada a la entrada principal (dos canales, correspondientes a los componentes U y V de la imagen final), dando lugar a una entrada de dimensiones (W,H,3). La salida del discriminador consiste en un tensor de dimensiones (30,30,1), en el que cada uno de los elementos es la probabilidad de que el parche correspondiente sea real. Para comprobar el funcionamiento del discriminador, se utilizan imágenes reales, y las imágenes obtenidas del generador en la sección anterior. La arquitectura del discriminador se puede ver en la figura 3.8, donde cada uno de sus bloques **d** está compuesto por las capas mostradas en la tabla 3.6

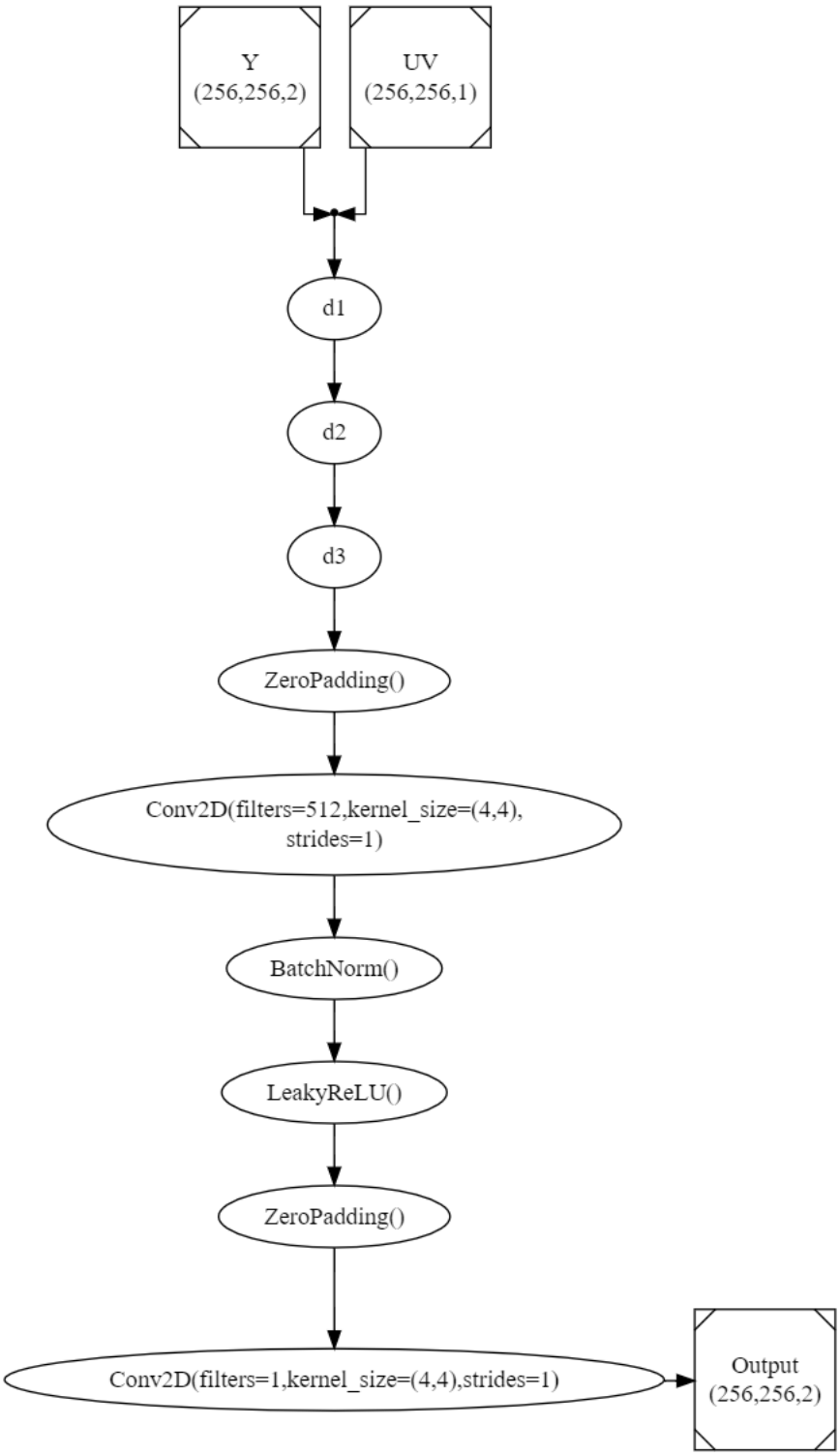


Figura 3.8: Arquitectura *PatchGAN* del discriminador condicionado.

Pruebas

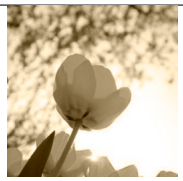
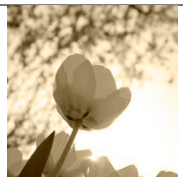
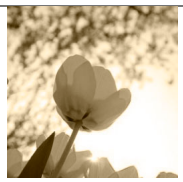
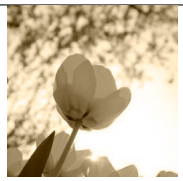
epoch	imagen real	resultado	imagen generada	resultado
0		Real		Real
10		Real		Real
20		Falsa		Falsa
50		Real		Falsa

Tabla 3.1: Evolución del rendimiento del discriminador para una imagen de prueba

Como se puede ver en la figura 3.1, el discriminador es capaz de distinguir entre las imágenes reales y las falsas, disminuyendo progresivamente su *loss*, lo cual indica que su proceso de aprendizaje funciona correctamente.

3.5 Prototipo 4: Integración de los modelos

Una vez desarrollados ambos modelos, deben comunicarse para dar lugar a una *GAN*. En esta sección se explica el diseño de dicha red y su algoritmo de entrenamiento.

3.5.1 Diseño

Una vez se ha comprobado la viabilidad de ambos submodelos, se integran en la arquitectura final. En esta, la salida del generador se acopla con la entrada del discriminador, y se utiliza un algoritmo modificado, que utiliza los gradientes del discriminador para actualizar el generador.

3.5.2 Desarrollo

En este punto, el algoritmo de entrenamiento cambia para utilizar el proceso de entrenamiento de una *GAN*, descrito a continuación. Este proceso se basa en el pseudocódigo mostrado en [2], que se puede consultar en la figura 3.9:

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

  for k steps **do**

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
  - Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

Figura 3.9: Algoritmo de entrenamiento de una *GAN* [2]

1. Se genera un *minibatch* de imágenes ”falsas” provenientes del generador.

2. Se extrae un *minibatch* de imágenes reales del *dataset*.

3. Se actualizan los pesos del discriminador por separado con las imágenes ”falsas” y reales. Este paso puede repetirse k veces por cada iteración de entrenamiento. Inicialmente el hiperparámetro k se ha fijado a 1.

4. A continuación, se genera un nuevo *minibatch* de imágenes falsas, que serán introducidas en el discriminador, con la particularidad de que se etiquetan como imágenes **reales**.

5. Con los gradientes del discriminador en este paso se actualizan los pesos del generador, que aprenderá a maximizar la probabilidad de que el discriminador clasifique su salida como ”real”.

En la figura 3.5.2 se muestra un pseudocódigo de cada paso de entrenamiento del modelo.

```
1
2  #Creación de una imagen falsa.
3  gen_output = generator([input_image,noise])
4  #Imágenes reales
5  disc_real = discriminator([input_image,noise,target])
6  #Imágenes generadas
7  disc_gen = discriminator([input_image,noise,gen_output])
8  #Loss del generador
9  g_loss = gen_loss(disc_gen,gen_output,target)
10 #Loss del discriminador
11 d_loss_fake,d_loss_real = disc_loss(disc_real,disc_gen)
12 #Se actualizan los pesos del generador
13 generator_weights.update(g_loss)
14 #Se actualizan los pesos del discriminador
15 discriminator_weights.update(d_loss_fake+d_loss_real)
16
17 return g_loss,d_loss_fake,d_loss_real
```

3.6 Prototipo 5: Modificación del entrenamiento

3.6.1 Desarrollo

En el entrenamiento original de una *GAN*, los pesos del generador se actualizan exclusivamente en base a la salida del discriminador. En el coloreado de imágenes, la localidad espacial es especialmente importante, ya que se requiere que las regiones de color coincidan perfectamente con las regiones de la imagen en blanco y negro. Debido a esto, se ha modificado la función objetivo del generador para incluir de forma ponderada las funciones de regularización L1 y L2. La función objetivo del generador queda indicada en la ecuación 3.1.

$$\max_{\theta_g} \left[\mathbb{E}_{z\sim p_z} log(D_{\theta_d}(G_{\theta_g}(z))) + \lambda_1 \ell^1 + \lambda_2 \ell^2\right] \tag{3.1}$$

En base a la exploración empírica, se han seleccionado los parámetros $\lambda_1 = 5; \lambda_2 = 5$.

3.6.2 Pruebas

3.7 Prototipo 6: Técnicas de estabilización del entrenamiento

Las *GAN* son extremadamente inestables y difíciles de entrenar, pero existen ciertas modificaciones que permiten facilitar y estabilizar el entrenamiento [35].

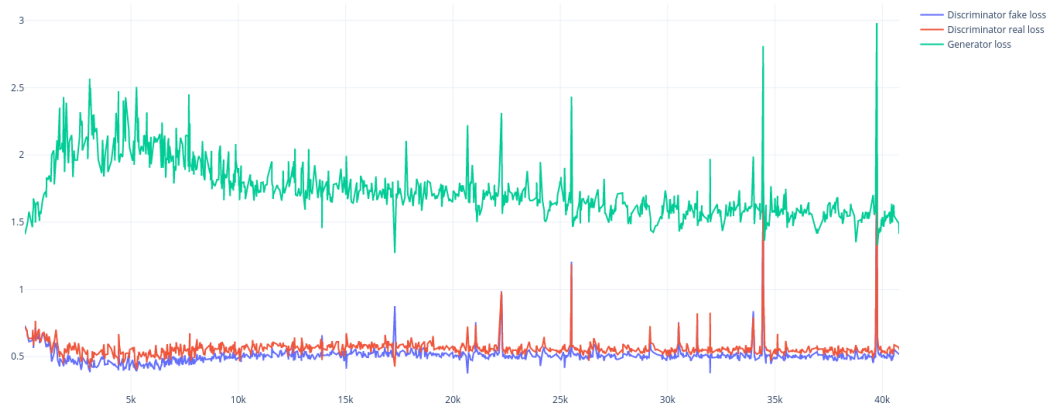


Figura 3.10: *Losses* a lo largo de los pasos de entrenamiento.

3.7.1 Label flipping

En una *GAN*, el generador ”aprende” basándose en los gradientes proporcionados por el discriminador. En las primeras iteraciones de entrenamiento, los gradientes del discriminador para una salida con *label*=0 pueden ser muy pequeños. Esto provoca desvanecimiento de gradientes, y evita que el generador aprenda correctamente, desequilibrando la balanza a favor del discriminador, lo cual puede provocar que se deje de producir un aprendizaje. Una técnica fácil de implementar para prevenir este problema consiste en ”invertir” las *labels*, haciendo que las imágenes reales tengan una *label* de 0 y las falsas una *label* de 1. Esto favorece el flujo de gradientes desde el discriminador hacia el generador, acelerando las etapas iniciales del entrenamiento.

3.7.2 *Instance noise*

En las *GAN*, el generador genera nuevas muestras basándose en un espacio latente, modelado como una distribución aleatoria. Debido a esto, en la salida del generador es común encontrar ruido proveniente del espacio latente. Durante el entrenamiento, el discriminador puede aprender a utilizar este ruido como “prueba” de que una entrada es falsa. Visto de otro modo, las distribuciones de las imágenes reales y falsas son muy concentradas, y normalmente no se solapan, lo que las hace fácilmente separables. En el artículo *Amortised MAP Inference for Image Super-resolution* [36] se propone añadir un ruido gaussiano a ambas distribuciones, de modo que se solapen más y el discriminador no tenga tan fácil el separarlas. Añadiendo a la entrada del generador un componente de ruido gaussiano, se palia este problema, y se fuerza al discriminador a abstraer otras características para distinguir entre imágenes reales

y falsas, estabilizando así el entrenamiento. [33]

3.8 Resultados del entrenamiento

Figura 3.11: Ejemplo del resultado a las 400 epochs.

En la figura 3.11 se muestra el resultado de la ejecución del entrenamiento en la epoch 400. En la columna izquierda se muestran de arriba a abajo los componentes de la imagen original, y en la columna derecha los componentes de la imagen generada. Como se puede apreciar, a pesar de que los colores no son iguales, se genera una imagen creíble. Esto es debido al uso de una *GAN*, que al contrario de las redes tradicionales, no utiliza una función de *loss* basada en la diferencia entre el *ground-truth* y la imagen generada, si no que se basa la pérdida del discriminador. Modificando el vector latente se puede variar el resultado del coloreado para una misma imagen, lo cual hace posible generar imágenes muy variadas.

Capítulo 4

Desarrollo de la herramienta

Este capítulo se dedica a describir la aplicación creada con el fin de facilitar el uso de los modelos desarrollados y descritos en el capítulo 3 de la presente memoria. El objetivo de la aplicación no es desarrollar un modelo final para producción, sino un ejemplo de cómo se podría empotrar un sistema inteligente con el fin de facilitar el desarrollo y como ambas partes pueden llevar desarrollos paralelos,pudiendo sustituirse cada una según evolucione.

4.1 Metodología

Pese a que en este trabajo se trata la herramienta como un solo módulo, el *backend* y el *frontend* son proyectos separados, que pueden ser desarrollados en paralelo, como ya se muestra en el diagrama de la página 21. Por lo tanto, se sigue una aproximación en la que los prototipos de ambos módulos se desarrollan en paralelo, aprovechando el tiempo en el que el modelo se está entrenando. Hasta que el modelo no esté plenamente entrenado, se utilizan diversas aproximaciones para evaluar y desarrollar los modelos.

4.2 Requisitos

En esta sección se pasan a describir los requisitos de la herramienta, tanto funcionales como no funcionales. Este es un paso clave en la metodología de prototipado, ya que ayuda a definir las iteraciones y evaluar el resultado de cada prototipo.

4.2.1 Requisitos funcionales

- El usuario debe poder acceder a la herramienta web desde cualquier dispositivo con navegador web.
- El usuario debe poder cargar una o varias imágenes desde la herramienta web.

- La herramienta debe ser capaz de transmitir dichas imágenes al *backend*, y recibir el resultado del coloreado.

- En caso de fallo en el *backend*, la herramienta debe devolver al usuario información sobre el error, y mantenerse operativa.

4.2.2 Requisitos no funcionales

- La aplicación debe ser modular. Es decir, el *frontend* debe ser independiente del *backend*, y poder operar de manera desacoplada.

- La aplicación debe ser escalable, pudiéndose replicar en varias máquinas de manera sencilla.

- La aplicación debe ser tolerante a fallos. Si uno de los componentes deja de funcionar, debe monitorizarse y relanzarse, con el mínimo *downtime* para el usuario.

4.3 Prototipo 1:Diseño del *backend*

4.3.1 Desarrollo

El *backend* para la herramienta de coloreado es relativamente simple, ya que como se ha dicho, no nace con la idea de ser un producto final sino más bien una mera demostración. Es por ello que cuestiones como la autenticación o la seguridad no se han tenido en cuenta como si se tratase de una aplicación que fuese a ser puesta en producción. Aun así, se ha buscado que el diseño de la aplicación sea lo más general posible, con el fin de extenderla y contener nuevas funcionalidades si así llegase el caso. Con esos principios en mente, la aplicación expone simplemente un *endpoint* REST, en el que se reciben las imágenes en base64.

Analizando el funcionamiento deseado por parte de la aplicación, el flujo quedaría como se muestra en la figura 4.1. Según esta, al iniciar el servidor Flask, este se encarga de cargar el modelo disponible en memoria, permaneciendo a la espera de recibir una petición con una imagen como parámetro. Al recibir una petición, se recupera la imagen pasada como parámetro, que debe decodificarse de la base64. Una vez la imagen ya está en el formato adecuado, se introduce como entrada al modelo junto con un ruido generado aleatoriamente, que actúa como vector latente. Es entonces cuando el modelo contenido en el módulo de Inteligencia Artificial procesará la imagen, y devolverá el resultado del coloreado. Dicho resultado debe de codificarse nuevamente en base64 y se devuelve como resultado de la petición al API, para ser mostrado por el *frontend*.

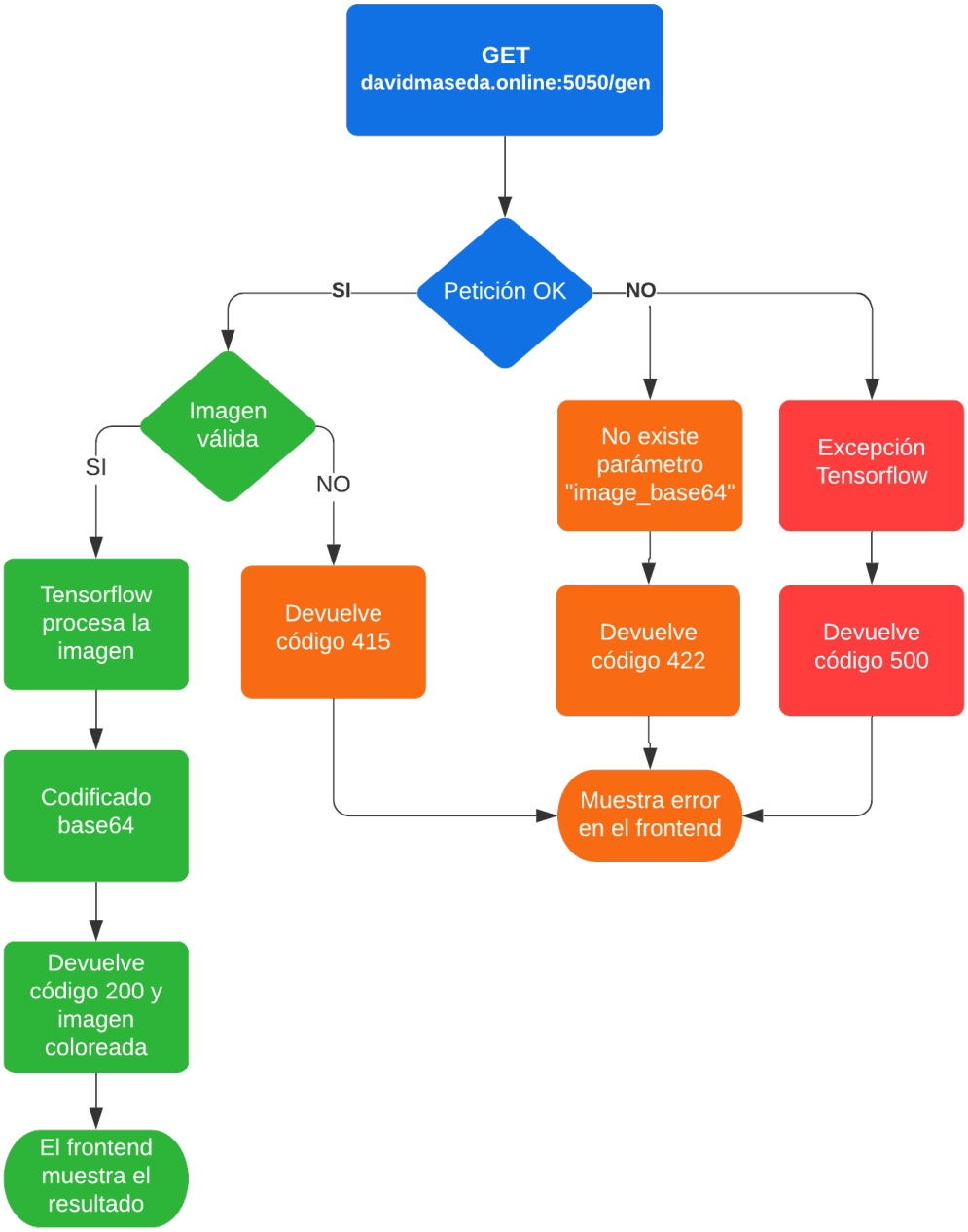


Figura 4.1: Diagrama de flujo de la API

4.3.2 Pruebas

Al no estar disponible el modelo, se desarrollan las pruebas del *endpoint* REST de manera independiente a la inteligencia artificial. Al llegar una petición, se comprueba su integridad y

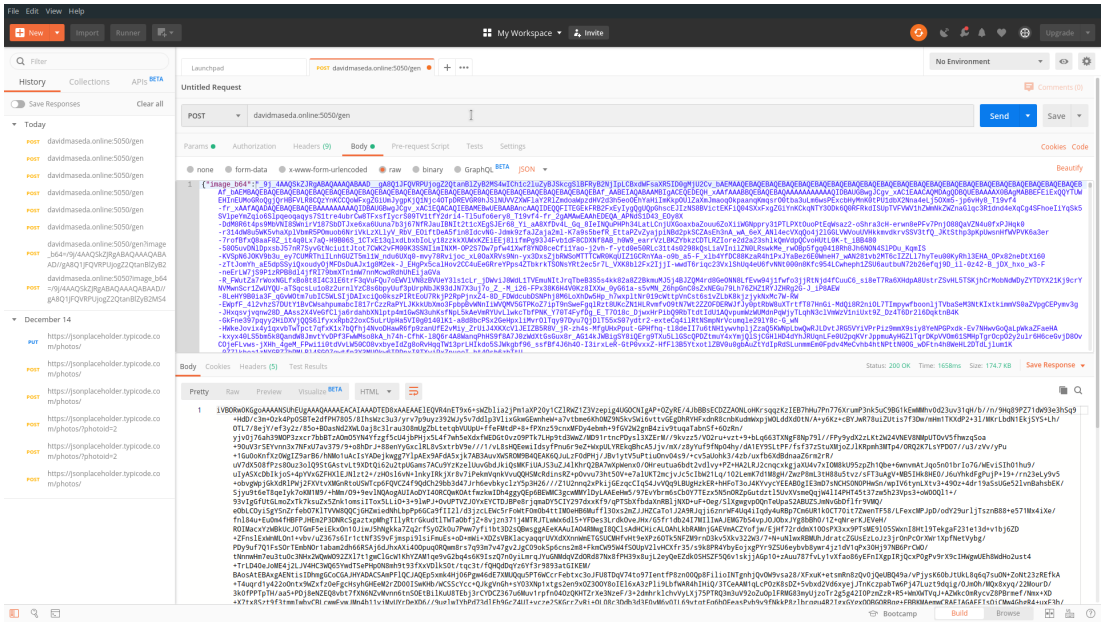


Figura 4.2: Petición POST para probar el *backend*

validez, y se decodifica la imagen de base64. Una vez decodificada, se comprueba que tiene un formato válido, y se vuelve a codificar para mandarla como respuesta. De este modo, aunque la respuesta es igual a la petición, se puede comprobar que el *backend* recibe correctamente peticiones POST, y devuelve una respuesta apropiada. Al no estar desarrollado todavía el primer prototipo de *frontend*, las peticiones se hacen de manera manual utilizando un cliente REST, como **Postman** [37]. Una de estas peticiones puede ser observada en la figura 4.2.

4.4 Prototipo 2:Diseño del *frontend*

4.4.1 Desarrollo

En este caso, debido a la naturaleza de la herramienta, el *frontend* de la misma podría calificarse como sobrio o sencillo. Este consta de un área donde se permitirá adjuntar las imágenes que se desean colorear. Dicha área permite la inserción tanto mediante *drag and drop* o un diálogo de selección de archivos.

Una vez cargada la imagen, esta se codifica en base64 y se emite una petición asíncrona a la API con estos datos de cada una de las imágenes adjuntadas, quedando a la espera de las respuestas.

Una vez llegan las respuestas de las peticiones, se decodifica el resultado para extraer la imagen y se muestra el resultado junto a la imagen original.

Para hacer más fácil el proceso de evaluación manual, la herramienta permite introducir

imágenes a color, que serán transformadas a blanco y negro, y coloreadas posteriormente, ofreciendo así una comparativa entre la imagen original y la generada por el modelo. En el futuro, esta herramienta puede expandirse para permitir opciones como la selección del modelo, o realizar ajustes automáticos o manuales a la imagen generada.

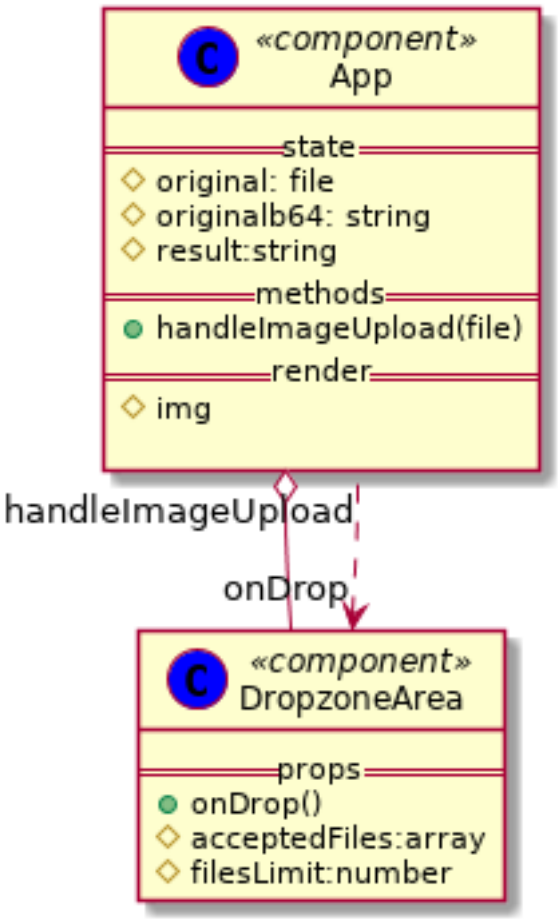


Figura 4.3: Diagrama de clases del *frontend*.

4.4.2 Pruebas

En este momento el *backend* ya está parcialmente desarrollado, y se puede probar realizándole peticiones. El resultado de las pruebas es satisfactorio, ya que se puede observar fácilmente que la imagen resultante es idéntica a la imagen original, por lo que se asegura que la comunicación entre el *backend* y el *frontend* es correcta.

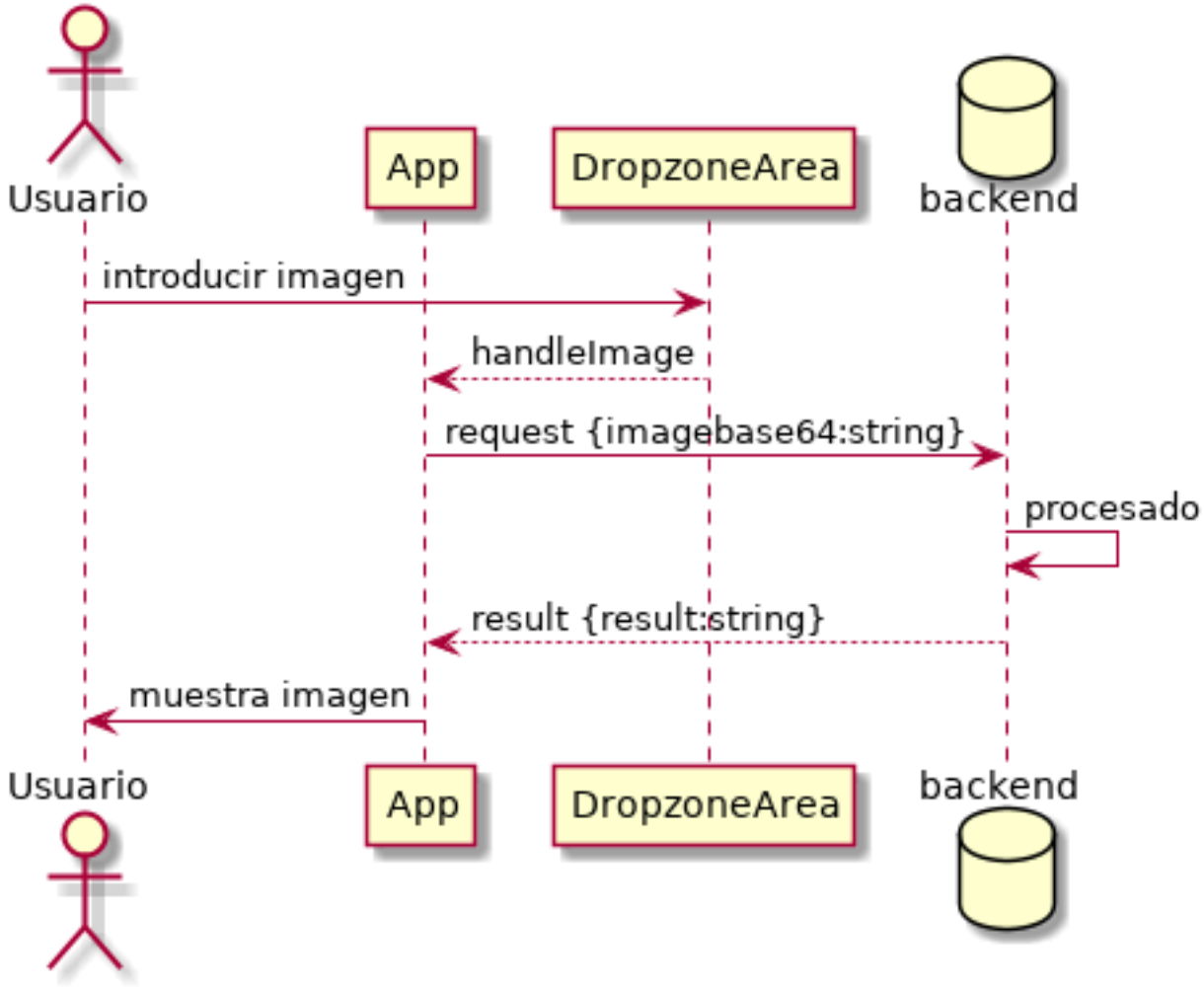


Figura 4.4: Diagrama de secuencia del *frontend*.

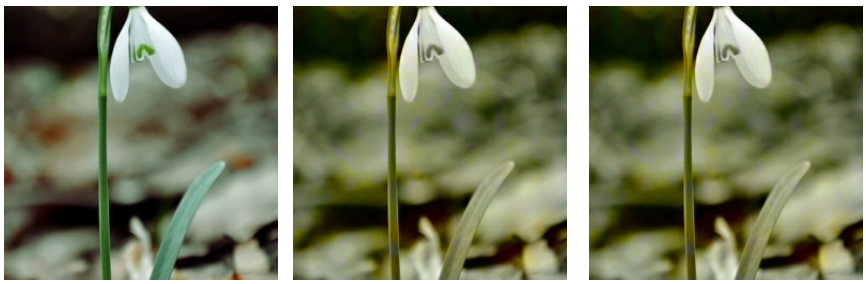
4.5 Prototipo 3: Integración del modelo

4.5.1 Desarrollo

Una vez un modelo de coloreado está entrenado, se puede integrar en el *backend* desarrollado con Flask. Al estar desarrollado en Python, la integración en sumamente sencilla, ya que una vez decodificada la imagen, se puede mandar al modelo (previamente cargado al iniciar la aplicación) y recibir la imagen resultante, que posteriormente será codificada y enviada como respuesta al *frontend*.

4.5.2 Pruebas

Para probar la integración del modelo en la herramienta, se desarrolla un pequeño *script* en Python que carga el modelo y permite colorear una imagen desde la línea de comandos. El resultado de este coloreado debe ser idéntico al resultante de utilizar la herramienta web. Se corrobora que las pruebas son satisfactorias (como se puede comprobar en la figura 4.5), por lo que se verifica el prototipo como válido.



(a) Imagen original (b) Imagen coloreada por la herramienta (c) Imagen coloreada por el modelo

Figura 4.5: Verificación de que la herramienta y el modelo en bruto devuelven el mismo resultado

4.6 Prototipo 4: Implementación en microservicios

Para facilitar el despliegue de la herramienta, se ha optado por integrar cada uno de los módulos en Docker [31]. Esto permite desplegar la herramienta en cualquier máquina con Docker instalado, de una manera sencilla. Con esta aproximación, el escalado horizontal es rápido y fácil, y puede integrarse con herramientas como **Kubernetes** [38] o **Docker Swarm**.

4.6.1 Desarrollo

En cada uno de los submódulos de la herramienta se ha añadido un **Dockerfile**, un archivo de definición que contiene los pasos a seguir para crear una imagen con el submódulo. Un ejemplo de Dockerfile se muestra en la figura 4.6

```
# fetch basic image
FROM maven:3.3.9-jdk-8

# application placed into /opt/app
RUN mkdir -p /opt/app
WORKDIR /opt/app

# selectively add the POM file and
# install dependencies
COPY pom.xml /opt/app/
RUN mvn install

# rest of the project
COPY src /opt/app/src
RUN mvn package

# local application port
EXPOSE 8080

# execute it
CMD ["mvn", "exec:java"]
```

Figura 4.6: Ejemplo de DockerFile

El contenido de estos archivos se especifica en el anexo A, ya que la explicación de la infraestructura de Docker queda fuera del ámbito de este trabajo.

Además, al haber desarrollado cada uno de los submódulos en un repositorio **git** diferente, puede automatizarse la tarea de crear estas imágenes, utilizando **Docker Hub**, un repositorio de imágenes listas para producción. Se ha enlazado cada repositorio con su correspondiente imagen, por lo que cada vez que se hace *push* a la rama *master* del repositorio, se lanza un proceso de construcción de la imagen en Docker Hub.

Capítulo 5

Conclusiones y Futuros Desarrollos

Este capítulo pretende realizar un compendio de las conclusiones que se pueden sacar tras el desarrollo del proyecto, así como dibujar las posibles líneas que un futuro se podrían desenvolver a partir del trabajo realizado.

5.1 Conclusiones

Lo primero que se debe de destacar, a tenor de los objetivos marcados al comienzo del proyecto, que se pueden ver en la sección 1.3, es que todos sin excepción se han cumplido en un grado aceptable. Si bien es cierto que tanto el modelo de coloreado como la herramienta tienen margen de mejora, no es menos cierto que el objetivo del proyecto no era tanto ofrecer un resultado perfecto, como explorar las posibilidades de este conjunto de nuevas técnicas para ofrecer una solución a un problema de actualidad.

Para poder utilizar esta herramienta en producción de manera satisfactoria, se requiere un *dataset* mucho más amplio. En el desarrollo de este trabajo, con propósito de investigación y por tratarse de un reto particularmente difícil, se han utilizado solamente flores. Sin embargo, el modelo es potencialmente capaz de colorear diversos tipos de imagen con resultados satisfactorios si es entrenado más tiempo y con más datos.

Como ya se mencionó anteriormente, todos los objetivos se han cumplido en un grado satisfactorio.

La mayor dificultad en el desarrollo de este trabajo ha sido la enorme complejidad de los modelos *GAN*, tanto en coste computacional como en complejidad de entrenamiento. Así mismo, al ser una arquitectura reciente (surge inicialmente en 2014), es un área muy activa de investigación, con constantes cambios y nuevas aproximaciones.

Desde el inicio de la investigación para el trabajo, se han publicado numerosos artículos relativos a las *GAN*, y se han desarrollado nuevos métodos y algoritmos de entrenamiento que mejoran los resultados y los hacen más estables. Debido a esto, a lo largo del desarrollo

del proyecto, el alumno ha debido adaptarse y experimentar con el nuevo estado del arte, suponiendo un reto, pero aportando la experiencia de trabajar en un campo en constante desarrollo y con mucho futuro.

Así mismo se quiere destacar la puesta en valor de las competencias adquiridas durante el grado que se ven reflejadas en este trabajo. Si bien son muchas las aplicadas, es cierto que determinadas asignaturas, especialmente de mención, han tenido un peso muy importante en el desarrollo de este trabajo. En particular es necesario mencionar las asignaturas de "Proceso Software", "Sistemas Inteligentes", "Interfaces Persona Máquina", "Aprendizaje Automático", "Visión Artificial"y "Computación Gráfica y Visualización". De esta manera se ha podido poner en práctica gran parte de lo aprendido. Además, se valora especialmente la adquisición de otras competencias como, por ejemplo, la B1 (Capacidad de resolución de problemas) o la C7 (Asumir como profesional y ciudadano la importancia del aprendizaje a lo largo de la vida), ya que el alumno ha tenido que formarse en varias de las últimas tendencias en Sistemas Inteligentes y librerías utilizadas en el sector privado que no están contempladas dentro del Grado en Ingeniería en Informática.

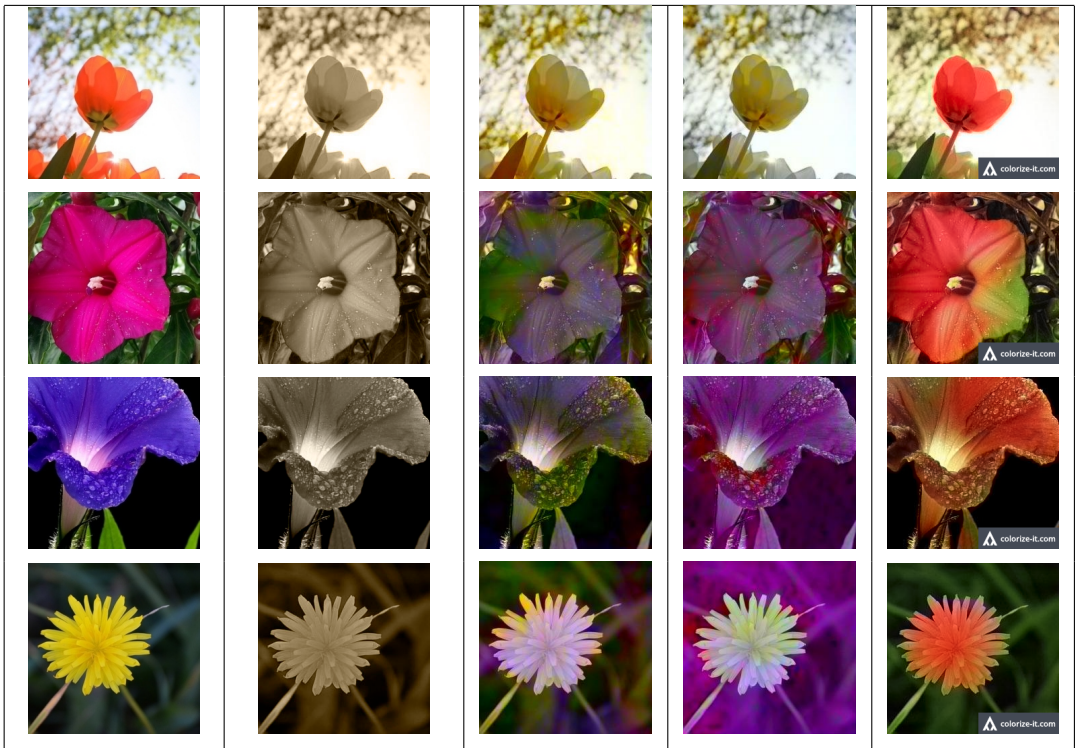


Tabla 5.1: Comparativa de resultados entre varias aproximaciones.

En la tabla 5.1, se muestran los resultados para varias arquitecturas desarrolladas en este

proyecto, y la aproximación *state of the art*, basada en el artículo [39]. Como se puede comprobar, los generadores clásicos no son viables para este tipo de problema, pero los basados en *Generative Adversarial Networks* proveen un resultado mucho más diverso. El parámetro λ mostrado en la figura anterior es el peso que se le asigna a las funciones L1 y L2 dentro de la *loss* del generador, como ya se ha explicado en la sección 3.3.

5.1.1 Hardware utilizado

Un punto que se debe destacar es que si bien ya se sabía que las redes neuronales profundas conllevan una enorme cantidad de cálculos, y que la potencia del *hardware* utilizado es esencial para entrenar un modelo de manera eficiente, este punto se ve acrecentado exponencialmente con el uso de redes tipo *GAN*.

En un primer momento, para el desarrollo y prototipado inicial se ha utilizado un portátil, con las siguientes especificaciones:

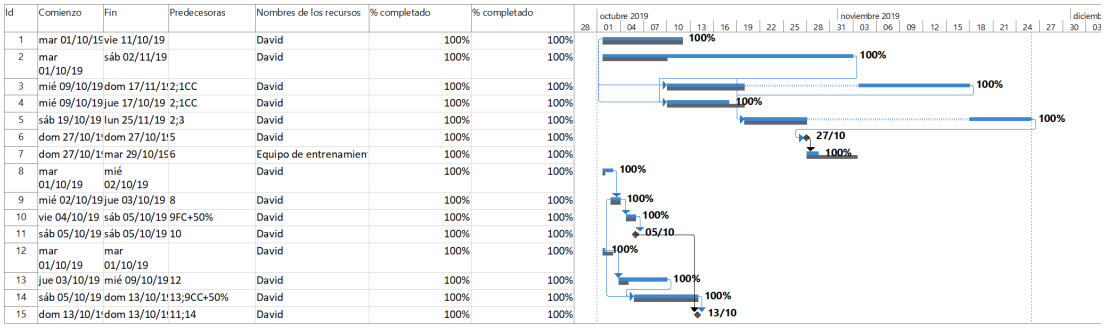
- NVIDIA GTX 1060 Max-Q/6GB
- Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz, 4 cores,8 threads.
- 16GB RAM

Una vez realizado el prototipado, y diseñado el modelo completo, las especificaciones técnicas de este equipo resultaron insuficientes, ya que el modelo ocupaba la práctica totalidad de la memoria de la GPU, y el tamaño de batch no podía ser superior a 4 imágenes. Debido a esto, se optó por el uso de una instancia p3.2xlarge de los Amazon Web Services, con las siguientes especificaciones:

- NVIDIA V100/16GB
- 8 vCores Intel Xeon E5-2686 v4 @ 2.70 GHz
- 64GB RAM

El uso de una instancia de Amazon redujo el tiempo por *epoch* en el entrenamiento final de 700 segundos a 60 segundos, aproximadamente. Lo que marca la necesidad de disponer de un soporte hardware adecuado si se quieren acometer desarrollos de mayor calado que el que aquí se expone.

5.1.2 Estadísticas finales del proyecto



Id	Comienzo	Fin	Predecesoras	Nombres de los recursos	% completado	% completado
1	mar 01/10/19	vie 11/10/19		David	100%	100%
2	mar 01/10/19	sáb 02/11/19		David	100%	100%
3	mié 09/10/19	dom 17/11/19	2;1CC	David	100%	100%
4	mié 09/10/19	jue 17/10/19	2;1CC	David	100%	100%
5	sáb 19/10/19	lun 25/11/19	2;3	David	100%	100%
6	dom 27/10/19	dom 27/10/19	5	David	100%	100%
7	dom 27/10/19	mar 29/10/19	6	Equipo de entrenamien	100%	100%
8	mar 01/10/19	mié 02/10/19		David	100%	100%
9	mié 02/10/19	jue 03/10/19	8	David	100%	100%
10	vie 04/10/19	sáb 05/10/19	9FC+50%	David	100%	100%
11	sáb 05/10/19	sáb 05/10/19	10	David	100%	100%
12	mar 01/10/19	mar 01/10/19		David	100%	100%
13	jue 03/10/19	mié 09/10/19	12	David	100%	100%
14	sáb 05/10/19	dom 13/10/19	13;9CC+50%	David	100%	100%
15	dom 13/10/19	dom 13/10/19	11;14	David	100%	100%

Figura 5.1: Seguimiento al 100% del proyecto.

	Comienzo	Fin
Actual	mar 01/10/19	lun 25/11/19
Previsto	mar 01/10/19	mar 12/11/19
Real	mar 01/10/19	lun 25/11/19
Variación	0d	9d

	Duración	Trabajo	Costo
Actual	39,13d?	644h	12.202,00 €
Previsto	30,13d	588h	0,00 €
Real	39,13d	644h	12.202,00 €
Restante	0d?	0h	0,00 €

Porcentaje completado:
Duración: 100% Trabajo: 100%
Cerrar

Figura 5.2: Estadísticas al final del proyecto.

Como se puede ver en la figura 5.2, el proyecto se ha desviado en tiempo y coste. La desviación en tiempo se ha debido a los errores en la estimación de tiempo para las tareas. En concreto, la tarea de ”Estudio de la implementación en Tensorflow” se ha desviado notablemente, debido a la migración de todo el proyecto desde la versión 1 a la 2 de dicha herramienta, que ofrecía un acceso a más bajo nivel del proceso de creación del modelo. Esto ha permitido desarrollar un modelo de manera más eficiente, pero ha supuesto la reescritura de gran parte del código original.

La desviación en coste es el resultado del alquiler de un servidor de Amazon para entrenar el modelo, como ya se ha citado en la sección 5.1.1. Esto ha supuesto un coste añadido, pero ha permitido el entrenamiento del modelo durante un número mayor de iteraciones, lo cual no hubiera sido posible con el *hardware* original.

5.2 Desarrollo futuro

En esta sección se definirán posibles vías y aproximaciones a seguir para el futuro desarrollo de este proyecto, con intención tanto de mejorar su desempeño como de añadir más funcionalidades de cara al usuario final.

5.2.1 Integración de un clasificador

El modelo desarrollado en este trabajo se basa únicamente en la imagen en blanco y negro para generar un coloreado. Esta aproximación, pese a ser válida, puede ser mejorada teniendo datos contextuales de la imagen a colorear. Como ejemplo, la fotografía de un perro no tiene los mismos colores que una fotografía en primer plano de una flor, pese a poder contener los mismos niveles de luminosidad blanco y negro. Si el sistema conociera la información del objeto o entorno que está coloreando, podría aplicar diferentes alternativas o ”elegir” diferentes colores que se aproximaran más a la imagen real. Una aproximación viable para incorporar esta aproximación es utilizar un clasificador como etapa previa. En este flujo de trabajo, la imagen en blanco y negro sería inicialmente procesada por un clasificador, que le asignaría una clase (perro, retrato, flor...), y esta información se propagaría tanto al generador como al discriminador, actuando como un segundo condicionante. Existen dos alternativas principales para implementar esta funcionalidad:

1. Utilizar un clasificador pre-entrenado, como **Inception Resnet V2** [40] o **YOLO** [41]. Estos modelos podrían ser utilizados directamente o refinados para adaptarlos al problema en concreto (proceso conocido como *fine tuning* [42]).

2. Desarrollar y entrenar un clasificador propio.

El principal problema en ambas opciones es que tener una segunda red para procesar las imágenes aumentaría el peso del modelo, por lo que afectaría tanto al rendimiento de coloreado como al tiempo y cantidad de GPU para el entrenamiento.

5.2.2 Exploración exhaustiva del espacio de hiperparámetros

Durante el desarrollo de un modelo complejo, existen multitud de variables que pueden afectar al comportamiento del sistema (por ejemplo, *learning rate* del generador y el discriminador, parámetros λ, número de filtros en cada capa, número de capas, iteraciones del discriminador por cada iteración del modelo completo...). Aunque durante el desarrollo de este trabajo se han optimizado algunos de estos parámetros, la cantidad de combinaciones posibles entre ellos es inmensa, quedando su desarrollo a expensas de un mayor ámbito temporal y recursos computacionales. Existen diversas alternativas para explorar exhaustivamente las

combinaciones de hiperparámetros, siendo una de las más populares la opción **HParams** de Tensorboard. Adaptando el modelo para utilizar HParams, se pueden comparar los resultados mediante la interfaz gráfica que provee Tensorboard, utilizando diversas estrategias (exploración por orden, exploración aleatoria...).

5.2.3 Diseño de una métrica de evaluación objetiva

Debido a la cantidad de resultados aceptables de colores para una imagen en blanco y negro, las métricas habituales para evaluar los resultados, como el error cuadrático medio (MSE) [43] o el SSIM [9] pueden no ser aptas.

Como se puede observar en la figura 5.3, el MSE es muy diferente entre dos coloreados que a primera vista podrían ser válidos.

(a) MSE 1 (b) MSE 2

Figura 5.3: MSE

De igual manera, otras métricas más avanzadas y orientadas a imágenes como el SSIM tampoco ofrecen una alternativa completa para evaluar el desempeño del modelo, como puede ser observado en la figura 5.4.

Durante el desarrollo del trabajo, se ha observado manualmente la evolución del entrenamiento. Sin embargo, para automatizar ciertas tareas, como la exploración del espacio de hiperparámetros, ya mencionada en la sección 5.2.2, sería necesario diseñar una función que permita evaluar de manera automática y objetiva el resultado del coloreado, teniendo en cuenta la cantidad de resultados válidos y diversos.

(a) SSIM 1 (b) SSIM 2

Figura 5.4: SSIM

5.2.4 Mejora y nuevas funcionalidades de la herramienta

Aunque el objetivo principal de este trabajo ha sido el diseño y evaluación de un modelo para el coloreado de imágenes, existen muchas posibles mejoras aplicables a la herramienta que actúa como interfaz de cara al usuario. Se listan a continuación algunas de ellas:

- Introducción de una *seed* para producir resultados deterministas y ”guardar” coloreados.
- Introducción de un campo para elegir el ruido, que actúa de vector latente, manualmente.
- Integración de funcionalidades para modificar la imagen, como curvas, niveles, edición y mejora del histograma...
- Mejora de la experiencia de usuario. Por ejemplo, hacer que las modificaciones anteriores en la imagen se reflejen automáticamente en el resultado del coloreado.
- Posibilidad de exportar la imagen en varios formatos.

Así mismo, aunque no es una funcionalidad en si misma, evidentemente se contempla el contacto con especialistas de la restauración de imágenes. Estos, conocidos como coloristas, pueden otorgar un *feedback* extremadamente valioso. Sus evaluaciones de este proyecto y su opinión sobre funcionalidades alternativas pueden hacer evolucionar este trabajo de un mero prototipo a una herramienta dentro de su flujo de trabajo.

Apéndices

Apéndice A

Instrucciones de despliegue

En este anexo se realizará un *walktrough* del despliegue de la herramienta, que ha sido adaptada para formar un conjunto de microservicios.

A.1 Requisitos

En esta sección se especifican los requisitos mínimos que debe tener la máquina para poder realizar el despliegue:

- Sistema Linux,OSX o Windows.
- Docker y docker-compose instalados.
- Puerto 8080 accesible y, en caso necesario, redirigido.

A.2 Dockerfiles

Cada uno de los servicios puede ser lanzado mediante Docker de manera independiente. Sin embargo, se ha desarrollado un pequeño repositorio, que incluye *frontend* y *backend* como *git submodules*. Así mismo, incluye los *scripts* **build.sh**, **start.sh** y **stop.sh**, que permiten desplegar el sistema con muy poco esfuerzo.

El primer paso es clonar el repositorio, e inicializar sus submódulos:

```
1 git clone https://github.com/wizenink/colorful-all.git
2 cd colorful-all
3 git submodule init
4 git submodule update --remote
```

En este punto, todos los archivos necesarios para desplegar el sistema están listos.

Para construir las imágenes, se puede utilizar el script **build.sh**.

Una vez construídas, se pueden lanzar con el script start

```
./start.sh {endpoint}
```

donde *endpoint* se sustituye por el dominio que contendrá la aplicación.

Una vez realizado este paso, se puede comprobar que ambos contenedores están ejecutándose utilizando la interfaz de comandos de Docker

```
docker container ps
```

Para detener los contenedores, se utilizará el script **stop.sh**

A mayores, se ha escrito un pequeño archivo de definición de **docker-compose**, que permite desplegar las imágenes simplemente con el siguiente comando,ejecutado en el directorio donde se encuentra el archivo:

```
docker-compose up
```

Apéndice B

Manual de uso

En este capítulo, se hará una breve descripción del proceso que debe seguir el usuario final para utilizar la herramienta. El proceso es sumamente sencillo para el usuario final. La interfaz web consta de tres bloques básicos:

- Un listado de modelos, que permite al usuario seleccionar la arquitectura a utilizar para realizar el coloreado. Esto permite comparar los resultados entre varios modelos, y obtener un mejor procesado, dependiendo de la imagen a colorear.

- Una zona donde se pueden introducir imágenes, tanto mediante un diálogo de subida de archivos, como haciendo *drag and drop*.

- Un botón de **colorear**, que se activará solamente en el caso de haber introducido imágenes en la zona mencionada anteriormente.

- Una zona de resultados, en la que se mostrarán las imágenes una vez coloreadas.

Este proceso se realiza de manera asíncrona, por lo que el usuario puede subir un máximo de 10 imágenes, que serán procesadas por el *backend* y devueltas como resultado, sin bloquear la interfaz de usuario.

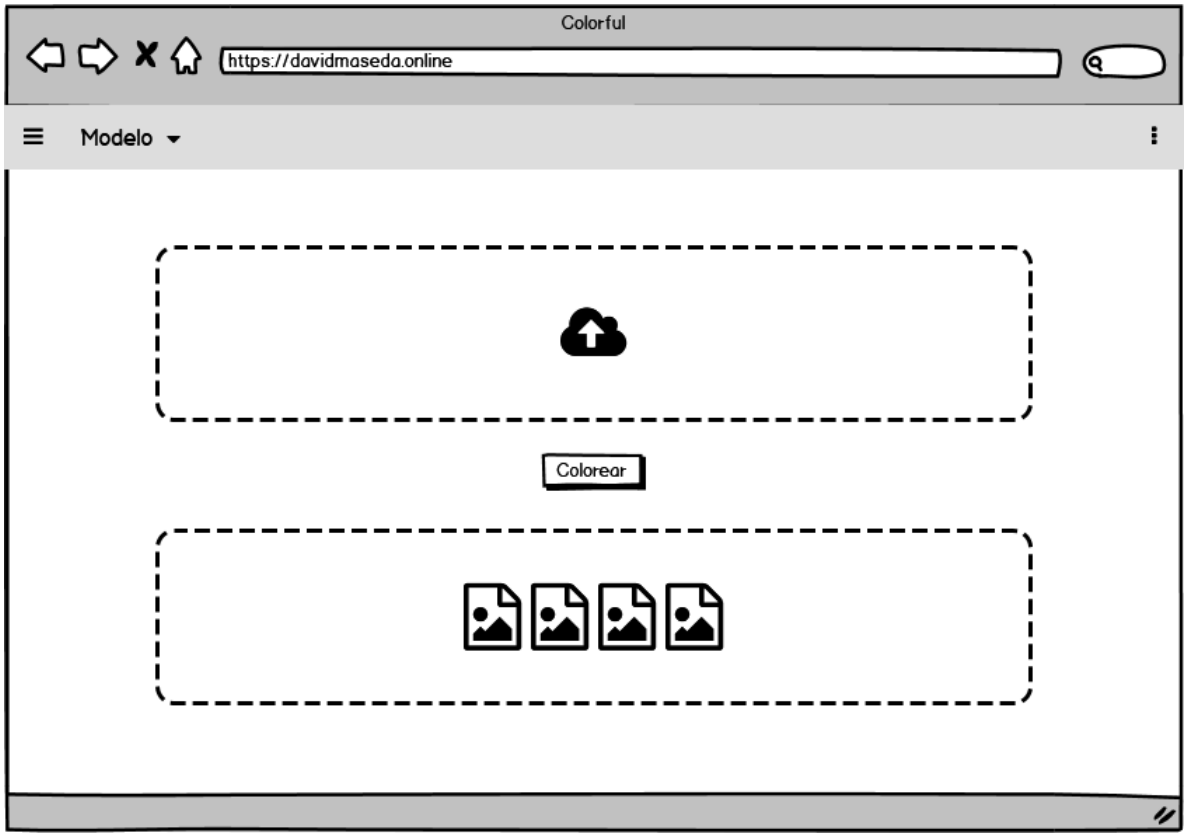


Figura B.1: Mockup de la interfaz web de la herramienta

Este proceso se puede comprobar mediante el uso de una *demo*, desplegada en el siguiente enlace: http://davidmaseda.online:8080

Lista de acrónimos

ASCII *American Standard Code for Information Interchange.*

API *Application Platform Interface.*

CGAN *Conditional GAN.*

CNN *Convolutional Neural Network.*

CRUD *Create-Read-Update-Delete.*

DCGAN *Deep Convolutional GAN.*

GAN *Genarative Adversarial Network.*

GPU *Graphical Processing Unit.*

MSE *Mean Squared Error.*

MVC *Model View ControlController.*

PatchGAN *Patched GAN.*

RGB *Red-Green-Blue.*

SAE *Stacked Auto Encoder.*

SPA *Single Page Application.*

SSIM *Structural Similarity Index.*

URI *Uniform Resource Identifier.*

VAE *Variational Auto Encoder.*

YUV *Luminance-Bandwidth-Chrominance.*

Glosario

Backend En arquitectura de software, el componente que actúa como base lógica y de almacenamiento de datos, y se comunica con el usuario a través del *frontend*

base64 Sistema de numeración que utiliza 64 como base. Permite representar datos binarios en formato ASCII.

Convolución Operación matemática que implica la aplicación de un filtro o *kernel* mediante la multiplicación de este a una señal, desplazando el *kernel*.

Equilibrio de Nash Principio de teoría de juegos que se aplica en la definición de las *GAN* y que se puede consultar en detalle en la página 5.

Frontend Capa que actúa de interfaz de cara al usuario y abstrae el funcionamiento en bajo nivel de un sistema de software.

L1 Función de normalización estadística, que minimiza la suma del valor absoluto de las diferencias entre el valor observado y el esperado.

L2 Función similar a la L1, pero minimizando el sumatorio del cuadrado de las diferencias entre el valor observado y el esperado.

Loss Función utilizada en optimización matemática y *deep learning*, que transforma una o varias variables en un número que representa el ”coste” de la solución. Normalmente, el objetivo es minimizar la función de *loss*.

Reinforcement learning Campo del aprendizaje automático que modela el aprendizaje como un agente que realiza acciones en un entorno para maximizar una recompensa acumulativa y evitar una penalización

REST principio de arquitectura de software que define un recurso sin estado, inequívocamente direccionable mediante una **URI** y un conjunto de operaciones denominado **CRUD**

Bibliografía

[1] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," mar 2016. [Online]. Available: http://arxiv.org/abs/1603.07285

[2] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative Adversarial Networks," jun 2014. [Online]. Available: http://arxiv.org/abs/1406.2661

[3] K. O'Shea and R. Nash, "An Introduction to Convolutional Neural Networks," nov 2015. [En línea]. Disponible en: http://arxiv.org/abs/1511.08458

[4] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, "Image-to-Image Translation with Conditional Adversarial Networks."

[5] F. A. Oliehoek, R. Savani, J. Gallego, E. van der Pol, and R. Groß, "Beyond Local Nash Equilibria for Adversarial Networks," jun 2018. [En línea]. Disponible en: http://arxiv.org/abs/1806.07268http://dx.doi.org/10.1007/978-3-030-31978-6{_}7

[6] N. Kodali, J. Abernethy, J. Hays, and Z. Kira, "On Convergence and Stability of GANs," may 2017. [En línea]. Disponible en: http://arxiv.org/abs/1705.07215

[7] K. Binmore, *La teoría de juegos: una breve introducción.* Madrid: Alianza Editorial, 211.

[8] D. Freedman, *Estadística*, 2nd ed., 1993.

[9] Z. Wang, A. C. Bovik, H. Rahim Sheikh, and E. P. Simoncelli, "Image Quality Assessment: From Error Visibility to Structural Similarity," *IEEE TRANSACTIONS ON IMAGE PROCESSING*, vol. 13, no. 4, 2004. [En línea]. Disponible en: http://www.cns.nyu.edu/{~}lcv/ssim/.

[10] R. H. Hahnloser, R. Sarpeshkar, Misha A Mahowald, R. J. Douglas, and H. S. Seung, "Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit," Tech. Rep., 2000. [En línea]. Disponible en: www.nature.com

[11] Machinelearningmastery, "How to Fix the Vanishing Gradients Problem Using the ReLU." [En línea]. Disponible en: https://machinelearningmastery.com/how-to-fix-vanishing-gradients-using-the-rectified-linear-activation-function/

[12] E. Shelhamer, J. Long, and T. Darrell, "Fully Convolutional Networks for Semantic Segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 4, pp. 640–651, apr 2017.

[13] K. Chellapilla, S. Puri, P. Simard, and P. S. High, "High Performance Convolutional Neural Networks for Document Processing," Tech. Rep., 2006. [En línea]. Disponible en: https://hal.inria.fr/inria-00112631

[14] X. Glorot, A. Bordes, and Y. Bengio, "Deep Sparse Rectifier Neural Networks," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, G. Gordon, D. Dunson, and M. Dudík, Eds., vol. 15. Fort Lauderdale, FL, USA: PMLR, 2011, pp. 315–323. [En línea]. Disponible en: http://proceedings.mlr.press/v15/glorot11a.html

[15] M. A. Kramer, "Nonlinear principal component analysis using autoassociative neural networks," *AIChE journal*, vol. 37, no. 2, pp. 233–243, 1991.

[16] G. E. Hinton and R. S. Zemel, "Autoencoders, minimum description length and Helmholtz free energy," in *Advances in neural information processing systems*, 1994, pp. 3–10.

[17] D. P. Kingma, M. Welling, and …, "An introduction to variational autoencoders," *Foundations and Trends in Machine Learning*, vol. 12, pp. 307—-392, 2019.

[18] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9351, pp. 234–241, 2015.

[19] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," pp. 19–27, 2016. [En línea]. Disponible en: http://ethanschoonover.com/solarizedhttp://arxiv.org/abs/1603.07285

[20] A. Radford, L. Metz, and S. Chintala, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks."

[21] M. Mirza and S. Osindero, "Conditional Generative Adversarial Nets."

[22] H. Kwak and B.-T. Zhang, "Ways of Conditioning Generative Adversarial Networks."

[23] Mart\'\in~Abadi, Ashish~Agarwal, Paul~Barham, Eugene~Brevdo, Zhifeng~Chen, Craig~Citro, Greg~S.~Corrado, Andy~Davis, Jeffrey~Dean, Matthieu~Devin, Sanjay~Ghemawat, Ian~Goodfellow, Andrew~Harp, Geoffrey~Irving, Michael~Isard, Y. Jia, Rafal~Jozefowicz, Lukasz~Kaiser, Manjunath~Kudlur, Josh~Levenberg, Dandelion~Mané, Rajat~Monga, Sherry~Moore, Derek~Murray, Chris~Olah, Mike~Schuster, Jonathon~Shlens, Benoit~Steiner, Ilya~Sutskever, Kunal~Talwar, Paul~Tucker, Vincent~Vanhoucke, Vijay~Vasudevan, Fernanda~Viégas, Oriol~Vinyals, Pete~Warden, Martin~Wattenberg, Martin~Wicke, Yuan~Yu, and Xiaoqiang~Zheng, "{TensorFlow}: Large-Scale Machine Learning on Heterogeneous Systems," 2015. [En línea]. Disponible en: https://www.tensorflow.org/

[24] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d\textquotesingle Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [En línea]. Disponible en: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[25] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems," dec 2015. [En línea]. Disponible en: http://arxiv.org/abs/1512.01274

[26] Theano Development Team, "Theano: A {Python} framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.0, may 2016. [En línea]. Disponible en: http://arxiv.org/abs/1605.02688

[27] F. Chollet, *Deep Learning with Python*. Manning Publications, 2017. [En línea]. Disponible en: https://www.ebook.de/de/product/28930398/francois{_}chollet{_}deep{_}learning{_}with{_}python.html

[28] J. Walke, Facebook, and Community, "React - A JavaScript library for building user interfaces," 2013. [En línea]. Disponible en: https://reactjs.org/

[29] A. Ronacher, "Flask (web framework)," 2010. [En línea]. Disponible en: http://palletsprojects.com/p/flask

[30] M. Fowler and J. Lewis, "Microservices, a definition of this new architectural term," 2014. [En línea]. Disponible en: https://martinfowler.com/articles/microservices.html

[31] S. Hykes and Docker, "Docker," 2013. [En línea]. Disponible en: https://www.docker.com/

[32] K. E. Lantz, *The prototyping methodology*. Prentice-Hall, 1986.

[33] J. Bjorck, C. Gomes, B. Selman, and K. Q. Weinberger, "Understanding Batch Normalization," may 2018. [En línea]. Disponible en: http://arxiv.org/abs/1806.02375

[34] S. Cai, Y. Shu, W. Wang, M. Zhang, G. Chen, and B. C. Ooi, "Effective and Efficient Dropout for Deep Convolutional Neural Networks," apr 2019. [En línea]. Disponible en: http://arxiv.org/abs/1904.03392

[35] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, "Improved Techniques for Training GANs."

[36] C. K. Sønderby, J. Caballero, L. Theis, W. Shi, and F. Huszár, "Amortised MAP Inference for Image Super-resolution," oct 2016. [En línea]. Disponible en: http://arxiv.org/abs/1610.04490

[37] Postman, "Postman | The Collaboration Platform for API Development." [En línea]. Disponible en: https://www.postman.com/

[38] Kubernetes, "Orquestación de contenedores para producción - Kubernetes." [En línea]. Disponible en: https://kubernetes.io/es/

[39] R. Zhang, P. Isola, and A. A. Efros, "Colorful image colorization," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9907 LNCS. Springer Verlag, 2016, pp. 649–666.

[40] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-ResNet and the impact of residual connections on learning," in *31st AAAI Conference on Artificial Intelligence, AAAI 2017*. AAAI press, feb 2017, pp. 4278–4284.

[41] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2016-Decem, pp. 779–788, jun 2015. [En línea]. Disponible en: http://arxiv.org/abs/1506.02640

[42] D2l.ai, "13.2. Fine Tuning — Dive into Deep Learning 0.7.1 documentation." [En línea]. Disponible en: https://d2l.ai/chapter{_}computer-vision/fine-tuning.html

[43] K. Das, J. Jiang, and J. N. K. Rao, "Mean squared error of empirical predictor," jun 2004. [En línea]. Disponible en: http://arxiv.org/abs/math/0406455http://dx.doi.org/10.1214/009053604000000201