Universitat Oberta de Catalunya

Uso de técnicas de Deep Learning para mejorar los gráficos de videojuegos antiguos

Autor: Francisco José Martínez Simón

Tutor: Jordi Duch Gavaldà

Profesor: Joan Arnedo Moreno

Diseño y desarrollo de videojuegos

Nuevas tecnologías e investigación

04/06/2022

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Uso de técnicas de Deep Learning para mejorar los gráficos de videojuegos antiguos*
Nombre del autor:	*Francisco José Martínez Simón*
Nombre del colaborador/a docente:	*Jordi Duch Gavaldà*
Nombre del PRA:	*Joan Arnedo Moreno*
Fecha de entrega (mm/aaaa):	*06/2022*
Titulación o programa:	*Desarrollo y diseño de videojuegos*
Área del Trabajo Final:	*Nuevas tecnologías e investigación*
Idioma del trabajo:	*Español*
Palabras clave	*Deep Learning, Red neuronal convolucional, super resolución*

Resumen del Trabajo (máximo 250 palabras): *Con la finalidad, contexto de aplicación, metodología, resultados y conclusiones del trabajo*

Hoy en día, muchos de los juegos que salen al mercado son adaptaciones de títulos antiguos, los cuales presentan mejoras técnicas (normalmente, en el apartado gráfico). Sin embargo, sigue habiendo una gran cantidad de videojuegos antiguos que no son relanzados con estas mejoras.

Pensamos que el principal inconveniente a la hora de preparar un juego del pasado, para su lanzamiento en el presente, es el coste técnico, por lo que algunas compañías optan por distribuir estos juegos centrándose solamente en que se puedan ejecutar en dispositivos modernos.

Por tanto, mediante este trabajo buscamos investigar sobre técnicas que nos permitan mejorar los gráficos de un videojuego usando la inteligencia artificial; más concretamente, Deep Learning.

Abstract (in English, 250 words or less):

Nowadays, many of the games that come out on the market are adaptations of old titles, which present technical improvements (normally, in the graphic section). However, there are still many older videogames that are not re-released with these improvements.

We think that the main drawback when preparing a game from the past, for its release in the present, is the technical cost, which is why some companies choose to distribute these games focusing only on the fact that they can be run on modern devices.

Therefore, through this work we seek to investigate techniques that allow us to improve the graphics of a video game using artificial intelligence; more specifically, Deep Learning.

Este Trabajo Fin de Máster está dedicado a mis amigos y compañeros de la UOC, con los que he ido aprendiendo cada día. Además, este proyecto también está dedicado a mi tutor, Jordi, que me ha ayudado a dar forma a este proyecto.

Agradecimientos

Quisiera dar las gracias a mis amigos, y compañeros de estudios (especialmente, a José) por ayudarme a resolver mis dudas (tanto en forma de ánimos, como de conocimiento). De no contar con ellos, la elaboración de este trabajo habría sido mucho más difícil, puede que, incluso, imposible.

No deben quedar sin mencionar mis otros amigos en estos agradecimientos. Ellos han sido pacientes conmigo, cuándo yo estaba ocupado con este trabajo, y, cuando necesitaba distraerme un poco y despejarme, siempre estuvieron disponibles para mí.

Por último, aprovecho para agradecer a todo el equipo docente; los cuales, gracias al conocimiento que me han impartido, me han permitido llegar hasta este punto.

Resumen

Hoy en día, muchos de los juegos que salen al mercado son adaptaciones de títulos antiguos, los cuales presentan mejoras técnicas (normalmente, en el apartado gráfico). Sin embargo, sigue habiendo una gran cantidad de videojuegos antiguos que no son relanzados con estas mejoras.

Pensamos que el principal inconveniente a la hora de preparar un juego del pasado, para su lanzamiento en el presente, es el coste técnico, por lo que algunas compañías optan por distribuir estos juegos centrándose solamente en que se puedan ejecutar en dispositivos modernos.

Por tanto, mediante este trabajo buscamos investigar sobre técnicas que nos permitan mejorar los gráficos de un videojuego usando la inteligencia artificial; más concretamente, *Deep Learning*.

Abstract

Nowadays, many of the games that come out on the market are adaptations of old titles, which present technical improvements (normally, in the graphic section). However, there are still many older videogames that are not re-released with these improvements.

We think that the main drawback when preparing a game from the past, for its release in the present, is the technical cost, which is why some companies choose to distribute these games focusing only on the fact that they can be run on modern devices.

Therefore, through this work we seek to investigate techniques that allow us to improve the graphics of a video game using artificial intelligence; more specifically, Deep Learning.

Palabras clave

Deep Learning, Red neuronal convolucional, super resolución

Notaciones y Convenciones

A lo largo de este documento, hemos decidido seguir las siguientes pautas a la hora de elaborar diferentes aspectos de este:

- Todas las Figuras y Mesas del documento se encuentran en fuente Arial, con un tamaño de ocho.
- Para las palabras provenientes de un idioma extranjero, así como terminología tecnológica, hemos decidido por representarlas mediante cursiva.
- Hemos decidido escribir los títulos de cada capítulo en fuente Arial, con un tamaño de veinte y resaltarlas en negrita.
- Los títulos de las secciones de capítulos (incluyendo cualquier posible nivel de subsección) se encuentran escritos en fuente Arial, con un tamaño de trece y resaltadas en negrita.
- El resto del texto (de cada capítulo) se encuentra en fuente Arial y con un tamaño de once.

Índice

Figuras y tablas

Índice de figuras

Índice de tablas

1. Introducción

A través de este Trabajo Fin de Máster, buscamos estudiar y desarrollar técnicas que nos permitan mejorar el apartado gráfico de videojuegos, los cuales han quedado obsoletos, desde el punto de vista artístico. Para cumplir con este objetivo, vamos a recurrir a la inteligencia artificial; más concretamente, nos centraremos en el área del *Deep Learning*.

1.1. Descripción

En la actualidad, podemos apreciar cómo una considerable parte de los videojuegos que salen al mercado, o son anunciados, son relanzamientos de videojuegos antiguos. Señalemos un ejemplo cercano, el clásico de 1997, "*Final Fantasy VII*" [1], recibió un "*remake*" y se lanzó una nueva versión de él en el año 2020, "*Final Fantasy VII Remake*" [2]. Sin embargo, este juego contaba con un gran trabajo detrás: cambios en la historia, nueva banda sonora y nuevo sistema de combate, entre otros. Por otro lado, debemos señalar otros casos más modestos, donde los cambios sean más sutiles; ejemplo de esta situación podría ser "*The Legend of Zelda: Skyward Sword HD*" [3], el cual salió en 2021, un "*remaster"* del videojuego de 2011: "*The Legend of Zelda: Skyward Sword*" [4]. En este caso, los cambios estaban centrados en el apartado artístico del juego, dotando a esta nueva versión de gráficos en alta definición, hecho que se aproxima más a lo que intentamos encontrar mediante este proyecto.

A la hora de justificar el aumento de los relanzamientos de juegos antiguos, encontramos dos principales motivos: el envejecimiento de la población que juega a videojuegos y la posibilidad de jugar a juegos que dejaron de estar disponibles.

En primer lugar, centrémonos en el argumento relativo al envejecimiento de la población de videojuegos. Si consultamos los datos de "*2021 Essential Facts About the Video Game Industry*" [5], podemos observar cómo más del 40% de personas que juegan a videojuegos, en los Estados Unidos de América, tienen más de 35 años, observar el gráfico de la Figura 1. Aunque dentro de este grupo podríamos encontrar a jugadores nóveles, una parte de este grupo serán jugadores veteranos, los cuales podrían sentirse atraídos por la idea de volver a jugar a clásicos que disfrutaron siendo más jóvenes.

Figura 1: Gráfico de distribución de jugadores por rangos de edades

Respecto al segundo argumento, partiremos también del gráfico de la Figura 1; tal y cómo podemos apreciar en él, el 20% de personas que juegan a videojuegos es menor de 18 años (en Estados Unidos), lo que significa que no han tenido la oportunidad de probar una gran cantidad de videojuegos que salieran antes del 2000. Sin embargo, esto no significa que esta parte del público no esté interesada en esos juegos. Por ejemplo, un joven que hoy jugara al videojuego "*The Legend of Zelda: Breath of the Wild*" [6], podría interesarse por clásicos de la saga que no estuvieran hoy a su disposición. Además, existen juegos cuyo lanzamiento se limitó a alguna zona específica, cómo el videojuego "*Fire Emblem: Ankoku Ryū to Hikari no Tsurugi*" [7], el cual solo estuvo disponible en Japón hasta el año 2020.

Con esto, habríamos presentado los que pensamos que son los principales motivos del aumento del número de relanzamientos de videojuegos antiguos. Sin embargo, aún debemos finalizar otra tarea, unir este hecho con el propósito de nuestro trabajo, mejorar los gráficos de estos juegos.

Para justificar la importancia de mejorar los gráficos de videojuegos antiguos, vamos a basarnos en los datos de "*2019 Essential Facts About the Computer and Video Game Industry*" [8]. En este caso, este estudio contempla distintos motivos que sirven a los jugadores a la hora de realizar sus compras. Uno de estos motivos es la calidad gráfica,

motivo que afecta a un 63% de las personas que compran videojuegos en los Estados Unidos.

De esta forma, podemos justificar, así como definir más fidedignamente, el propósito de este trabajo. Aunque veamos un incremento en el número de videojuegos clásicos que reciben readaptaciones y vuelven a ser lanzados al mercado, muchos títulos antiguos quedan olvidados, pues no todos los videojuegos pertenecen a las sagas más famosas o tuvieron la misma repercusión en su época, lo que significa que el costo de adaptar estos juegos puede que no sea cubierto. Por tanto, queremos estudiar una tecnología que podría facilitarnos esta tarea, reduciendo costes y haciendo que un mayor número de videojuegos puedan ser disfrutados en el presente.

1.2. Objetivos

El objetivo principal de este trabajo es la obtención de una red neuronal que nos permita mejorar los gráficos de un videojuego. Para realizar esto, nos valdremos de una técnica conocida cómo Super Resolución, la cual nos permite incrementar la resolución de una imagen, de forma que los nuevos píxeles nos permitan obtener la misma imagen con mayor calidad.

Además, partiendo del objetivo principal que acabamos de comentar, podemos definir la existencia de otros objetivos secundarios, los cuales componen el principal. Algunos de estos objetivos serían: la elaboración del conjunto de datos de entrenamiento y prueba (*Dataset*), la elaboración de las operaciones de preprocesamiento que aplicaremos sobre nuestros datos antes de introducirlos en la red neuronal y el diseño de la arquitectura de nuestra red neuronal, entre otros.

Por último, debemos también destacar cómo este trabajo puede repercutir personalmente en nosotros. Debido a la naturaleza de investigación de este proyecto, podremos aumentar nuestro conocimiento en una de las áreas más interesantes de la informática, el *Deep Learning*; además, podremos combinarlo con el tema del máster que estamos cursando, Diseño y Desarrollo de Videojuegos; lo que nos permite combinar de forma satisfactoria dos de nuestros mayores intereses, relacionados con la informática.

1.3. Metodología y proceso de trabajo

En primer lugar, debemos contextualizar este Trabajo Fin de Máster dentro de los estudios que estamos cursando, el “Máster de Desarrollo y Diseño de Videojuegos”. Dentro del máster, existen tres opciones a la hora de realizar este trabajo; en nuestro caso, nos hemos decantado por la opción de “Nuevas Tecnologías e Investigación”; este hecho es muy importante y tendrá una gran repercusión en cómo gestionemos este proyecto.

Debido al tipo de nuestro trabajo, nuestro objetivo no es desarrollar un videojuego o algún producto relacionados con ellos; en su lugar, lo que buscamos es realizar un estudio sobre cómo podemos emplear la inteligencia artificial para mejorar la calidad gráfica de videojuegos ya existentes.

Dado que vamos a trabajar en el desarrollo de una red neuronal, las distintas fases de nuestro proyecto estarán bastante claras (veremos estas fases de forma detallada en la siguiente sección) y, además, estarán interrelacionadas entre ellas; por poner un ejemplo, no podemos empezar a entrenar la red neuronal sin haber conseguido los datos de entrenamiento.

Partiendo de los puntos indicados anteriormente, hemos decidido que para desarrollar este Trabajo Fin de Máster nos basaremos en un modelo en cascada [9]. El modelo en cascada destaca por estar estructurado en etapas, donde cada etapa comienza con el fin de otra etapa (excepto la primera etapa). En el modelo en cascada, se debe definir al principio las estimaciones temporales, así como los procesos que componen el proyecto sobre el que se va a aplicar esta metodología de desarrollo software. En la mayoría de los casos, los modelos en cascada suelen seguir una estructuración parecida a la siguiente:

1) Fase de Obtención y Documentación de Requisitos.
2) Fase de Diseño del Programa.
3) Fase de Implementación.
4) Fase de Realización de Pruebas.
5) Fase de Despliegue del Programa.
6) Fase de Mantenimiento.

Sin embargo, consideramos que las dos últimas fases de la estructura planteada no son relevantes para este proyecto.

1.4. Planificación

Para poder asegurarnos de cumplir con los objetivos de este Trabajo Fin de Máster, así como cumplir con los plazos temporales, debemos recurrir a técnicas relacionadas con la gestión de proyectos y la ingeniería del software.

Nuestro primer paso será estudiar el problema que tenemos que afrontar y seleccionar la mejor estrategia para hacerlo. Si analizamos el proceso de elaboración de un Trabajo Fin de Máster, podemos destacar los siguientes puntos claves:

- El marco temporal en el que desarrollar las distintas partes del Trabajo Fin de Máster son claras, tal y como se puede ver en la Tabla 1.
- Analizando los objetivos de nuestro propio proyecto, podemos ver que existe una dependencia directa entre muchas partes. Por ejemplo, no podemos obtener el modelo de *Deep Learning* sin tener un conjunto de datos previo (*Dataset*).

	Intervalos de Fechas
PEC-1: Plan del proyecto	16/02/2022-06/03/2022
PEC-2: Estado del arte	07/03/2022-03/04/2022
PEC-3: Implementación	04/04/2022-08/05/2022
PEC-4: Memoria	09/05/2022-05/06/2022
PEC-5: Defensa	13/06/2022-19/06/2022

Mesa 1: Fechas de actividades

Para definir el modelo en cascada que vamos a usar, nos centraremos en materias relacionadas con la implementación y el diseño, Debido a esto, nuestra siguiente tarea será identificar cada parte relacionada con estas dos áreas, explicándolas brevemente y otorgándoles una estimación temporal. A continuación, se muestran los procesos relacionados con las fases de implementación y de diseño.

1) Recopilación de información sobre las redes neuronales convolucionales; también conocidas como *CNNs* y de información sobre la Super Resolución (acrónimo del inglés *Convolutional Neural Network*) [10, 11, 12]. Estimación: 40 horas.
2) Creación del conjunto de datos. Para poder entrenar nuestra *CNN*, necesitamos un conjunto de imágenes relacionadas con nuestro problema. Estimación: 40 horas.

3) Preparación del entorno de trabajo. Tendremos que instalar las herramientas necesarias para desarrollar el proyecto: principalmente, un *IDE* de *Python* [13] y la librería *Keras* [14]. Estimación: 20 horas.
4) Diseño de la *CNN*. Con nuestro software instalado y con los conocimientos necesarios adquiridos, podremos crear la arquitectura de nuestra *CNN*. Estimación: 60 horas.
5) Entrenamiento, prueba y ajuste del modelo. Con las fases anteriores completadas, realizaremos de forma iterativa el siguiente proceso: entrenar la red, analizar los resultados obtenidos y ajustar los valores necesarios (desde cambios en la arquitectura, hasta reconfiguraciones de los hiperparámetros). Este proceso se repetirá hasta que alcancemos una precisión notable y el sobreajuste (*overfitting*) se reduzca a un valor aceptable [10, 11, 12]. Estimación: 60 horas.
6) Elaboración del documento del Trabajo Fin de Máster. Durante la ejecución de cada fase anterior se realizará de forma paralela un documento que refleje el trabajo realizado. Estimación: 60 horas.
7) Preparación de la defensa. Como última fase del proyecto, destinaremos una semana, mientras seguimos acabando el documento, a preparar la defensa del trabajo. Estimación: 20 horas.

Si contabilizamos la estimación de cada fase, obtenemos un total de 300 horas. Dado que este trabajo se realizará entre los días 21/02/2022 y 05/06/2022, eso nos da un total de 15 semanas, lo que supone 20 horas de trabajo por semana de media.

De esta forma, podemos definir los hitos de nuestro de proyecto (las siete fases que acabamos de mencionar) y calcular cómo se extenderá su duración a lo largo de las quince semanas que tenemos. Además, podremos compararlos y ver cómo se ajustan a las fechas de entregas de las actividades en la que se encuentra dividida esta asignatura.

Tareas	Comienzo de la Tarea	Fin de la Tarea	Duración(horas)
Recopilación de información	21/02	06/03	40
Creación del Dataset	07/03	20/03	40
Preparación del entorno de trabajo	21/03	27/03	20
Diseño de la CNN	28/03	17/04	60
Entrenamiento, prueba y ajuste del modelo	18/04	08/05	60
Elaboración del documento del trabajo fin de grado	21/02	05/06	60
Preparación de la defensa	30/05	05/06	20

Mesa 2: Comienzo, fin y duración de las tareas

Si comparamos las dos tablas anteriores, Mesa 1 y Mesa 2, podemos ver cómo las actividades casan perfectamente. Sobre todo, debemos destacar que para la fecha de entrega de la segunda actividad (relacionada con el estado del arte), habremos completado la tarea relacionada con la recopilación de información. Además, debemos también destacar que para la fecha de entrega de la tercera actividad (relacionada con la implementación), estimamos que habremos terminado de ajustar nuestro modelo.

Continuando con la gestión del proyecto, hemos diseñado un diagrama de *Gantt* que nos ayude a comprender la organización de las fases de forma más clara, tal y cómo puede observarse en la siguiente figura.

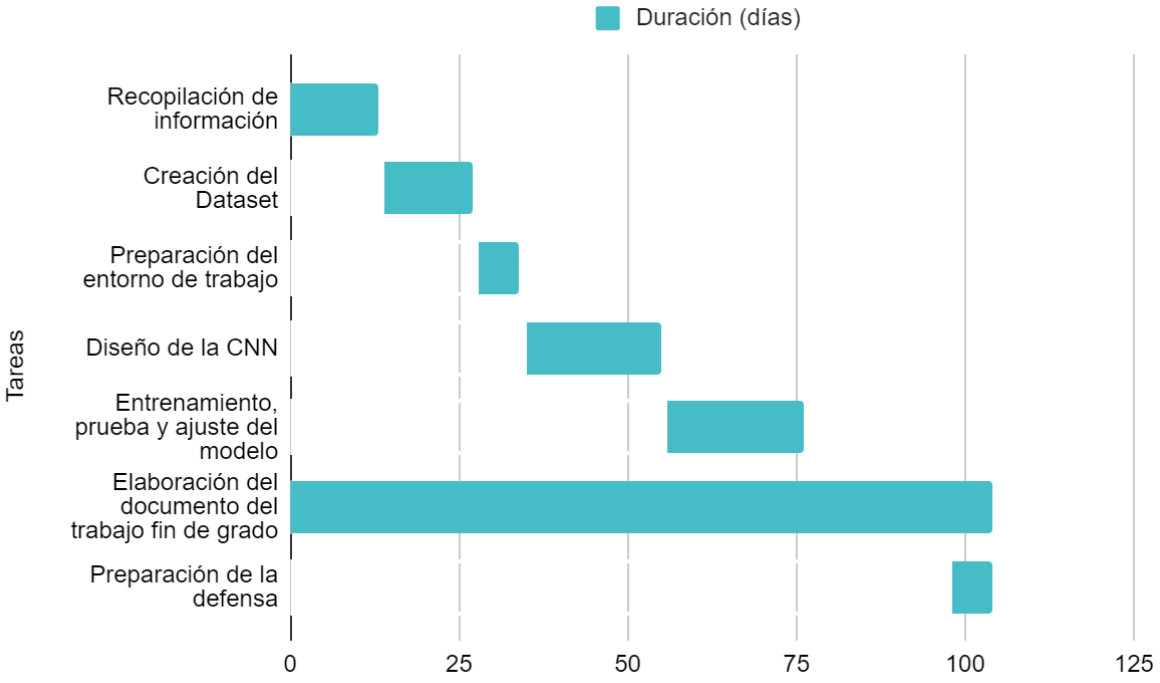

Figura 2: Diagrama de Gantt

1.5. Presupuesto

A la hora de realizar nuestro proyecto, debemos considerar los medios con los que contamos. Dado que vamos a entrenar nuestra red neuronal en nuestro propio ordenador, tendremos que especificar los componentes de dicho dispositivo:

- Tarjeta Gráfica: Es uno de los principales requisitos hardware de nuestro proyecto. Este elemento de nuestro ordenador será el encargado del entrenamiento de nuestra red neuronal.
- Procesador: Las actividades ligadas a este componente serán las relacionadas con la creación del *Dataset*, por lo que, aunque sea menos crucial que la tarjeta gráfica, debemos tenerlo en cuenta.
- Memoria RAM: Durante el entrenamiento, esta memoria gestionará la información que le pasamos a nuestra red, por lo que debemos asegurarnos de tener la capacidad suficiente.

Además, debemos también destacar que no necesitaremos de ningún coste adicional que pueda estar relacionado con: la obtención de material académico, el acceso a un conjunto de datos de entrenamiento o alguna herramienta software. De esta forma, tal y cómo se podrá corroborar en futuras partes de este documento, recurriremos a recursos gratuitos, así como otros recursos ya a nuestro alcance.

Tipo de Componente	Componente del Dispositivo
Tarjeta Gráfica	Gigabyte GeForce GTX 1050Ti
Procesador	Intel Core i5-7400
Memoria RAM	Kingston Fury

Mesa 3: Componetes del dispositivo

Otra información de interés relativa a los componentes anteriores puede encontrarse a continuación:

- Tarjeta Gráfica: Memoria GDDR5 de 4GB.
- Procesador: Frecuencia de 3.0 GHz.
- Memoria RAM: Memoria DDR4 de 16GB.

1.6. Estructura del documento

Para finalizar el primer capítulo de este documento, mostraremos un resumen del contenido de cada capítulo:

- Capítulo 1: Introducción. En este capítulo se introduce el Trabajo Fin de Máster, describiendo los objetivos que se han planteado, la planificación temporal que se ha llevado a cabo, así como la motivación y contexto por la cual se ha realizado este proyecto.
- Capítulo 2: Materiales, Métodos y Estado del Arte. En esta sección se definirán las herramientas que hemos utilizado para desarrollar este Trabajo Fin de Máster. Y ya, por último, localizaremos nuestro proyecto en el contexto tecnológico actual.
- Capítulo 3: Propuesta. Para cumplir los objetivos definidos en el primer capítulo existen múltiples soluciones; por lo tanto, en este capítulo nos centraremos en analizar algunas soluciones distintas para distintos problemas. Además, terminaremos de perfilas algunos objetivos de este trabajo.
- Capítulo 4: Diseño. Durante este capítulo, realizaremos los primeros pasos necesarios para obtener nuestro modelo de inteligencia artificial. Por un lado, obtendremos los datos de entrenamiento. Por otro lado, especificaremos la red neuronal de la que partiremos.
- Capítulo 5: Implementación. Este capítulo estará centrado en el entrenamiento de la red. Podremos ver los resultados obtenidos para las distintas configuraciones de varios parámetros de nuestro proyecto.
- Capítulo 6: Demostración. Analizaremos los resultados obtenidos hasta este punto. Además, los compararemos con resultados obtenidos mediante métodos distintos y analizaremos los posibles usos del proyecto.
- Capítulo 7: Conclusiones y líneas de futuro. En este capítulo final, indicaremos las principales conclusiones obtenidas tras finalizar nuestro proyecto. Por último, se detallarán una serie de mejoras que puedan ser beneficiosas para un futuro.

2. Materiales, Métodos y Estado del Arte

A lo largo de este capítulo, veremos todas aquellas herramientas, tecnologías y métodos empleados para la realización de este Trabajo Fin de Máster. Además, localizaremos nuestro proyecto en el contexto tecnológico actual, lo que nos ayudará a comprender mejor nuestro trabajo.

2.1. Tecnologías

En este apartado, veremos aquellas tecnologías que han sido usadas para desarrollar las distintas fases de las que se compone nuestro proyecto.

2.1.1. Python

Este lenguaje de programación que vio la luz en 1991 y fue desarrollado por Guido van Rossum en la década que va desde 1980 a 1990. A día de la redacción de este documento es uno de los lenguajes de programación más usados en el mundo [13]. El éxito de *Python* se debe a la sencillez que requiere usarlo y a la comunidad que se ha construido en torno al lenguaje en los últimos años, comunidad construida gracias a las siguientes características de *Python*: es un software libre, de código abierto y gratis.

Figura 3: Logo de Python

La actual popularidad de la que goza *Python* ha hecho que sea un lenguaje multipropósito, el cual es usado en variedad de campos, como el desarrollo web y la inteligencia artificial, dos campos que nos interesan para realizar este Trabajo Fin de Máster. Sin embargo, nuestro uso de *Python* se centrará en el segundo ejemplo, la Inteligencia Artificial.

Python cuenta con una gran cantidad de librerías enfocadas a la Inteligencia Artificial. Además, de las librerías que *Python* nos ofrece para este campo, muchas nos pueden ser útiles a nosotros, como *Keras* o *TensorFlow*, las cuales veremos en detalle más adelante.

Para terminar de revisar este lenguaje de programación, veremos algunas de las características básicas del mismo:

- Lenguaje Intepretado: Al ser *Python* un lenguaje interpretado, el código se compila mediante lo que se conoce como un intérprete. De esta forma, el intérprete traduce el código fuente a código máquina, instrucción por instrucción, sin necesidad alguna de almacenar el código máquina.
- Lenguaje Dinámico: Esto significa que no necesitamos especificar el tipo de los datos de las variables; en su lugar, *Python* adapta el valor de las variables a lo que aparezca en el código.
- Multiparadigma: Esta propiedad supone que podamos valernos de *Python* para hacer uso de distintos paradigmas de programación. Por ejemplo, aunque *Python* sea muy usado desde una aproximación orientada a objetos, también podríamos usar *Python* para proyectos basados en programación imperativa.

2.1.2. Tensorflow

Tensorflow [15] es una librería desarrollada y mantenida por *Google*, la cual nos ofrece métodos para el desarrollo de redes neuronales.

En este caso, no vamos a tratar en profundidad los beneficios que supone usar *Tensorflow*. En su lugar, simplemente destacaremos que usaremos *TensorFlow* como "*backend*" de *Keras*, otra librería de *Python* que veremos a continuación.

Figura 4: Logo de Tensorflow

2.1.3. Keras

Anteriormente, vimos que *TensorFlow* nos serviría como base para *Keras* [14], pero no llegamos a introducir esta tecnología. *Keras* nos ofrece una gran variedad de funciones enfocadas en *Deep Learning*.

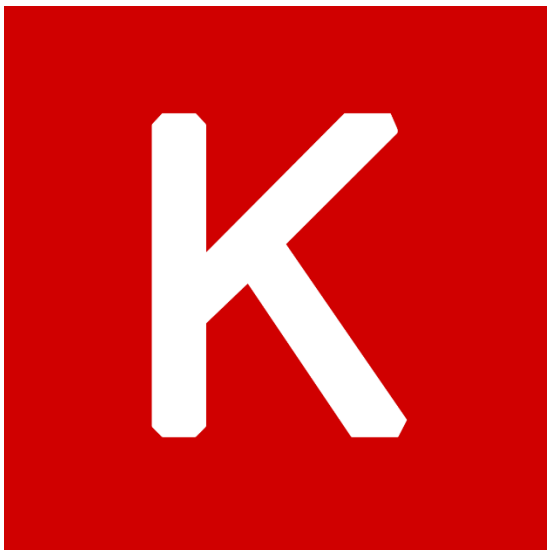

Figura 5: Logo de Keras

En nuestro caso, usaremos Keras para desarrollar los siguientes objetivos:

- Definir la arquitectura de nuestra red neuronal. Tal y cómo veremos más adelante, solo tendremos que importar dos tipos de capas (convolución y activación); así como, importar el tipo de modelo (secuencial).
- A la hora de entrenar el modelo, tendremos que elegir un optimizador, opción que también podemos importar gracias a esta librería.
- Para poder trabajar con nuestro modelo, una vez entrenado, tendremos que poder guardarlo y cargarlo, cosa que también nos permite hacer *Keras*.

2.1.4. OpenCV

Dado que vamos a trabajar con imágenes, necesitaremos de tecnologías que nos permitan realizar las operaciones que necesitemos con ellas. Para cumplir con nuestros objetivos, hemos decidido incorporar a nuestro proyecto la librería *OpenCV* [16].

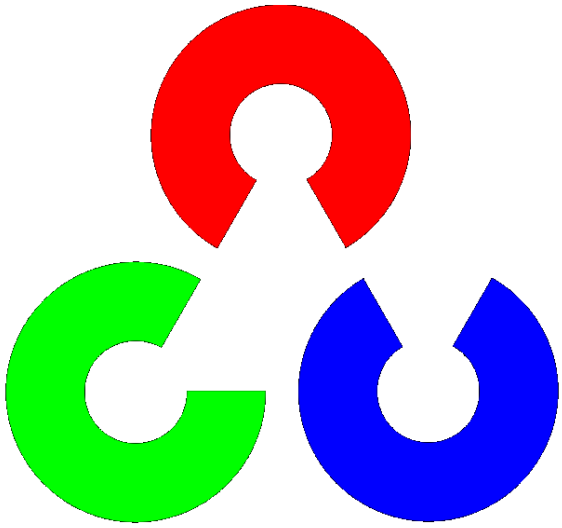

Figura 6: Logo de OpenCV

Tal y como hicimos con *Keras*; a continuación, veremos los usos que le daremos a la librería que estamos comentado en este apartado:

- Cuando probemos el funcionamiento del modelo entrenado, nos ayudará a cargar una imagen para que ejecutemos nuestra red neuronal sobre ella.
- Para construir nuestro *Dataset*, necesitaremos de *OpenCV* para que nos ayude a procesar las imágenes de las que se compondrá.

2.1.5. Otras Librerías

Con esto, habríamos visto las principales librerías que vamos a emplear en nuestro proyecto (*TensorFlow*, *Keras* y *OpenCV*). Sin embargo, existen otras librerías que vamos a emplear, aunque su relevancia sea menor.

De esta forma, en este último apartado, relativo a las tecnologías del proyecto, vamos a concentrar aquellas otras librerías que, sin ser tan importantes cómo las ya vistas, merecen ser mencionadas.

Más concretamente, vamos a comentar las dos siguientes librerías:

- *Numpy* [17]: Esta es una de las librerías más conocidas de *Python* y sirve para realizar algunas operaciones matemáticas. En nuestro caso, queremos aprovechar la capacidad de *Numpy* para trabajar con vectores y matrices, capacidad muy importante en el campo del aprendizaje automático.
- *MatPlotLib* [18]: Esta es la otra librería que pensamos que debe ser comentada, pues, aunque no tenga un impacto directo sobre el modelo que vamos a desarrollar, si nos permite agilizar el trabajo. *MatPlotLib* es una librería que nos facilita funciones para crear gráficas, las cuales, en el contexto de este proyecto, nos permitirán saber cómo se está comportando nuestra red neuronal

Figura 7: Izquierda, logo de Numpy. Derecha, logo de MatPlotLib

2.2. Herramientas

En la sección anterior, vimos aquellas tecnologías software necesarias para cumplir con los objetivos de este Trabajo Fin de Máster. En esta nueva sección, veremos las herramientas que utilizaremos para poder trabajar con las tecnologías anteriormente mencionadas.

2.2.1. PyCharm

PyCharm es un *IDE* (*Integrated Development Environment*, o en español, Entorno de Desarrollo Integrado) de *Python*, el cual fue desarrollado por la compañía *JetBrains* [19].

Figura 8: Logo de PyCharm

PyCharm provee a sus usuarios de muchas funcionalidades; entre estas funcionalidades, vamos a destacar la siguientes:

- Integra un depurador *Python*.
- Fácil gestión de librerías.
- Métodos de Refactorización.

Además, debemos destacar que *PyCharm* nos permite trabajar con todas las librerías de *Python* que vamos a necesitar para desarrollar este proyecto, desde *Keras*, hasta *Numpy*.

2.2.2. GitLab

Para poder organizar y gestionar nuestro trabajo, vamos a emplear un software de gestión de repositorios; más concretamente, nos hemos decantado por *GitLab* [20].

GitLab está basado en la tecnología *Git*, por lo que incorpora control de versiones, pero también incorpora funcionalidades relacionadas con la ingeniería del software.

Además, *GitLab* será una buena herramienta para poder compartir el trabajo que realicemos.

Figura 9: Logo de GitLab

2.2.3. SourceTree

Tal y como vimos en el apartado anterior, vamos a usar *GitLab* para gestionar el software que desarrollaremos. Sin embargo, a nivel local, usaremos *SourceTree* [21] para gestionar nuestro repositorio.

Una vez creemos nuestro repositorio en *GitLab*, podremos descargarlo en nuestro equipo gracias a esta herramienta. Además, *SourceTree* nos facilitará las distintas operaciones que tendremos que realizar sobre el repositorio, cómo hacer "*commit* "o actualizar el repositorio de *GitLab*.

Figura 10: Logo de SourceTree

2.3. Base Teórica

En este apartado, pretendemos dejar claros los conocimientos de los que nos hemos valido para trabajar con técnicas *de Deep Learning*.

De esta forma, iniciaremos un viaje que nos hará ver la evolución de las técnicas empleadas, comenzando con el perceptrón y tratando finalmente con las redes neuronales convolucionales. Además, este viaje nos permitirá desplazarnos también a lo largo del tiempo.

2.3.1. Perceptrón

En el año 1958, Frank Rosenblant introdujo el perceptrón, un algoritmo de aprendizaje automático que se basa en el uso de una función lineal y un vector de características [22].

El perceptrón se encuentra inspirado en las neuronas, las células básicas que forman el sistema nervioso. Estas células se valen de impulsos eléctricos para comunicarse con otras neuronas. Nuestros conocimientos actuales de las neuronas están basados en la doctrina de la neurona, idea desarrollada por el científico español Santiago Ramón y Cajal, galardonado con el premio Nobel de medicina en el año 1906 [23]. El punto clave de la doctrina de la neurona es la formación de redes por parte de estas células, siendo cada neurona una entidad discreta.

Con respecto a las partes de la neurona, podemos distinguir los siguientes elementos:

- Dendritas: Esta parte tiene labores de comunicación. En el caso de las dendritas, sus tareas de comunicación se limitan a la capacidad de recepción. Las dendritas de una neurona se encuentran conectadas a más de mil neuronas distintas y reciben los impulsos eléctricos que ellas le mandan. Las dendritas tienen la capacidad de, al recibir señales por parte de otras neuronas, hacer variar la proporción de neurotransmisores sinápticos. La adaptación matemática que se decidió realizar de este proceso, tal y como se verá más adelante, es la de dotar a las neuronas de la capacidad de recibir datos y potenciar cada valor (operación de multiplicación) de forma distintas, a través de un vector de pesos.
- Soma: También conocido como el cuerpo de una neurona, recibe la información captada por las dendritas y las unifica en una sola señal. En las redes neuronales artificiales, esta capacidad del soma se aprecia cuando, tras haber recibido los valores de entrada y haberlos multiplicado por su peso asociado, se suma cada uno de los resultados anteriores.
- Axón: Si las dendritas solo realizan actividades de comunicación receptivas, las neuronas necesitan de una forma de poder enviar información, de esto es encargan los axones. Cuando se alcanza una potencia eléctrica suficiente en el cuerpo de la neurona, se transmite un pulso eléctrico a través del axón, el cual conectará con las dendritas de otras neuronas. Este concepto, en el contexto de las redes neuronales artificiales, se traduce como la función de activación, la cual veremos en detalle más adelante.

Dentro de una neurona, podemos diferenciar más partes; sin embargo, para nuestra introducción a la base biológica de las redes neuronales artificiales, nos vale con estas tres partes comentadas.

Continuando con el perceptrón, nos centraremos ahora en ver cómo funciona. Para empezar, necesitamos definir las siguientes características:

- Función lineal: Se encarga de calcular el producto escalar de los datos de entrada y el vector de características.

- Datos de entrada: Un perceptrón cuenta con N entradas, las cuales definirán los parámetros sobre los que ejecutar el algoritmo.
- Sesgo: El sesgo es un componente del perceptrón que nos ayuda a ajustar el resultado obtenido. El sesgo no es un valor que se introduzca en el perceptrón, pero se comporta como si fuera un dato de entrada que vale uno.
- Vector de características: Representa el valor asociado con cada valor de entrada y con el sesgo. Los componentes del vector de características se multiplicarán con sus valores asociados.
- Función de activación: Función que se aplica sobre el resultado de la función lineal. Tradicionalmente, se trata de una función umbral que retorna cero o uno, en base a una condición.

Si tomamos los componentes que acabamos de definir y los juntamos, obtendremos la arquitectura del perceptrón, la cual se muestra en la Figura 11.

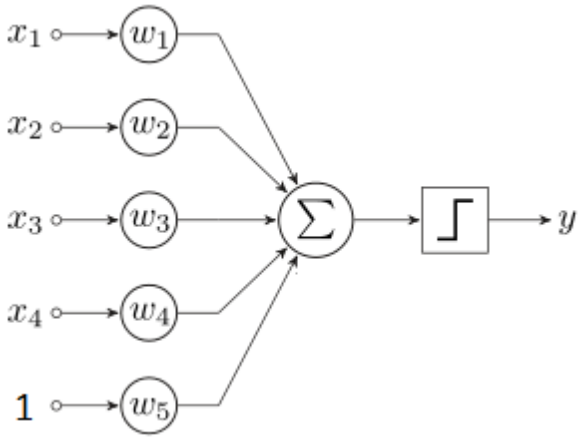

Figura 11: Arquitectura del Perceptrón

Sin embargo, para terminar de definir cómo funciona un perceptrón, pensamos que lo mejor es hacerlo mediante un par de ejemplos. Más concretamente, mostraremos cómo se comporta el perceptrón para resolver una puerta lógica *AND* y cómo se comporta el perceptrón para resolver una puerta lógica *XOR*.

En primer lugar, comenzaremos con la puerta lógicas *AND*, la tabla Mesa 3 muestra las salidas esperadas para las entradas posibles.

Entrada-A	Entrada-B	Salida
0	0	0
0	1	0
1	0	0
1	1	1

Mesa 4: Resultados para la operación AND

Si representamos el problema anterior en un plano, obtendremos algo similar a la Figura 12. En dicha figura podemos apreciar que los ceros y el uno son linealmente separables por una recta. Cuando aplicamos el perceptrón sobre un problema como este, lo que buscamos es obtener esa recta que separa los datos.

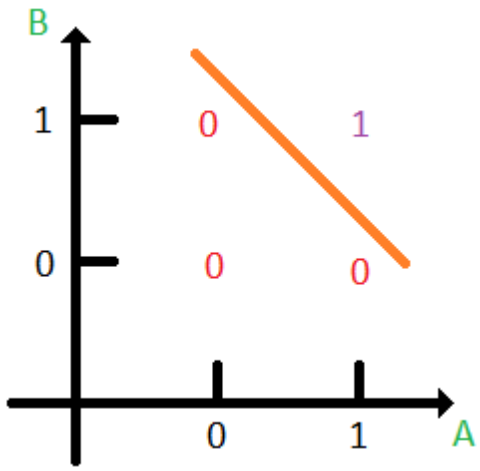

Figura 12: Representación de la solución del Perceptrón para la puerta lógica AND

Por otro lado, si intentamos recrear el experimento anterior sobre la puerta lógica *XOR*, encontraremos resultados totalmente distintos.

Entrada-A	Entrada-B	Salida
0	0	0
0	1	1
1	0	1
1	1	0

Mesa 5: Resultados para la operación XOR

Si, tal y como hicimos con la puerta lógica *AND*, representamos en un plano la tabla anterior, podremos apreciar una situación totalmente diferente, los ceros y los unos no pueden separarse usando una línea recta. La Figura 13 muestra varios intentos de separar estos datos.

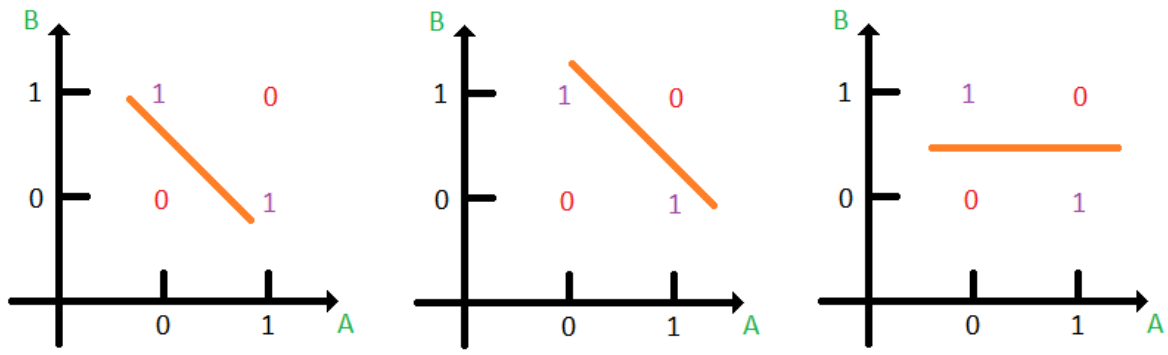

Figura 13: Intentos del Perceptrón para solucionar el problema de la puerta lógica XOR

El problema que acabamos de apreciar con el perceptrón se debe a que este algoritmo es incapaz de trabajar con datos que no sean linealmente separables. Este hecho fue demostrado por Minsky y Papert en 1969 [24]. Este hecho supuso el estancamiento de las investigaciones relacionadas con el perceptrón. Si embargo, debemos destacar que, en años posteriores, avances en el campo del perceptrón supusieron que volviera a hacerse popular.

2.3.2. Redes Neuronales

En el apartado anterior, vimos lo que era un perceptrón; sin embargo, no vimos el resultado de agrupar varios de ellos, formando una red neuronal. Dado que el perceptrón representa una neurona del sistema nervioso, la unión de varios perceptrones se conoce como red neuronal [25].

Debemos tener en cuenta que los perceptrones (a partir de ahora, neuronas) se agrupan en capas. De esta forma, podemos diferenciar tres tipos principales de capas:

- Capa de entrada: Esta capa define la información que viene del exterior y sobra la que se busca resolver un problema (por ejemplo, de clasificación).
- Capa oculta: A diferencia de la capa de entrada, en una red neuronal no hay una sola de estas capas; en su lugar, contamos con varias capas que se encargan de realizar las operaciones necesarias para obtener el resultado deseado.
- Capa de salida: Es la última capa de la red neurona. Partiendo de que el objetivo de nuestro proyecto es un problema de clasificación, en nuestro caso, nos devolverá la probabilidad de pertenencia a cada etiqueta de clase.

A la hora de trabajar con una red neuronal, la información viaja entre capas a través de lo que se conoce como el algoritmo de "*Forward Pass*". Este algoritmo se encarga de conectar los resultados obtenidos en las neuronas de una capa con las entradas de las neuronas de las siguientes capas. Para que se pueda apreciar mejor esta idea, debemos mostrar la arquitectura de una red neuronal. En la Figura 14 se muestra esta arquitectura, en la que cada esfera representa una neurona.

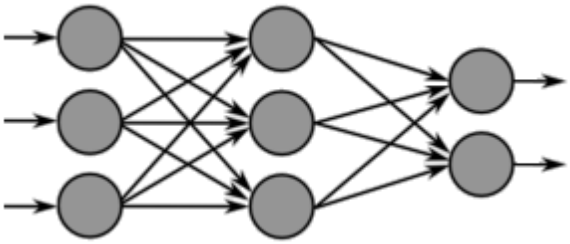

Figura 14: Arquitectura de la red neuronal

A continuación, veremos una serie de herramientas que podemos encontrar dentro de las redes neuronales y que son necesarias para poder obtener resultados y para entrenarlas.

- Función de Activación: Cuando vimos el perceptrón, también vimos que el resultado de la suma de los productos de los valores de entrada (y el sesgo) multiplicados por sus valores asociados se introducía en una función umbral. En las redes neuronales, se hace uso de un concepto similar; sin embargo, no solo usamos un tipo de función, sino que usamos varios. A continuación, se muestran algunas de las funciones de activación más usadas:

    - Unidad lineal rectificada (*ReLU*). Los valores negativos que recibe se convierten en cero, mientras que los valores positivos no varían. La fórmula de esta función es: $f(x)=\max(0,x)$.
    - Sigmoide. Devuelve valores entre cero y uno. Para realizar esto, se vale de asíntotas. La fórmula de esta función es: $f(x)=\frac{1}{1-e^{x}}$.
    - Tangente hiperbólica. Parecida a la función Sigmoide, pero, en esta ocasión, los valores que devuelven oscilan entre uno y el negativo de uno. La fórmula de esta función es: $f(x)=\frac{2}{1+e^{-2x}}-1$.

- Softmax. Función útil para obtener resultados en forma de probabilidades. La fórmula de esta función es: $f(x)_i = \frac{e^{x_i}}{\sum_{k=1}^{K} e^{x_k}}$. La suma de las probabilidades de la función Softmax debe devolver uno.

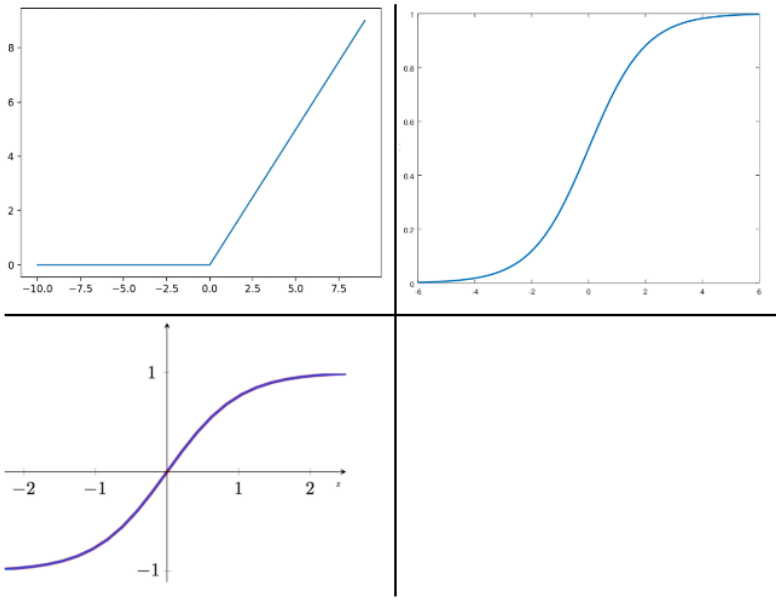

Figura 15: Arriba a la izquierda, ReLU. Arriba a la derecha, Sigmoide. Abajo, Tangente Hiperbólica

- Función de Pérdida. Estas funciones nos permiten saber cómo se está comportando nuestra red neuronal. Para realizar esto, comparan el valor de los resultados predichos con el valor de los resultados esperados. Esta función, aparte de como indicador de desempeño de nuestra red, nos permitirá entrenar nuestra red, pero eso lo veremos más adelante. A continuación, mostraremos algunas funciones de pérdida importantes:

  - Entropía cruzada. La entropía cruzada nos permite cuantificar la diferencia entre dos distribuciones de probabilidad. Su fórmula es: $H(p,q) = -\sum p(x)\log(q(x))$.
  - Error cuadrático medio. Se corresponde con el valor esperado de la pérdida del error al cuadrado. Su fórmula es $MSE(y,y') = \frac{\sum_{i=1}^{n}(y_i - y'_i)^2}{n}$.

Sin embargo, el componente más importante de las redes neuronales es el algoritmo de "*Backpropagation*", sobre todo, la versión que plantearon Rumelhart, Hinton y Williams en 1986 [26]. Las redes neuronales son algoritmo de aprendizaje automático, por lo que deben

tener la capacidad de ajustar sus propios parámetros y de mejorar sus capacidades con el tiempo y, para lograr esto, se usa el algoritmo de "*Backpropagation*". A continuación, se resumen los pasos a seguir para aplicar este algoritmo:

1) Se alimenta la red neuronal y se realiza un "*Forward Pass*", lo que nos dará un resultado, el cual podremos comprobar si es el correcto o no.
2) Revertiremos el "*Forward Pass*", yendo desde la última capa hasta la primera. Durante este proceso, calcularemos el gradiente de la función de pérdida. De esta forma, buscamos analizar la responsabilidad que tuvo cada neurona en el resultado obtenido.
3) Tendremos que ajustar los valores de los vectores de características de la capa actual.
4) Repetiremos los tres pasos anteriores para cada capa, tendremos que ir capa por capa, desde la última hasta la primera.

Aún nos quedan algunos conceptos por ver de las redes neuronales (sobre todo, conceptos ligados con el entrenamiento), pero los veremos más adelante en este capítulo.

2.3.3. Redes Neuronales Convolucionales

Las redes neuronales convolucionales fueron introducidas en 1998 por Yann LeCun. Este tipo de redes es conocido por destacar en tareas de visión artificial, motivo por el cual nosotros usaremos este tipo de red neuronal para nuestro problema [27].

Anteriormente, vimos que las redes neuronales alimentaban las neuronas de una capa con los resultados de las neuronas de la capa anterior. A partir de ahora, este tipo de capas se conocerán como *Fully-Connected*. Para poder hablar de redes neuronales convolucionales, tenemos la necesidad de introducir más de un tipo de capas, por lo que debemos tener claro cuándo realizaremos operaciones como en las redes neuronales clásicas.

La característica principal de las redes neuronales convolucionales, y aquella que la hace especialmente buena en problemas relativos al tratamiento digital de imágenes, es el uso de convoluciones, hecho que da nombre a estas redes.

Una convolución es un operador que nos permite, a partir de dos funciones, obtener una tercera función. En el caso de la convolución de imágenes, lo que hacemos es, a partir de una imagen y de un filtro, obtener una imagen filtrada.

La convolución de una imagen de tamaño NxM con un filtro de tamaño RxS funciona de la siguiente forma:

1) Situamos el filtro en el principio de la imagen, de forma que ocupe la intersección de las primeras 1-R filas y de las primeras 1-S columnas
2) Una vez está situado el filtro, multiplicaremos los elementos de la imagen con los elementos del filtro que tengan superpuestos; es decir, haremos un total de RxS multiplicaciones.
3) Cuando hayamos hecho todas las multiplicaciones, sumaremos los resultados. El resultado de esta suma será dividido por la suma de los elementos del filtro. Tras esto, tendremos el valor del primer elemento de nuestra imagen filtrada.
4) Nuestro siguiente paso será desplazar el filtro por la imagen. Si es posible, priorizaremos el desplazamiento horizontal. En caso contrario, tendremos que desplazar nuestra imagen verticalmente, lo que nos permitirá volver a desplazarla horizontalmente. Cada vez que desplacemos el filtro, tendremos que repetir los tres primeros pasos. De esta forma, podremos construir una nueva imagen, nuestra imagen filtrada.

Para que el concepto de la convolución quede más claro, haremos uso de un pequeño ejemplo. A continuación, se mostrará la imagen que usaremos para este ejemplo.

Figura 16: Imagen de prueba para aplicar filtro

Sin embargo, a la hora de trabajar con la imagen anterior, aunque sea una imagen a color, haremos uso de un solo canal. El siguiente paso será especificar el filtro a usar; en nuestro caso, hemos decidido hacer uso de un filtro de suavizado (lo que permite reducir el ruido de una imagen, así como simplificarla); más concretamente, usaremos el filtro de la media (una matriz 3x3, donde todos los elementos valen uno). La Figura 17 muestra el proceso de obtener el píxel de la imagen filtrada.

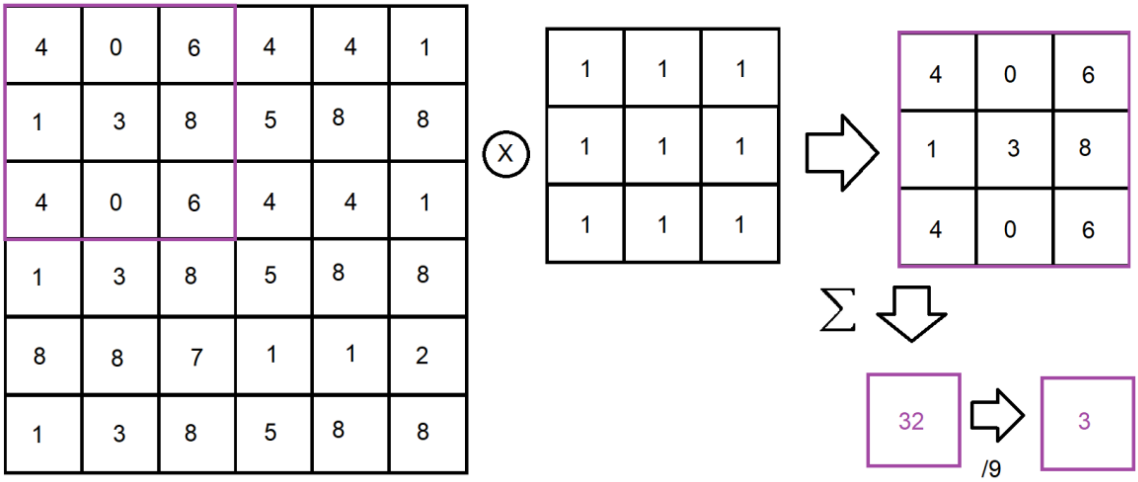

Figura 17: Obtención del primer píxel de la imagen filtrada de prueba

No obstante, este es solo el primer paso. A continuación, veremos qué ocurre si aplicamos el filtro en todo el fragmento de la imagen de la Figura 17. Además, debemos considerar los desplazamientos; al tener el fragmento de la imagen un tamaño de 6x6 y el filtro un tamaño de

3x3, el fragmento filtrado tendrá un tamaño de 4x4. La Figura 18 muestra el resultado de filtrar el fragmento de la imagen comentado.

3	4	5	4
3	4	6	5
5	4	4	3
5	5	5	5

Figura 18: Resultado de filtrar todo el fragmento seleccionado de la imagen de prueba

Para terminar este ejemplo, mostraremos cómo debe quedar la imagen de la Figura 16 tras aplicar nuestro filtro de suavizado en su totalidad. El resultado de aplicar este filtro nos muestra una imagen mucho más difuminada, como se puede apreciar a continuación.

Figura 19: Resultado de suavizar la imagen de prueba mediante un filtro de media

Este concepto de la convolución que acabamos de introducir es imprescindible para el tipo de redes neuronales que estamos comentado. Más concretamente, la convolución se implementa como una capa de convolución, donde hay varios filtros del mismo tamaño, los cuales se aplican sobre los valores de entrada. Sin embargo, para poder ser usadas en un

algoritmo de *Machine Learning*, los parámetros de los filtros son aprendidos por la red, pero el número de filtros y el tamaño de los mismo debe ser introducido por un humano. Debemos también destacar que hemos dejado sin comentar algunos conceptos de la convolución, como es el tratamiento de bordes, pero veremos esto más adelante, cuando hablemos de los tipos de capas.

Para terminar con las redes neuronales convolucionales y pasar al entrenamiento de nuestro modelo, veremos una serie de capas que, ya que ahora vamos a trabajar con un tipo de red neuronal más avanzado. Así pues:

- Capa de Convolución: Crea *kernels* de convolución para aplicarlos sobre la imagen. El conjunto de los resultados de cada *kernel* es la salida. Para no reducir el tamaño de la imagen, se pueden aplicar técnicas de *padding* (el *padding* extiende los bordes de una imagen de forma que, si tenemos una imagen de tamaño NxM y un filtro de tamaño RxS, que la imagen filtrada siga siendo de tamaño NxM.
- Capa de Activación: Necesarias para aumentar la complejidad de nuestra red. Usadas siempre después de las capas de convolución o de las capas tradicionales (*Dense*). En esta ocasión, vamos a destacar dos tipos, *ReLU* y *Softmax*. Por un lado, *ReLU*, unidad lineal rectificada, es una función del tipo f(x)=max(0,x), como vimos anteriormente. Por otro lado, tenemos la función *Softmax*, la cual nos permite obtener la posibilidad de que una imagen pertenezca a cada etiqueta de clase, como también vimos anteriormente.

Existen otros muchos tipos de capas (cómo las capas de *Pooling* o de *Batch Normalization*), pero, debido a la arquitectura de nuestro modelo, no necesitaremos usar esas otras capas.

2.4. Estado del Arte

Antes de comenzar a desarrollar nuestro proyecto, sería interesante contextualizar nuestro trabajo dentro del marco tecnológico actual.

2.4.1. Reescalar Imágenes

Un problema habitual del tratamiento digital de imágenes consiste en ajustar el tamaño de una imagen a unos nuevos valores. Ante esta situación y ante nuestro problema, veremos distintos algoritmos que intentan resolver este problema. Para realizar esto, además, nos valdremos de una imagen, la cual será nuestra base para mostrar los resultados de cada algoritmo.

Figura 20: Samus en Metroid

El primer algoritmo que vamos a probar se conoce cómo: algoritmo del vecino más cercano. La idea detrás de esta técnica es el de considerar, para un nuevo valor en la imagen resultado, el píxel más cercano de la imagen original. En este caso, duplicaremos el tamaño de la imagen base.

Otra técnica para aumentar el tamaño de una imagen es la interpolación lineal. Antes de entrar en profundidad con este método, debemos comentar que la interpolación es un proceso matemático que nos permite deducir resultados a partir de otros. En este caso, usaremos funciones continuas (que calcularemos con los valores que ya conocemos de la imagen original), para, así, obtener el valor de un nuevo píxel.

Figura 21: Samus en Metroid reescalado mediante el Vecino Más Cercano

Figura 22: Samus en Metroid reescalado mediante interpolación lineal

La otra técnica, antes de entrar en la super resolución, es también una función de interpolación; más concretamente, se trata de interpolación cúbica. Mientras que la

interpolación lineal se valía de funciones continuas, esta interpolación se basa en una ecuación de tercer grado. La siguiente imagen muestra el resultado obtenido mediante interpolación cúbica.

Con esto, habríamos visto tres maneras de distintas de reescalar una imagen. Sin embargo, aún no hemos visto la solución que vamos a desarrollar en este trabajo. Más adelante, en este mismo documento, mostraremos el resultado obtenido para nuestro modelo (en el siguiente apartado nos aproximaremos a él desde el punto de vista más teórico; ahora, solo queríamos mostrar los resultados de algoritmos convencionales para reescalar imágenes).

Figura 23: Samus en Metroid reescalado mediante interpolación cúbica

2.4.2. Modelos de Deep Learning

Anteriormente, vimos la base teórica subyacente a este proyecto; sin embargo, no llegamos a profundizar en lo que sería propiamente una red neuronal convolucional de super resolución, pues lo haremos en este apartado.

En primer lugar, debemos indicar que, pese a que el *Deep Learning* basado en redes neuronales convolucionales nos proporciona muy buenos resultados a la hora de trabajar con

imágenes, no todos los problemas merecen la misma aproximación. Por ejemplo, veamos distintos problemas que podemos resolver mediante redes neuronales convolucionales:

- Clasificación de imágenes: Imaginemos que queremos un algoritmo que, a través de una imagen de entrada, sea capaz de asignarle una etiqueta, cómo pasarle una foto de un electrodoméstico y esperar que sepa detectar que se trata de un frigorífico. Ante este tipo de problemas, desarrollaremos una red neuronal cuyo resultado sea el porcentaje de pertenencia a cada categoría posible, por lo que la arquitectura de la red estará marcada por este hecho.
- Generación de imágenes: En esta situación, queremos desarrollar una red neuronal que sea capaz de crear nuevas imágenes de algún tipo concreto. Esto se puede realizar mediante lo que se conoce cómo: red generativa antagónica [28]. Además, tal y cómo veremos en el análisis de viabilidad del siguiente capítulo, este tipo de red también podría haber servido para resolver el problema sobre el que gira este trabajo.
- Adaptación de estilos: Un uso moderno de las redes neuronales convolucionales es el adaptar una imagen a un estilo artístico. La técnica qué acabamos de describir se conoce cómo *Neural Style Transfer* [29] (en español: transferencia de estilo neuronal). La base de esta técnica es recorrer una red neuronal convolucional al revés.

Los ejemplos anteriores nos permiten hacernos una idea del estado actual de las redes neuronales convolucionales. Con esto, habríamos visto que, según el problema que debamos afrontar, existen distintas aproximaciones. En nuestro caso, queremos aumentar el tamaño de una imagen, hecho que marcará la arquitectura y el comportamiento de nuestro modelo.

Respecto a las redes neuronales convolucionales para super resolución, uno de los hechos más notorios es que la precisión que obtengamos durante el entrenamiento no será muy relevante, nos centraremos, principalmente, en los filtros que aprenda la red durante este proceso. Además, dado que nuestro principal objetivo son los filtros, no necesitamos optimizar la función de perdida en cada imagen que queramos reescalar.

Las bases para este tipo de redes fueron establecidas por Chao Dong; quién, junto con su equipo, realizó dos publicaciones sobre el tema. En la primera, describía lo que se conocería cómo *Super Resolution Convolutional Netowrk* [12]; mientras que, en la segunda publicación, mejoraría la versión anterior, con el modelo *Fast Super Resolution Convolutional Netowrk* [30].

3. Propuesta

En esta parte del trabajo, formalizaremos los objetivos que pretendemos conseguir al terminar este trabajo (sin embargo, a diferencia de cómo lo hicimos en el primer capítulo, esta vez nos centraremos en objetivos relacionados directamente con nuestro modelo).

Por otro lado, aclararemos algunas decisiones que se han tomado para diseñar e implementar nuestra red neuronal.

3.1. Definición de objetivos

Llegados a este punto del documento, hemos dejado claro cuál es nuestro objetivo general: diseñar y entrenar un modelo de *Deep Learning* que nos ayude a mejorar los gráficos de videojuegos antiguos. Sin embargo, en esta sección, desglosaremos este objetivo en puntos intermedios que debemos lograr.

- Creación del *Dataset*: Para poder cumplir con nuestro trabajo, una parte indispensable es la obtención de datos que nos permitan entrenar y probar nuestro modelo. Sin embargo, dado que queremos enfocar nuestro proyecto a los videojuegos, podría ser interesante que nuestro *Dataset* estuviera compuesta de imágenes relacionadas con ellos. En el análisis de viabilidad, veremos más sobre la creación del *Dataset*.
- Entrenamiento de la red: Una vez tengamos nuestro conjunto de datos preparado, diseñaremos una arquitectura para nuestro modelo y empezaremos el entrenamiento. Debemos destacar también que tendremos que ajustar distintos parámetros de nuestra red conforme vayamos entrenándola, para mejorar sus resultados
- Prueba y análisis de la red: Este es punto que mejor define el objetivo general del proyecto. Tras entrenar nuestra red, debemos comprobar que los resultados que nos ofrece sean aceptables y que, como mínimo, mejore los resultados de las técnicas convencionales que vimos anteriormente. Además, tendremos que analizar no solo los resultados, sino también el desempeño, de forma que podamos estudiar mejor los usos de la red que desarrollemos.

3.2. Análisis de Viabilidad

Nuestro primer paso será estructurar los problemas a resolver en tres partes: obtención del *Dataset*, definición del *Backend* de *Keras* [14] y elección de la arquitectura de nuestro modelo. De esta forma, procederemos a mostrar las distintas opciones planteadas para cada parte, junto con la solución tomada.

3.2.1. Dataset

Comenzaremos este análisis de viabilidad del proyecto con los datos que vamos a usar para entrenar y probar nuestra red. Sin embargo, primero debemos aclarar cómo vamos a trabajar con nuestro *Dataset*.

A la hora de entrenar nuestra red, queremos que sea capaz de relacionar una imagen con baja resolución de entrada con la misma imagen en mejor resolución. Por tanto, el *Dataset* que usemos deberá contar con la misma imagen representada de dos formas distintas.

Dada la naturaleza del *Dataset* que necesitamos, para entrenar nuestra red podríamos usar un *Dataset* cualquiera y aplicar sobre él las operaciones necesarias para obtener cada tipo de imagen. Sin embargo, nos gustaría usar imágenes que provengan de videojuegos.

Ante esta situación, aparece una decisión ante nosotros: usar un *Dataset* ya existente o crear nuestro propio *Dataset*. Por un lado, crear un *Dataset* ya existente nos supone una mayor comodidad, pues hay una gran cantidad de recursos disponibles que podemos usar. Pero, por otro lado, usar imágenes que vengan de videojuegos podría suponer una mejora para nuestra red, pues serían imágenes con un contexto más parecido a las que podríamos aplicar nuestro algoritmo

Finalmente, hemos decidido crear nuestro propio *Dataset*. Aunque veremos este proceso en el siguiente capitulo, podemos destacar que nos valdremos de técnicas de *Web Scrapping* para crearlo y que tendrá una longitud de unas cien imágenes (sin contar las nuevas que generaremos, con menos resolución, para entrenar la red).

3.2.2. Backend de Keras

Keras es la principal librería que usaremos en nuestro proyecto, pues cuenta con una gran cantidad de funciones relacionadas con el *Deep Learning*. Sin embargo, a la hora de usar *Keras* en nuestro proyecto, debemos configurar sobre que *Backend* se ejecutará.

Actualmente, podemos encontrar las siguientes opciones (aunque hemos decidido omitir *Microsoft Cognitve Toolkit*):

- *TensorFlow* [15]: Una de sus principales ventajas es que nos ofrece una serie de optimizadores ya preparados para que los usemos. Además, *TensorFlow* permite ejecución en múltiples gráficas. Por último, destacaremos que *TensorFlow* goza actualmente de más popularidad, por lo que podríamos encontrar más información al respecto actualizado.
- *Theano* [31]: Esta librería es más antigua que *TensorFlow* y nos ofrece, en algunos casos, mejores velocidades. Pese a que la comunidad actual sea menor que la de *TensorFlow*, tiene una cantidad de documentación superior.

Finalmente, hemos decidido decantarnos por *Tensorflow*, pues, tras investigar por Internet, hemos descubierto que se suele recomendar *TensorFlow* para trabajar en proyectos relacionados con imágenes.

Figura 24: Logo de Theano

3.2.3. Arquitectura de la Red

Aunque anteriormente ya hayamos aclarado que modelo vamos a implementar, en las fases tempranas de este proyecto (antes de la redacción de este documento), tuvimos que tomar una importante decisión, cómo cumplir con nuestro objetivo.

La idea de valernos de *Deep Learning* para solucionar el problema siempre estuvo presunte, pero, cuándo investigamos las opciones que había disponibles, acabamos teniendo que decidir entre las tres siguientes opciones:

- *Super Resolution Convolutional Network* [12]: Se trata del primer intento para resolver este tipo de problemas mediante redes neuronales convolucionales. Tiene una arquitectura simple y se centra en que los filtros (relativos a las capas de convolución) puedan hacer su trabajo, dejando de lado otros tipos de capas más complejas.
- *Super Resolution Residual Network* [32]: Tomando cómo inspiración el modelo anterior y las redes residuales [33] (un tipo más complejo de redes neuronales convolucionales, la cual tiene la capacidad de saltarse capas)
- *Super Resolution Generative Adversarial Network* [32]: A través de dos redes, una red generadora y otra red discriminadora, se puede usar la tecnología de las *GAN* [28] para aproximarse a nuestro problema. Sin embargo, esta arquitectura es mucho más compleja que las dos anteriores.

Finalmente, nos hemos decantado por la primera opción. Aunque las otras dos opciones suponen mejores resultados, son también más complejas. Por tanto, pensaos que, cómo el propósito de este trabajo es ver cómo puede servir la super resolución para mejorar gráficos de videojuegos antiguos, conseguir esto con la primera opción, más fácil de implementar, sería suficiente.

4. Diseño

Antes de comenzar a entrenar nuestro modelo de *Deep Learning*, debemos realizar una serie de configuraciones en nuestro entorno de trabajo. Más concretamente, tendremos que estar preparados para trabajar con nuestro *Dataset* y diseñar la arquitectura de nuestra red neuronal convolucional.

4.1. Arquitectura general

Antes de empezar con el entrenamiento de nuestro modelo de *Deep Learning*, aún debemos elegir qué arquitectura usaremos para dicho modelo. Anteriormente, ya comentamos que usaríamos la arquitectura de red neuronal de convolución de super resolución.

Usando como base el modelo *SRCNN* planteado por Dong et Al [12], construiremos la arquitectura de nuestra red neuronal convolucional. Uno de los motivos por lo que nos hemos decantado por esta arquitectura es su sencillez, lo que, cómo estudiantes, nos sirve perfectamente para introducirnos en el área, de la inteligencia artificial, de la super resolución.

En este apartado, nuestro objetivo es definir el funcionamiento general de nuestra red, así como todos los pasos y requisitos que debemos desarrollar para poder entrenar nuestra red. Para guiarnos, podemos servirnos del diagrama apreciable en la siguiente imagen.

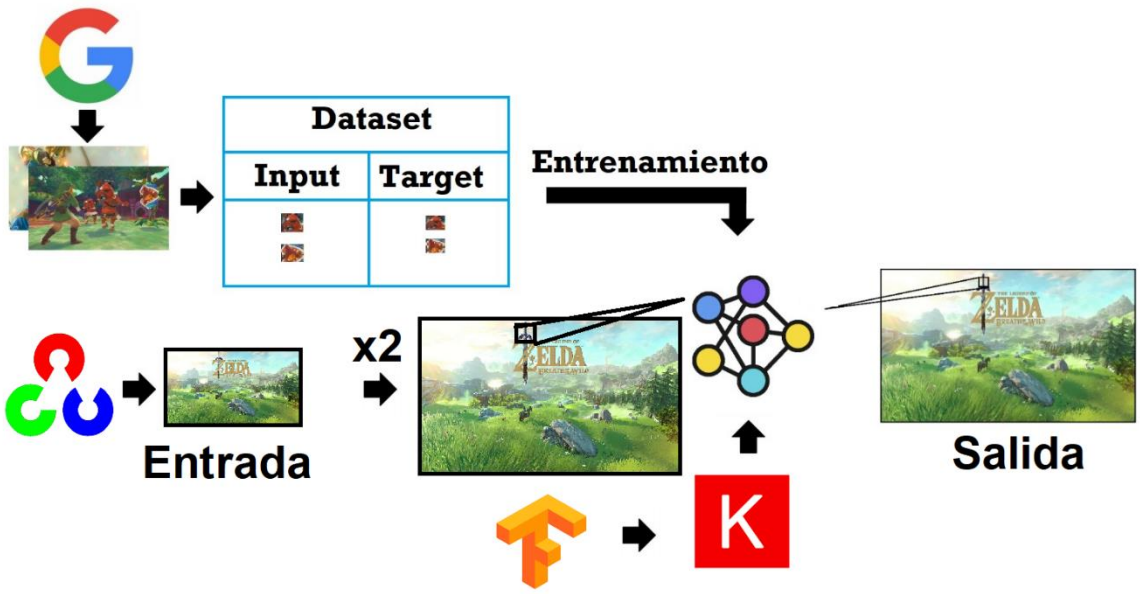

Figura 25: Diagrama de componentes del proyecto

Partiendo de la imagen anterior, podemos enumerar los pasos principales a continuación:

1. Obtención del *Dataset*: Consiste en la tarea por el cual nos procuraremos las imágenes necesarias para entrenar nuestro modelo. Necesitaremos crear pares de imágenes de baja resolución y de alta resolución, las cuales deberán representar lo mismo. No obstante, más adelante, en es mismo capítulo, veremos más sobre esta fase del proyecto.
2. Diseño de la red. Implementaremos en *Python* el modelo que vamos a usar, para que podamos llamarlo en las fases de entrenamiento y de prueba. Tal y como ocurría con la fase anterior, podremos encontrar más información al respecto en este mismo capítulo.
3. Entrenamiento de la red. Una vez hayamos finalizado las dos fases anteriores, podremos dedicarnos a entrenar el modelo de *Deep Learning*. Una explicación detallada sobre el entrenamiento de la red puede encontrarse en el siguiente capítulo; sin embargo, en el punto 4.4, podemos apreciar una introducción a esta fase, realizada mediante la recopilación de los hiperparámetros de nuestro modelo.
4. Prueba de la red. Cada vez que terminemos de entrenar nuestra red, podremos probarla y comparar los resultados obtenidos con los resultados obtenidos mediante métodos convencionales, así como con los resultados obtenidos en entrenamientos anteriores.

Para terminar este apartado, debemos destacar que las fases de entrenamiento y de prueba se repetirán varias veces, hasta que obtengamos resultados que nos parezcan satisfactorios.

4.2. Obtención del Dataset

Tal y cómo hemos visto anteriormente en este documento, uno de los pasos principales de nuestro proyecto consiste en la elaboración de un *Dataset*, el cual nos permitirá entrenar nuestra red neuronal.

En primer lugar, para obtener nuestro *Dataset*, vamos a hacer uso de contenedores de información en línea, cómo puede ser Google Imágenes [34]. Esta herramienta nos ofrece una gran cantidad de recursos a los que podemos acceder fácilmente. Sin embargo, para

agilizar el proceso, recurriremos a métodos de *Web Scrapping*, los cuales nos permitirán obtener los enlaces de unas cien imágenes.

A continuación, nos encargaremos de descargar las imágenes de la red e importarlas a nuestro entorno local. Además, durante este proceso, nos encargaremos de que las imágenes sean almacenadas en el formato correcto, con el título correspondiente y en el lugar adecuado. De esta forma, completamos nuestro primer paso en la obtención del *Dataset*, conseguir las imágenes originales.

No obstante, aún debemos desarrollar la parte más importante, obtener el *Dataset* final. Para nuestra siguiente tarea, volveremos a crear un programa en *Python*, el cual realizará los siguientes pasos:

1. Leemos, con la ayuda de *OpenCV*, cada una de nuestras imágenes (97 en total).
2. Sobre cada imagen:
    a. Empeoramos su resolución, para lo que nos valemos de interpolación bicúbica. Primero, reducimos la resolución de la imagen y, posteriormente, la devolvemos a su tamaño original; en ambos procesos nos valdremos de la interpolación bicúbica, mencionada anteriormente.
    b. Iteramos sobre la imagen, recorriendo cada paso una parte de ella. De esta forma, generaremos dos imágenes por sección: una qué pasaremos a nuestra red y otra que servirá de etiqueta. De forma similar al paso anterior, las imágenes representarán la misma idea, pero una con mayor calidad que la otra. Gracias a esto, ampliaremos nuestro *Dataset* de 97 imágenes iniciales a decenas de miles.
3. Una manera de agilizar el entrenamiento de nuestra red es servirnos del formato de datos jerárquico (más concretamente, en su quinta versión), también conocido como *HDF5* [35]. De esta forma, una vez tengamos el nuevo *Dataset*, incluiremos nuestras imágenes en dos ficheros de este tipo, uno para las imágenes de entrada y otro para las imágenes objetivo.

Para mejorar la explicación anterior, mostraremos a continuación un ejemplo de cómo generamos imágenes de peor calidad. La siguiente Figura muestra el proceso por el cual, a partir de una imagen, generamos, mediante un método convencional, una imagen más

pequeña, para, finalmente, revertirla a su tamaño original; este proceso conlleva una pérdida de calidad con respecto a la original.

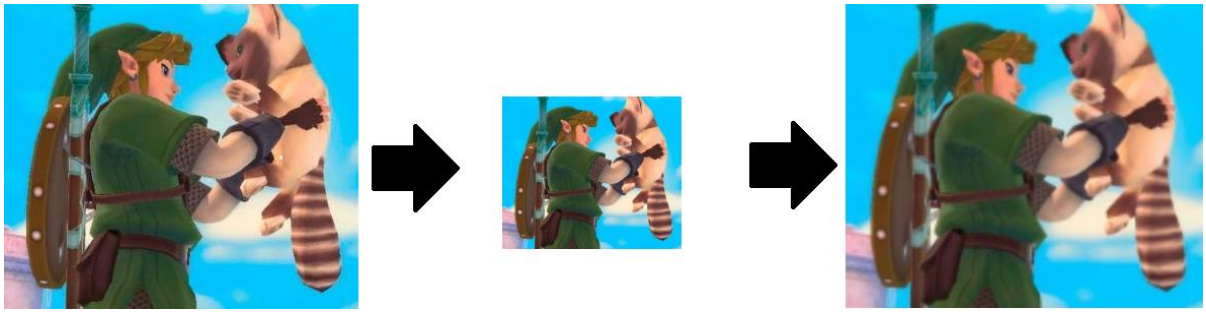

Figura 26: Proceso de creación de imágenes de baja resolución

Además, mostraremos visualmente otra de las actividades descritas anteriormente; más concretamente, la obtención de fragmentos de la imagen para aumentar el tamaño de datos de entrenamiento. En esta ocasión, iteraremos sobre nuestras dos imágenes y seleccionaremos un fragmento de cada una; es importante aclarar que serán de tamaños distintos, en base al tamaño de entrada de la red y al tamaño de salida de la misma. Para el siguiente ejemplo, usaremos los parámetros base: 33x33 píxeles de entrada y 21x21 píxeles de salida.

El proceso descrito en el párrafo anterior puede apreciarse en la imagen apreciable en la Figura 27. En dicha Figura, la parte superior representa a la imagen original, mientras que, por el contrario, la imagen inferior representa la imagen de peor calidad.

Repitiendo este proceso sobre todas las imágenes de nuestro *Dataset*, podremos aumentar los datos con los que contamos, teniendo la información estructurada en dos tipos: imágenes de entrada (baja resolución) e imágenes de salida (alta resolución).

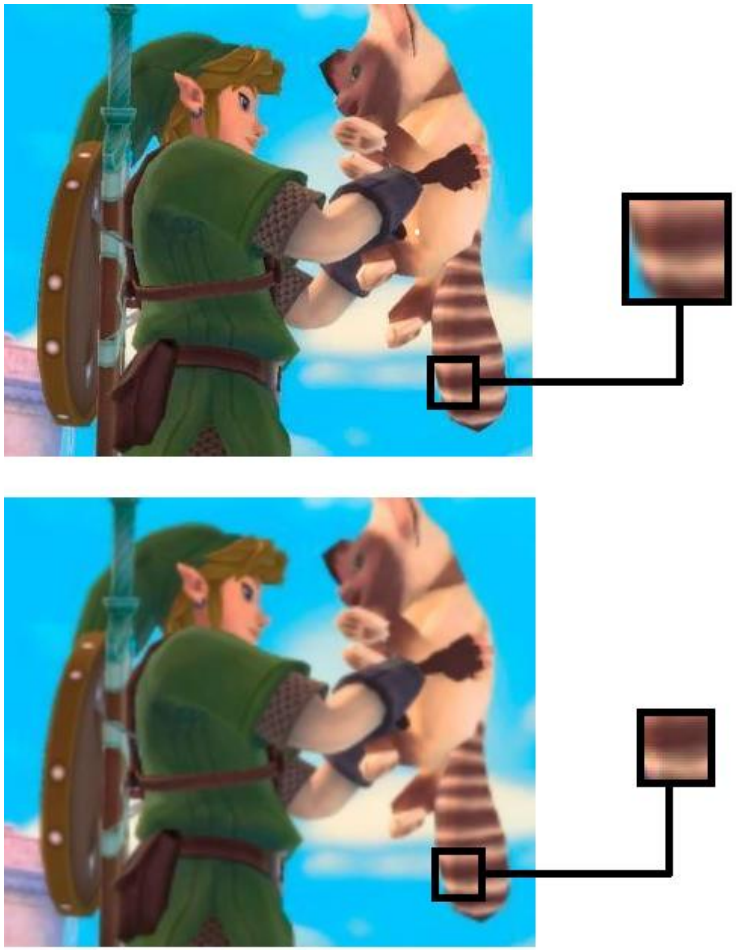

Figura 27: Extracción de pares alta resolución – baja resolución

4.3. Diseño de la Red

Tal y como dijimos anteriormente, la red que vamos a implementar es muy simple. Más concretamente, tendremos tres capas de convolución, cada una seguida por una capa de activación. Al contar únicamente con capas de convolución, podemos describir nuestra red neuronal cómo totalmente convolucional.

A continuación, se muestra la arquitectura que se ha decidido implementar para resolver nuestro problema de clasificación:

- Extracción y Representación de *Patches*: En esta primera parte de la red, introduciremos la imagen de baja resolución deseada, la cual tendremos que ajustar, mediante métodos tradicionales (cómo la comentada interpolación bicúbica), para introducirla en la red. De esta forma, con la imagen dentro de la red, procederemos a

aplicar la primera capa de convolución. Durante esta parte de la red, esperamos obtener información relacionada con lo que representa la imagen.

- Mapeo no lineal: Aplicaremos la segunda capa de convolución, con el objetivo de transformar información representada en un vector unidimensional a información bidimensional. Además, debemos destacar que usaremos tamaños de filtro 1x1, pues dota a nuestro resultado de mayor no linealidad.
- Reconstrucción: En esta última fase, haremos uso de la tercer, y última, capa de convolución. Mediante esta fase, reconstruimos la imagen (pues los resultados anteriores eran vectores), obteniendo así la imagen de alta resolución.

Con esto, habríamos visto las tres fases que forman la red a implementar. Sin embargo, aún tendríamos que ver cómo será la implementación inicial, especificando los valores de los parámetros de nuestra red (lo que podremos ver en el siguiente apartado).

Sin embargo, pensamos que imperativo añadir, en este apartado, cómo se aplica la red sobre una imagen para mejorar su resolución. Más concretamente, podemos distinguir los siguientes procesos:

1. Escalado de la imagen de entrada. Mediante un método convencional, aumentaremos la resolución de la imagen de entrada (hasta que se ajuste a las medidas deseadas).
2. Iteración sobre la imagen escalada. Recorreremos la imagen obtenida, extrayendo de ella sub-imágenes, las cuales se corresponderán con el tamaño de entrada de la red. Sobre el resultado que arrojen, iremos construyendo una nueva imagen de alta resolución. Esta fase se repetirá hasta que completemos la nueva imagen.

Para terminar este apartado, adjuntamos una imagen obtenida del artículo del que nos hemos basado para diseñar nuestro modelo, Dong et al. [12]. En esta imagen, podemos apreciar la arquitectura de la red de manera gráfica.

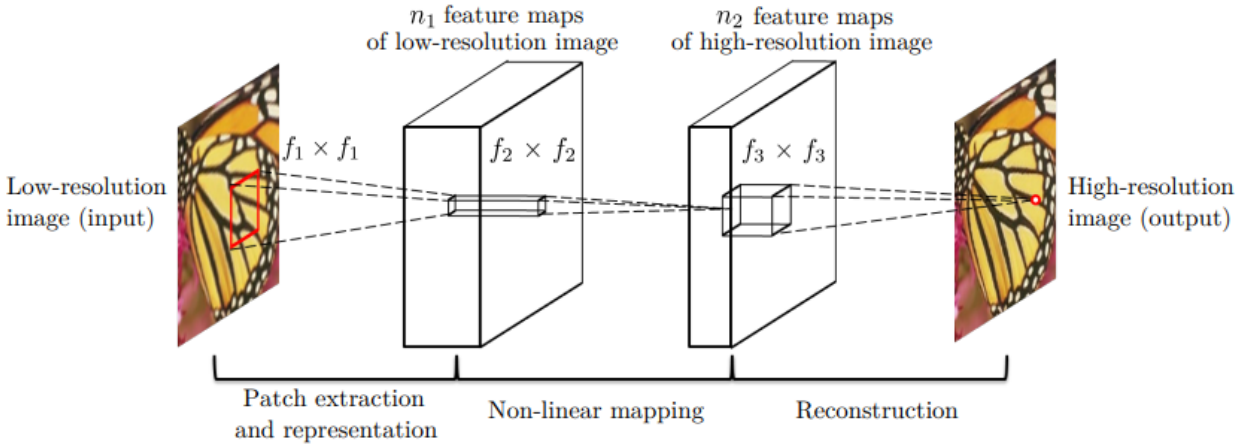

Figura 28: Arquitectura de SRCNN

4.4. Parámetros Destacados

En los apartados anteriores, se ha mencionado algunos parámetros y componentes, los cuales, si modificamos sus valores, pueden hacer variar los resultados de nuestra red. Por tanto, durante esta parte del cuarto capítulo, recopilaremos y comentaremos aquellas piezas de nuestro problema que puede servirnos para mejorar nuestros resultados en el siguiente capítulo.

- Tamaño de entrada: Para poder usar la red neuronal, nuestro primer paso debe ser definir un tamaño de entrada (de ahora en adelante, *InputSize*). Además, cuando apliquemos el modelo sobre una imagen, tendremos que iterar sobre ella, obteniendo porciones de este tamaño.
- Número de canales: Al introducir una imagen en la red neuronal, debemos especificar el número de canales que tiene la imagen. Además, durante la fase de reconstrucción, también debemos especificar este valor. No obstante, debemos indicar que este parámetro no se verá modificado, pues siempre trabajaremos con imágenes a color, pero pensamos que es importante mencionarlo, pues determina el valor de algunos parámetros de nuestra red.
- Tamaño de salida: Representa el tamaño de la imagen resultado de nuestro modelo (de ahora en adelante, *OutputSize*). El valor de este parámetro depende del tamaño de entrada y de las capas de la red (más concretamente, del tamaño de los filtros).

- Valor de desplazamiento: Para generar nuestras imágenes de entrenamiento, nos desplazaremos por las imágenes originales, obteniendo imágenes de los tamaños pertinentes. Sin embargo, para determinar cuántas imágenes nuevas podemos obtener de una, usaremos el valor de desplazamiento (de ahora en adelante, *Stride*). Cada vez que obtengamos una sub-imagen, nos desplazaremos la cantidad indicada por *Stride* para obtener la siguiente.
- Escala: Es uno de los componentes clave de nuestro proyecto, define el factor por el cual estamos redefiniendo el tamaño de la imagen. La influencia de este parámetro nos afectará durante la creación del *Dataset*, así como en la aplicación de la red.
- Método convencional de reescalado: Para aplicar nuestra red neuronal convolucional sobre una imagen y aumentar su resolución, debemos, de forma previa, aumentar su resolución mediante herramientas más básicas y sencillas. De esta forma, partiendo de una imagen a escalar y del valor de escala, aplicaremos el método elegido para obtener una imagen con el tamaño deseado.
- Parámetros de la primera capa: Esta capa, tal y como se mencionó anteriormente, se corresponde con el proceso de extracción y representación de *Patches*. A continuación, podemos encontrar los distintos parámetros de esta capa que podemos ajustar:

    - Número de filtros: Durante esta capa, los filtros nos ayudarán a obtener características relacionadas con la imagen de entrada (de baja resolución). Durante el resto de este documento, nos referiremos a este parámetro como *n1*.
    - Tamaño de filtros: Representa el tamaño de los filtros de esta capa. Durante el resto de este documento, nos referiremos a este parámetro como *f1*.
    - Función de activación: Una vez finalicen las actividades de convolución de esta capa, pasaremos el resultado obtenido (un vector de dimensión *n1*, donde cada elemento representa un filtro) a una función de activación.

- Parámetros de la segunda capa: Esta capa, tal y como se mencionó anteriormente, se corresponde con el proceso de mapeo no lineal. A continuación, podemos encontrar los distintos parámetros de esta capa que podemos ajustar:

  - Número de filtros: Durante esta capa, los filtros nos ayudarán a obtener características relacionadas con la imagen de entrada (de baja resolución). Durante el resto de este documento, nos referiremos a este parámetro como *n2*.
  - Tamaño de filtros: Representa el tamaño de los filtros de esta capa. Durante el resto de este documento, nos referiremos a este parámetro como *f2*.
  - Función de activación: Una vez finalicen las actividades de convolución de esta capa, pasaremos el resultado obtenido (un vector de dimensión *n2*, donde cada elemento representa un filtro) a una función de activación.

- Parámetros de la tercera capa: Esta capa, tal y como se mencionó anteriormente, se corresponde con el proceso de mapeo no lineal. A continuación, podemos encontrar los distintos parámetros de esta capa que podemos ajustar:

  - Número de filtros: Durante esta capa, los filtros nos ayudarán a obtener características relacionadas con la imagen de entrada (de baja resolución). Durante el resto de este documento, nos referiremos a este parámetro como *n3*.
  - Tamaño de filtros: Representa el tamaño de los filtros de esta capa. Durante el resto de este documento, nos referiremos a este parámetro como *f3*.
  - Función de activación: Una vez finalicen las actividades de convolución de esta capa, pasaremos el resultado obtenido (un vector de dimensión *n3*, donde cada elemento representa un filtro) a una función de activación.

- Tamaño de lotes: Si cada vez que pasásemos un dato por nuestro modelo, tuviéramos que actualizar los valores de nuestra red, entonces tardaríamos una considerable cantidad de tiempo más en comparación a si lo hiciéramos después de que hayan llegado varios datos. La idea que acabamos de presentar es el concepto que define un *Batch*, lo que podemos traducir al castellano como lote. Durante el resto del documento, nos referiremos a este parámetro como *BatchSize*.
- Número de épocas: Para poder aprender, nuestro modelo hará uso del *Dataset*. El número de épocas representa cuántas veces nuestro modelo usará los datos de

entrenamiento para ajustar sus valores. Durante el resto del documento, nos referiremos a este parámetro como *EpochsNumber*.

- Optimizador: Anteriormente, no mencionamos este componente de las redes neuronales, pese a ser imprescindible para dotarlas de la capacidad de aprender. La función de optimalización (u optimizador) nos permite ajustar los valores de una función, de forma que nos aproximemos más a los resultados que deseamos.
- Ratio de Aprendizaje: Este parámetro se encuentra altamente ligado al optimizador, pues será uno de los valores que sirvan para definirlo. La ratio de aprendizaje (o *Learning Rate*) nos indica la cantidad en la que se ajustarán los valores de nuestro modelo; es decir, una vez obtenida la dirección en la que debemos ajustar nuestros valores, la ratio de aprendizaje determinará el tamaño de este ajuste en dicha dirección.
- Función de perdida: Anteriormente, ya se introdujo este concepto relativo al entrenamiento de redes neuronales; por lo que, en esta ocasión, solamente debemos destacar que será uno de los parámetros que tendremos que considerar.

Partiendo de la lista anterior, podemos, a continuación, definir los parámetros iniciales de cada uno de sus elementos. Sin embargo, antes de comenzar, destacaremos que usaremos la misma función de activación en todas las capas, por lo que, en la siguiente tabla, nos referiremos a ellas cómo Función de Activación de manera global.

Hiperparámetro	**Valor Base**
InputSize	33x33
Channels	3
OutputSize	21x21
Stride	14
Escala	2
Método convencional de reescalado	Interpolación Bicúbica
N1	64
N2	32

N3	3
F1	9x9
F2	1x1
F3	5x5
Función de Activación Capa-1	Unidad Lineal Rectificada (*ReLU*)
Función de Activación Capa-2	Unidad Lineal Rectificada (*ReLU*)
Función de Activación Capa-3	Unidad Lineal Rectificada (*ReLU*)
BatchSize	128
EpochsNumber	20
Optimizador	*Adam*
Ratio de Aprendizaje	0.001
Función de Perdida	Error cuadrático medio

Mesa 6: Configuración inicial de los hiperparámetros

No obstante, antes de poder continuar con otros aspectos del documento, debemos hacer un breve inciso y facilitar información relativa al optimizador, así como a la ratio de aprendizaje escogidos. Dado que, como se mencionó anteriormente, no incluimos al optimizador en la explicación de las redes neuronales convolucionales, tampoco introdujimos ningún ejemplo, por lo que la función de optimalización que hemos indicado como inicial, *Adam* [36], no ha sido explicada.

Respecto al optimizador, simplemente destacaremos que *Adam* es un optimizador basado en el método del descenso del gradiente estocástico, el cual, además, incorpora el uso de momentos (sin embargo, la explicación de estos conceptos vamos a dejarlos fuera de los propósitos de este proyecto).

Por último, respecto a la ratio de aprendizaje, debemos indicar que hemos añadido un factor de decadencia, por lo que la ratio de aprendizaje se reducirá cada época en el valor obtenido de dividir la ratio de aprendizaje entre el número de épocas.

4.5. Lenguajes de programación y librerías

Aunque anteriormente ya vimos las herramientas más importantes que íbamos a usar, en el segundo capítulo, aprovecharemos esta sección para mostrar todas las librerías de *Python* que usamos en nuestro proyecto; además, indicaremos su versión.

Librerías	**Versión**
Keras [14]	2.4.3
Tensorflow [15]	2.4.1
OpenCV [16]	4.5.1.48
MatPlotLib [18]	3.4.1
H5py [37]	2.10
Numpy [17]	1.19.5

Mesa 7: Versiones de las librerías de Python

La mayoría de estas librerías ya fueron comentadas anteriormente, por lo que hemos decidido indicar solamente la versión que hemos usado de ellas durante la elaboración de este trabajo.

Por último, aportaremos información relacionada con otras herramientas empleadas y que, aún, no ha sido mencionada:

- La versión de *Python* [13] que estamos usando es la 3.8.
- Tal y como mencionamos anteriormente, para gestionar y desarrollar nuestro proyecto estamos usando el *IDE PyCharm* [19]; más concretamente, estamos usando *PyCharm Community Edition 2020*.3.2.

5. Entrenamiento

A lo largo de este capítulo, terminaremos de hacer las preparaciones necesarias en nuestro entorno para poder entrenar nuestra red neuronal convolucional. Además, una vez terminadas las últimas preparaciones mencionadas, documentaremos el proceso de entrenamiento del modelo de *Deep Learning*, representado los cambios realizados, así como los resultados obtenidos.

5.1. Métricas de Calidad de Imágenes

Antes de comenzar con el entrenamiento de nuestro modelo, debemos definir una función que nos permita evaluar la calidad de nuestras imágenes, para evitar depender (de forma exclusiva) de nuestro criterio y aportar objetividad al análisis de los resultados que obtengamos.

Afortunadamente para nosotros, no tenemos que preocuparnos por diseñar una métrica específica para nuestros fines desde cero; en su lugar, podemos optar por valernos de la métrica que usan Dong et al en su trabajo [12], esta métrica es "*Peak signal-to-noise ratio*" (en castellano, ratio máxima de señal-a-ruido), más conocida por su abreviación *PSNR*.

Esta métrica destaca por sus usos en el campo de la informática del tratamiento de imágenes; más concretamente, suele utilizarse para medir la calidad de los resultados de procesos de reconstrucción de imágenes. El objetivo de *PSNR* es medir cómo el ruido afecta al resultado.

Para calcular el *PSNR*, debemos seguir la siguiente fórmula: $PSNR = 10 * log_{10}(\frac{MAX_I^2}{MSE})$. Si desglosamos la función anterior, podemos destacar los siguientes aspectos:

- El resultado de la función se expresará en decibelios.
- Debido al puto anterior (comparamos el nivel de pico con el nivel de ruido), debemos expresar los resultados en escala logarítmica.
- MAX_I^2 representa el cuadrado del valor máximo que puede tener un píxel.
- MSE denota el error cuadrático medio, durante el segundo capítulo de este documento (en la tercera sección) se introdujo este concepto como función de

perdida. De esta forma, podemos recordar la formular del error cuadrático medio como: $MSE(y, y') = \frac{\sum_{i=1}^{n}(y_i - y'_i)^2}{n}$, donde:

- y representa una imagen.
- y' representa otra imagen.
- n denota el tamaño de las dos imágenes anteriores, resultado de multiplicar su número de filas por su número de columnas.

Partiendo de estas consideraciones, crearemos un programa en *Python* al que podamos pasarle dos imágenes para que calcule el *PSNR*.

Partiendo del programa anterior, crearemos un conjunto de imágenes de prueba (formado por siete imágenes), sobre el que aplicaremos nuestra red neuronal. Sobre el resultado final, calcularemos la media, lo que usaremos como medida de calidad.

Figura 29: Componente de las imágenes de prueba de *PSNR*

Por último, para analizar mejor los resultados de nuestra red, previamente, calcularemos la *PSNR* para nuestras imágenes de prueba, habiendo sido procesadas por interpolación bicúbica. A continuación, podemos ver el resultado obtenido.

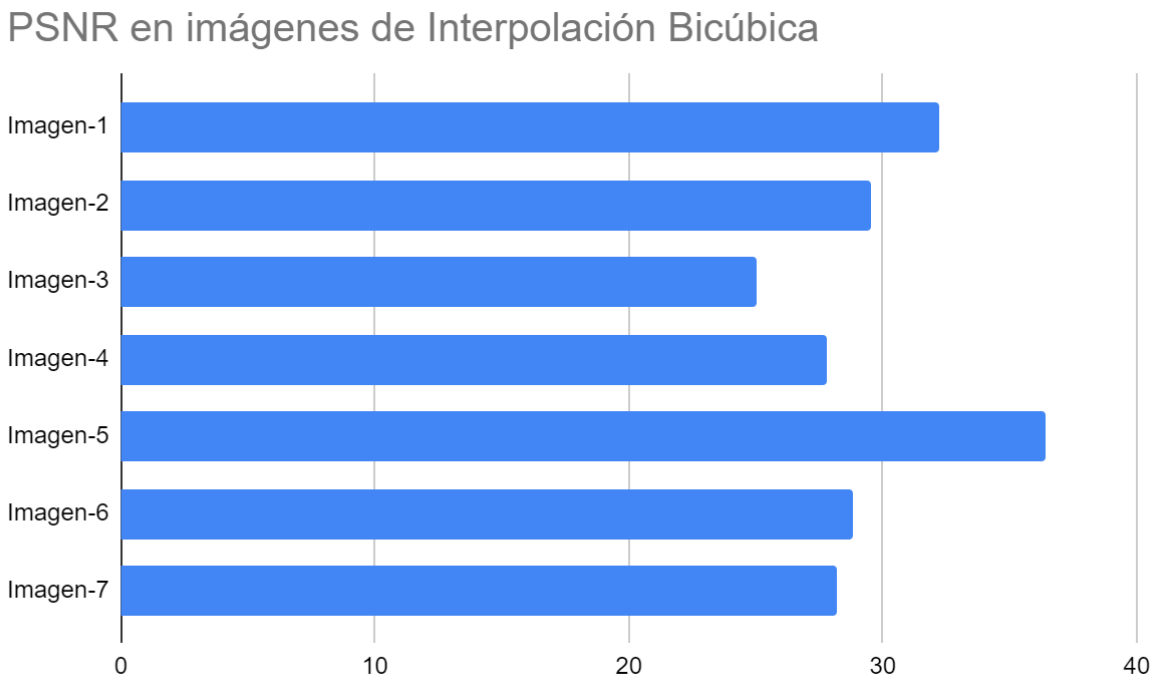

Figura 30: Resultados de PSNR para las imágenes de prueba

5.2. Entrenamiento

Durante esta sección del quinto capítulo, nos centraremos en entrenar nuestra red neuronal. Sin embargo, no trataremos únicamente este punto con el objetivo de mejorar los resultados, también veremos cómo algunos parámetros afectan a otros aspectos del resultado final.

5.2.1. Primer Entrenamiento

Tras el apartado anterior, ya tendríamos todos los requisitos necesarios para comenzar a entrenar nuestra red. En primer lugar, partiremos de la configuración que se especificó al final del capítulo anterior.

De esta forma, comenzaremos nuestro proceso de entrenamiento de la siguiente manera:

1. A partir de nuestras imágenes originales, crearemos un *Dataset* que se ajuste a nuestros valores de: escala (2), *InputSize* (33), *OutputSize* (21) y *Stride* (14). Además, usaremos interpolación bicúbica para escalar las imágenes.

Figura 31: Proceso de creación del *Dataset*

2. Tras obtener nuestro *Dataset*, el cual ha acabado compuesto por 390.434 pares de imágenes, convertiremos la información a archivos *HDF5* [35], lo que agilizará el paso de datos a la red, acelerando el proceso de entrenamiento.
3. Crearemos la red con los parámetros pertinentes. Debemos definir el número de filtros que la red aprenderá en cada capa: *N1* (64), *N2* (32) y *N3* (3). Debemos definir el tamaño de los filtros de cada capa: *F1* (9), *F2* (1) y *FR* (5), los cuales, partiendo de nuestro *InputSize*, deben permitirnos obtener una imagen del *OutputSize*; en el caso actual, con un *InputSize* de 33 y considerando que la primera y la última capa tienen un tamaño de filtro de 9 y 5, obtenemos el resultado esperado, 23, el tamaño actual del *OutputSize* Además, tendremos que considerar el número de canales de nuestras imágenes, en este primer entrenamiento, estamos usando imágenes de tres canales (el clásico *RGB*: rojo, verde y azul). Por último, nos encargaremos de que la función de activación sea la establecida para el primer entrenamiento, *ReLU*.
4. Con la arquitectura de la red creada, nuestro siguiente paso es terminar de ajustar algunos de sus parámetros. En primer lugar, definiremos el optimizador que vamos a usar, el cual será *Adam* [36] y fijaremos una ratio de aprendizaje de 0.001; además, le añadiremos un factor de decadencia, para que la ratio de aprendizaje varie durante el entrenamiento. A continuación, estableceremos la función de perdida, la cual será el error cuadrático medio.
5. Con los parámetros anteriores definidos, podemos comenzar el entrenamiento de nuestra red. De esta forma, compilaremos nuestro modelo, definiremos los datos de entrenamientos como los archivos *HDF5* del segundo paso, usaremos *BatchSize* para definir los datos que procesará por época, definiremos el número de épocas (en este

primer entrenamiento, 10) y realizaremos otros ajustes para que la red nos muestre información de cada época, se aleatorice la entrada de datos y comprueba cual fue la mejor época (la cual hemos decidido definir cómo aquella época dónde se alcance la mínima perdida), con el fin de almacenarla, de forma separada al modelo.

6. Una vez acabe el entrenamiento, tendremos almacenado el modelo y la mejor época que se obtuvo durante este proceso. Además, calcularemos una gráfica que nos muestre la evolución de los resultados obtenidos por nuestra función de perdida, así como su evolución a lo largo de las veinte épocas.

Tras seguir el proceso que acabamos de describir, habríamos acabado el primer entrenamiento de nuestro modelo de *Deep Learning*. En este caso, hemos obtenido la siguiente gráfica.

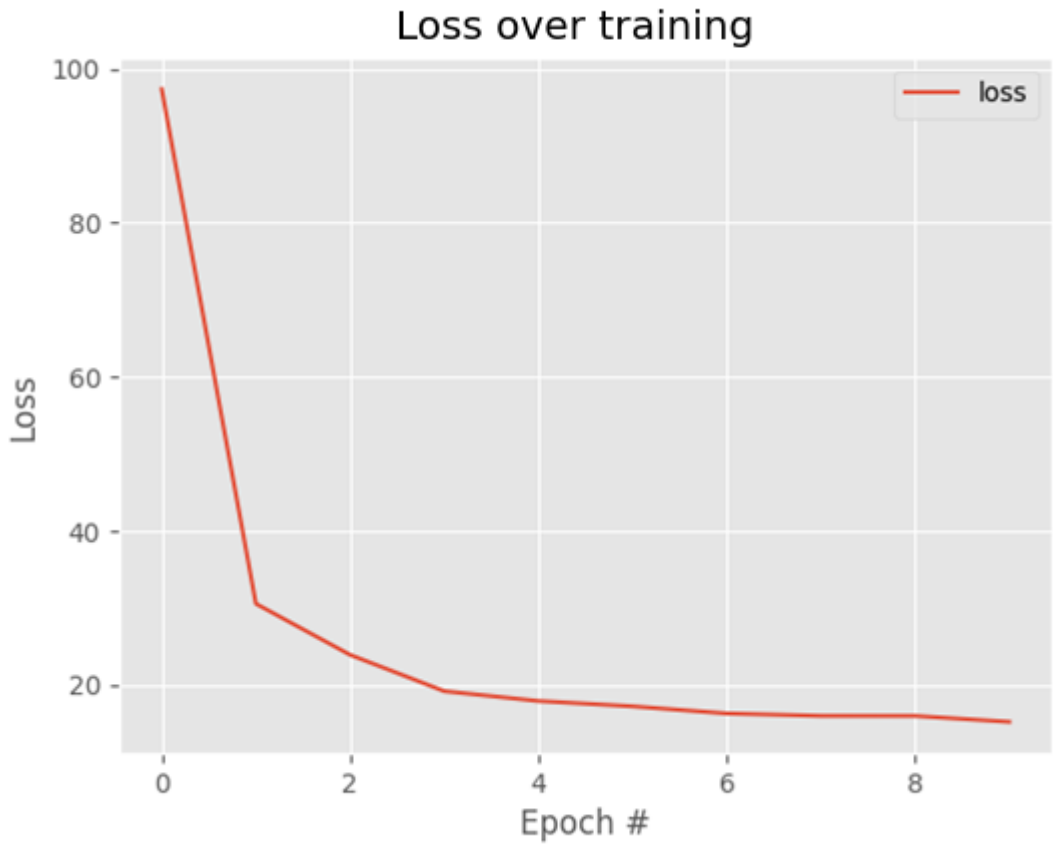

Figura 32: Gráfica de la evolución de la pérdida para el primer entrenamiento

La Figura-32 muestra la evolución del resultado de la función de perdida durante las épocas sobre las que entrenamos nuestro modelo. Más concretamente, nuestra red intenta ser capaz

de relacionar una imagen de baja resolución con su correspondiente de alta resolución, por lo que los resultados de la gráfica muestran la capacidad de emparejar los pares de imágenes (entrada-salida) de forma correcta.

No obstante, anteriormente desarrollamos una métrica que nos permitiese medir la calidad de nuestra imagen, por lo que, para terminar este primer entrenamiento, compararemos los resultados obtenidos para una imagen cuya resolución ha aumentado mediante interpolación bicúbica, junto con una imagen cuya resolución haya sido aumentada mediante el modelo que acabamos de entrenar.

Para cumplir este último objetivo, escalaremos las imágenes por la cantidad correspondiente (en este caso, estamos considerando un valor Escala de dos, por lo que duplicaremos su tamaño). A continuación, retornaremos las imágenes al tamaño de la imagen original, lo que nos permitirá calcular la *PSNR*; primero, entre la imagen original y la imagen obtenida mediante interpolación bicúbica; y, segundo, entre la imagen original y la imagen obtenida a través de nuestra red neuronal convolucional, sirviéndonos del método que implementamos anteriormente.

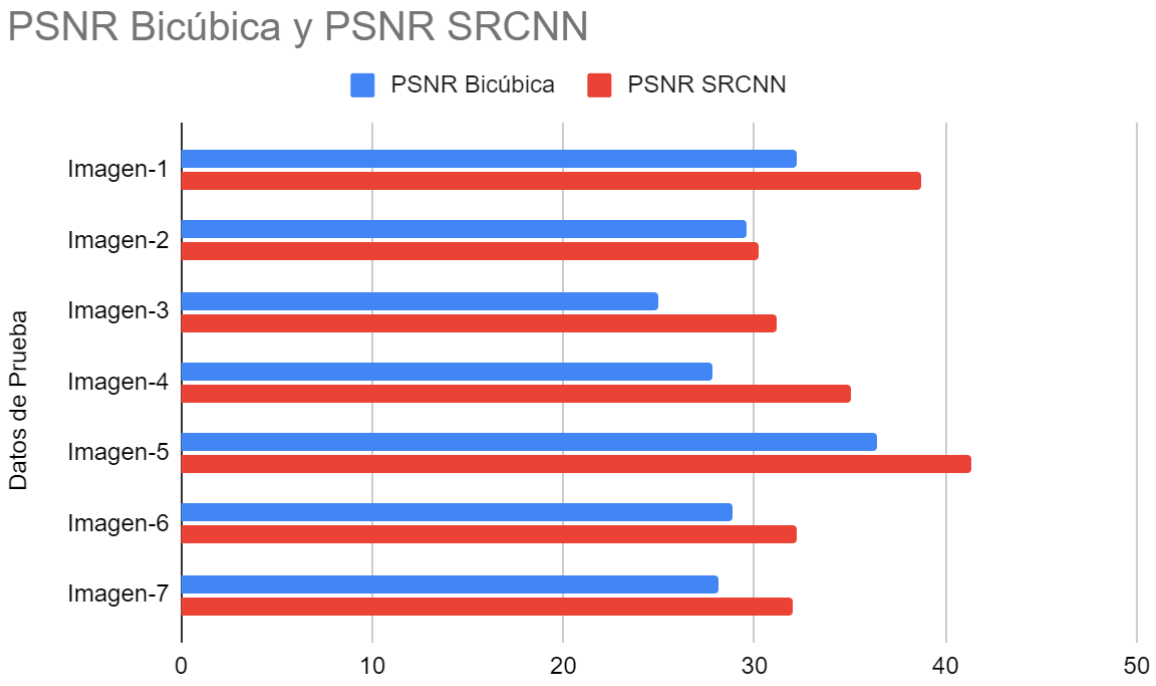

Figura 33: Comparativa entre el *PSNR* para imágenes bicúbicas y las del modelo del primer entrenamiento

Gracias a la gráfica anterior podemos apreciar que, pese a haber entrenado la red para solo diez épocas, somos capaces de mejorar los resultados obtenidos por interpolación bicúbica. Sin embargo, para entender mejor los resultados, las siguientes figuras nos permiten comparar los resultados obtenidos por cada uno de los métodos (para la misma imagen). Debemos aclarar que la siguiente imagen no forma parte de las siete imágenes sobre las que calculamos el *PSNR*.

Figura 34: Imagen obtenida mediante interpolación bicúbica

Figura 35: Imagen obtenida mediante el modelo del primer entrenamiento

Sin embargo, la red que hemos implementado cuenta con un pequeño inconveniente, las imágenes que genera pierden un poco de información; más específicamente, los bordes de la imagen original se ven recortados. En la siguiente figura, se puede apreciar la misma imagen que en la Figura 36, pero con el añadido de los bordes perdidos (aunque, en este caso específico, al contar la imagen con bordes negros verticales, la información relativa al videojuego se ha perdido únicamente en los bordes horizontales).

Figura 36: Información perdida de la imagen obtenida mediante *SRCNN*

En la imagen anterior, los bordes rojos representan la información que se ha perdido. Esto se debe a que, en los bordes horizontal superior y vertical izquierdo, sobre los que empezamos cada bucle mediante los que iteramos sobre la imagen, se pierde información debido al ajuste entre el tamaño de entrada y el tamaño de salida de la red neuronal. Por otro lado, los otros dos bordes, horizontal inferior y vertical derecho, pierden más información que sus opuestos, esto se debe a que, a parte de la diferencia entre el tamaño de entrada y el tamaño de salida, debemos también considerar que no toda imagen puede ser descompuesta de forma perfecta, en base al tamaño de entrada. Sin embargo, estos problemas del algoritmo de reconstrucción serán tratados más adelante.

5.2.2. Continuando con el Entrenamiento

En el punto anterior entrenamos por primera vez nuestro modelo; sin embargo, solo lo hemos entrenado para diez épocas, por lo que entrenarlo para un mayor número de épocas podría mejorar los resultados obtenidos.

En esta ocasión desarrollaremos un nuevo programa en Python que nos permita continuar entrenando un modelo ya creado. Podemos desglosar su funcionamiento como la siguiente lista de pasos:

1. Leer los datos de entrenamiento. De forma paralela al código que usamos para entrenar nuestro modelo, debemos asegurarnos de cargar los archivos *HDF5* que contiene nuestro *Dataset*.
2. Pasándole la ruta dónde se encuentra nuestro modelo almacenado en nuestro dispositivo, podemos cargar un modelo, tal y cómo haríamos para probar su funcionamiento.
3. Con el *Dataset* y el modelo preparado, pasaremos a compilar nuestro modelo. En este caso, continuaremos con la configuración que teníamos. De este modo, especificaremos los siguientes parámetros:

    3.1. Número de Épocas: Repetiremos el entrenamiento para diez épocas, lo que, contando las diez del primer entrenamiento, nos dan un total de veinte.
    3.2. Pasos por Época: El valor de este parámetro será el resultado de dividir el número de imágenes por el tamaño de lotes (128).
    3.3. Configuraciones Secundarias: Nos encargaremos de definir la verbosidad del proceso de entrenamiento, la aleatoriedad de la gestión de los datos y de un punto de guardado para la época con mejores resultados.

4. Una vez acabe el entrenamiento, tendremos almacenado el modelo y la mejor época que se obtuvo durante este proceso. Además, calcularemos una gráfica que nos muestre la evolución de los resultados obtenidos por nuestra función de perdida, así como su evolución a lo largo de las diez épocas.

Nuevamente, tras terminar este entrenamiento, mostraremos la gráfica que hemos guardado, con el objetivo de que podamos compararla con la obtenida en el primer entrenamiento.

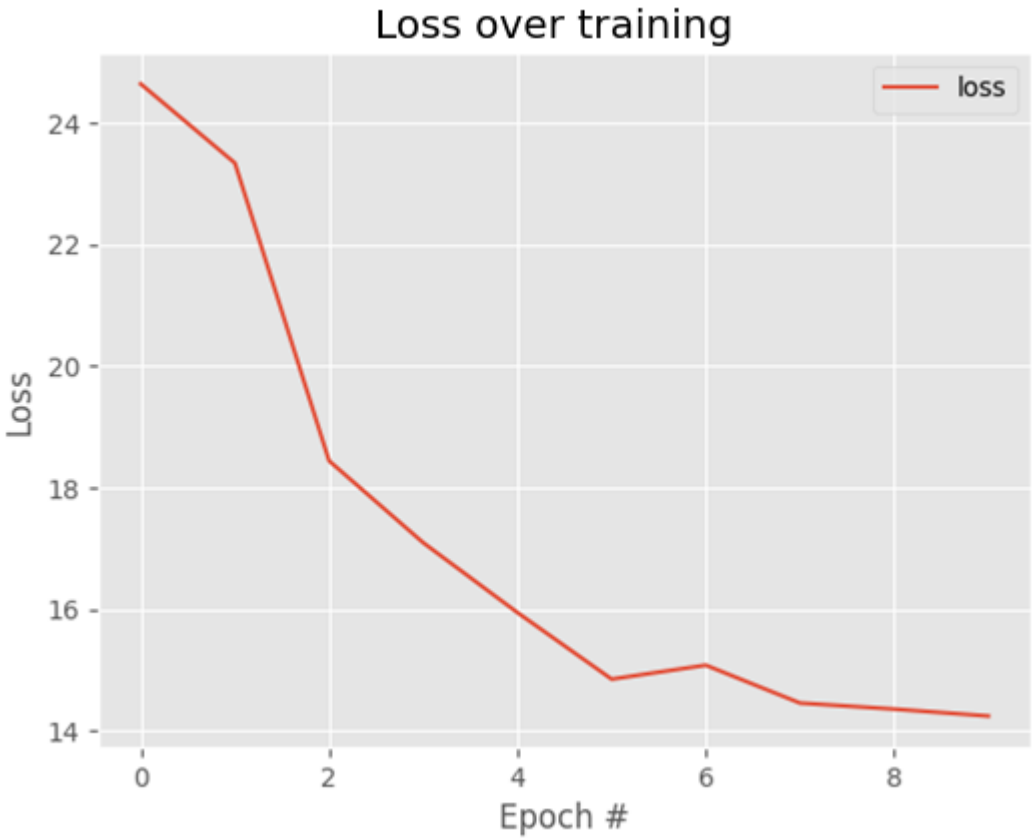

Figura 37: Gráfica de la evolución de la pérdida para el segundo entrenamiento

La gráfica anterior nos muestra resultados similares a los de la gráfica del primer entrenamiento; sin embargo, en esta ocasión hemos conseguido comenzar, como era de esperar, con una pérdida más baja. Además, hemos logrado reducir, aunque no mucho, la pérdida final, lo que nos indica que, si entrenáramos más nuestro modelo, podríamos seguir mejorando los resultados.

Nuestro siguiente paso será aplicar *PSNR* sobre nuestras imágenes de prueba con este modelo y comparar el resultado obtenido con el que obtuvimos anteriormente para la primera versión de nuestro modelo.

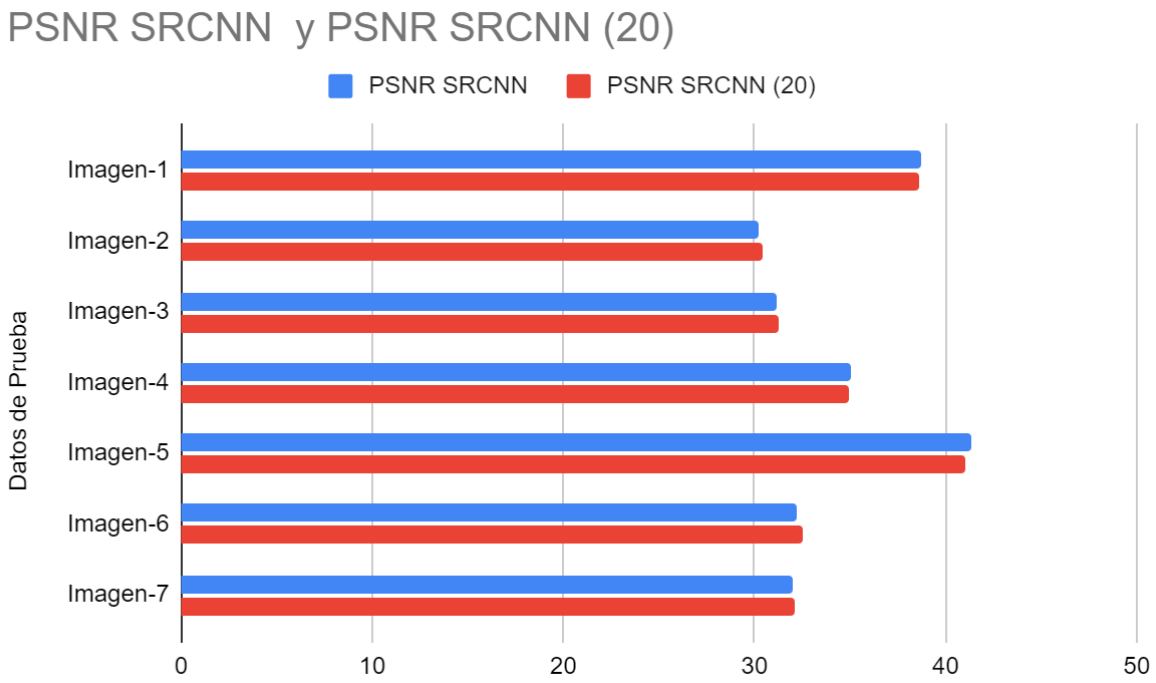

Figura 38: Comparativa entre el *PSNR* para imágenes bicúbicas y las del modelo del segundo entrenamiento

Si nos fijamos en los datos de *PSNR* obtenidos, podemos apreciar que no hay mucha variación. Además, en algunos casos, los resultados obtenidos han empeorado un poco; sin embargo, en la mayoría de los casos, los resultados han mejorado, lo que se refleja en que la nueva media (34.42 dB) es ligeramente mayor que la anterior media (34.37 dB).

De esta forma, podemos concluir que aún existe margen de mejora para nuestro modelo y, aunque ya nos reporta buenos resultados con un número reducido de épocas, exponerle a un proceso de entrenamiento más extenso mejorará los resultados. Sin embargo, estos dos hechos no son proporcionales, lo que significa que para que la diferencia sea significativa tendremos que recurrir a un mayor número de épocas.

Por último, mostraremos una versión de la imagen visible en la Figura 35, pero conseguida mediante el modelo tras ser entrenado otras diez épocas.

Figura 39: Imagen obtenida mediante el modelo del segundo entrenamiento

5.2.3. Aumentando el Número de Filtros

Durante este apartado, vamos a entrenar por segunda vez nuestra red neuronal. Sin embargo, en esta ocasión, cambiaremos uno de los hiperparámetros de nuestro modelo: el número de filtros.

Con el propósito de comprobar cómo puede afectar a nuestro proyecto aumentar el número de filtros, duplicaremos los tamaños actuales de *N1* y *N2*. Por tanto, *N1* (anteriormente, representaba 64 filtros) se actualizará a 128 filtros; mientras que, por otro lado, *N2* (anteriormente, representaba 32 filtros), se actualizará a 64 filtros.

Con respecto al resto de parámetros, los dejaremos inmutados. No obstante, si debemos aclarar que, tal y cómo hicimos anteriormente, lo entrenaremos en dos tandas de diez épocas cada una.

Si, tras aplicar los cambios comentados, repetimos el proceso de entrenamiento, tal y como fue descrito en el punto 5.2.1, volveremos a obtener un modelo y un gráfico con los resultados de la función de perdida, la cual podemos ver en la siguiente imagen.

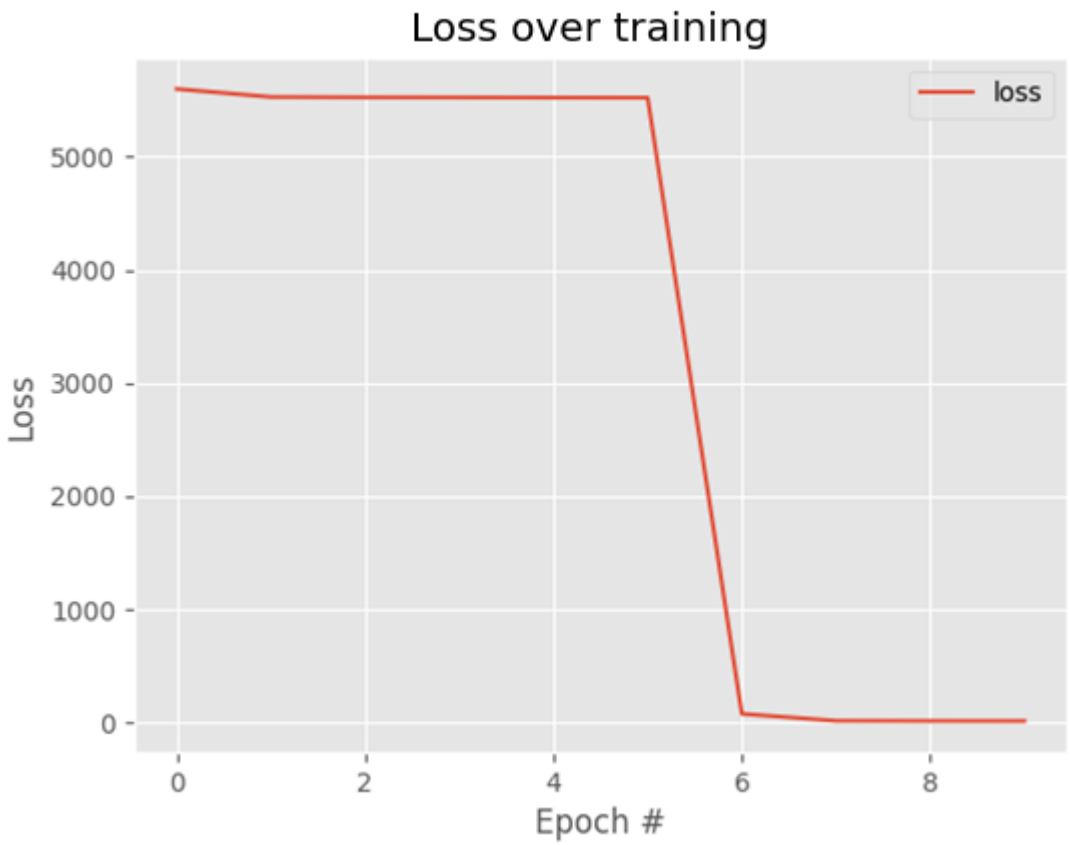

Figura 40: Gráfica de la evolución de la pérdida para el tercer entrenamiento

La gráfica anterior nos muestra un proceso distinto al del caso anterior, la función de perdida alcanza valores que ronda los 6.000, pero, en la sexta época, realiza una bajada drástica hasta valores cercanos al 100, los cuales se van reduciendo de forma más suave. Para la última (décima) época, acabaremos alcanzando valores cercanos al 30, lo que supone un peor resultado que para el anterior, pero pensamos que, debido al mayor tamaño de filtros, la red actual necesita de un mayor número de épocas para dar mejores resultados.

De esta forma, volveremos a entrenar nuestro modelo para otras diez épocas. Tras este segundo entrenamiento, cuya gráfica puede apreciarse en la siguiente ilustración, hemos conseguido reducir el resultado de la función de pérdida, el cuál ha bajado de (aproximadamente) 35 a 15, resultado similares al del caso anterior; además, esta reducción se ha realizado de forma menos abrupta.

Figura 41: Gráfica de la evolución de la pérdida para el cuarto entrenamiento

Debido a nuestra anterior teoría, de que necesitaríamos más épocas para mejorar el resultado, respecto al modelo anterior; para terminar este subapartado, volveremos a entrenarlo para otras diez épocas, pues ahora arroja resultados similares al anterior. A continuación, se muestra una gráfica con los resultados obtenidos; en ella, podemos observar

Figura 42: Gráfica de la evolución de la pérdida para el quinto entrenamiento

Con esto, solo nos quedaría comprobar el funcionamiento de nuestro nuevo modelo. Para hacer esto, nos valdremos de nuestra métrica, la cual podremos comparar con el resultado de nuestro último modelo.

En la gráfica apreciable en la Figura 43, podemos comprobar que el resultado para la *PSNR* ha mejorado, no solo de forma general (hemos pasado de una media de 34,42 dB a una media de 34,61 dB); además, a diferencia de con el caso anterior, hemos conseguido mejorar los resultados para cada una de nuestras imágenes de prueba.

Con esto, habríamos terminado el entrenamiento para nuestra red con un número de filtros que dobla al original (en las capas adecuadas). Para terminar de analizar el impacto del incremento de estos hiperparámetros, tendríamos que continuar entrenando nuestra red anterior y la actual, comparando los resultados alcanzados para el mismo número de épocas.

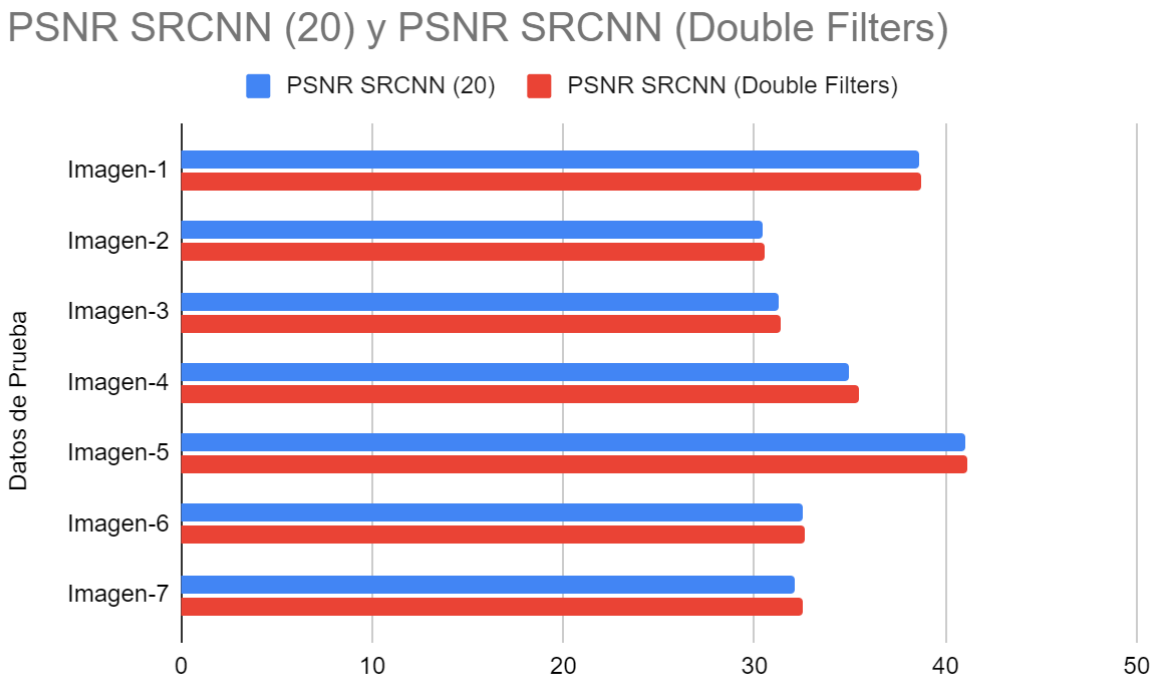

Figura 43: Comparativa entre el *PSNR* para imágenes bicúbicas y las del modelo del quinto entrenamiento

Por último, mostraremos una versión de la imagen visible en la Figura 35, pero conseguida mediante esta última versión de nuestro modelo.

Figura 44: Imagen obtenida mediante el modelo del quinto entrenamiento

5.2.4. Entrenamiento Final

En los puntos anteriores, hemos visto las bases del entrenamiento, así como las primeras etapas de este proceso. Sin embargo, en este apartado vamos a resumir todos los cambios que hemos realizado hasta obtener una versión final de nuestro modelo, pues, de otro modo, nos extenderíamos demasiado en este documento.

Dado que contamos con una tabla que resume los hiperparámetros a ajustar, volveremos a mostrar esta tabla, pero, ahora, actualizada con los valores definitivos de cada uno de sus elementos.

Hiperparámetro	**Valor Base**
InputSize	33x33
Channels	3
OutputSize	21x21
Stride	14
Escala	2
Método convencional de reescalado	Interpolación Bicúbica
N1	128
N2	64
N3	3
F1	9x9
F2	3x3
F3	5x5
Función de Activación Capa-1	Unidad Lineal Rectificada (*ReLU*)
Función de Activación Capa-2	Unidad Lineal Rectificada (*ReLU*)
Función de Activación Capa-3	Lineal
BatchSize	128
EpochsNumber	20
Optimizador	*Adam*

Ratio de Aprendizaje	0.003
Función de Perdida	Error cuadrático medio

Mesa 8: Configuración final de los hiperparámetros

La tabla anterior fue construida tras un proceso extensivo de entrenamiento, cuyos resultados nos permitieron configurar los hiperparámetros que son elementos de esta. No obstante, vamos a hacer hincapié en el tamaño de los filtros de la segunda capa (*F2*). Anteriormente, comentamos cómo el tamaño de estas capas determina, en base al tamaño de entrada, el tamaño de salida, pero, en la tabla anterior, podemos observar que ninguno de estos parámetros ha cambiado. Esto se debe a que, en el diseño de las capas, hemos incluido una serie de ajustes relacionados que nos permiten ajustar el tamaño de las imágenes en cada capa; de esta forma, podemos aumentar el tamaño de filtros de la segunda capa, sin necesidad de tener que generar un nuevo *Dataset*.

En primer lugar, dado que la red fue entrenada en varias iteraciones, vamos a mostrar el resultado de la gráfica para las diez primeras épocas del proceso de entrenamiento.

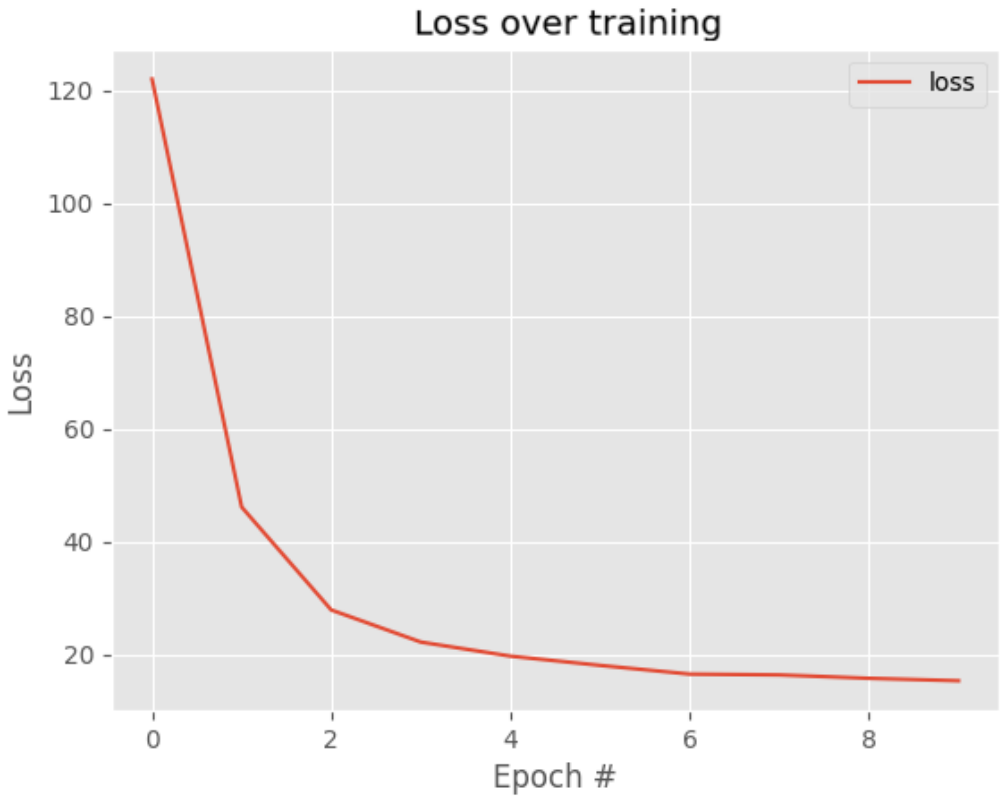

Figura 45: Gráfica de la evolución de la pérdida para el modelo final

Los resultados visibles en la gráfica de la Figura 45 nos ofrecen los mejores resultados obtenidos hasta el momento, pese a ser solo las diez primeras épocas. Para el final de nuestra época número veinte, hemos conseguido situarnos en valores que rondan un error de diez.

Sin embargo, cómo es costumbre, no vamos a conformarnos con la gráfica anterior para terminar el análisis de la red obtenida. En su lugar, nuestro siguiente paso será calcular la *PSNR* que nos devuelve nuestra red, si la aplicamos sobre las imágenes de prueba.

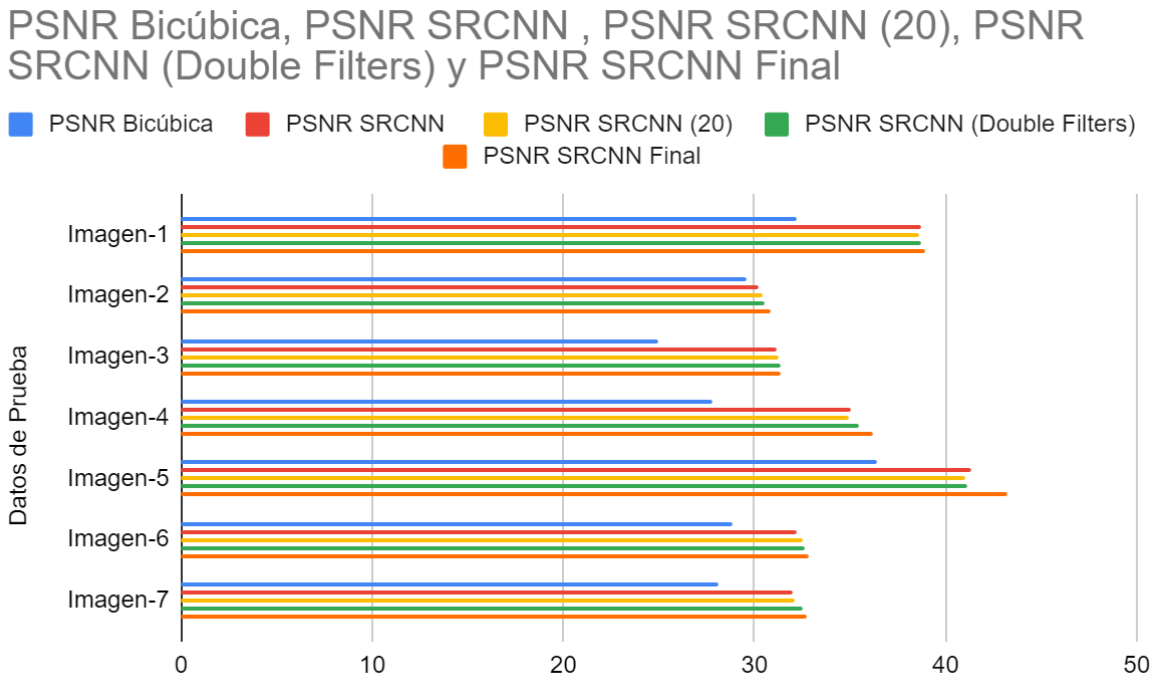

Figura 46: Comparativa entre el *PSNR* para imágenes bicúbicas y las del modelo final

Tal y como podemos comprobar, los resultados indican una nueva mejora en el *PSNR*. Tal y como paso, en el punto anterior, no solo hemos obtenido una mejora en términos globales (contamos con una nueva media de 35,16 dB); además, en cada una de las siete imágenes de prueba, hemos conseguido que el resultado obtenido sea superior.

Por último, mostraremos una versión de la imagen visible en la Figura 35, pero conseguida mediante el modelo descrito en este fragmento del documento de nuestro proyecto.

Figura 47: Imagen obtenida mediante el modelo final

5.3. Otros Conceptos Sobre el Entrenamiento

Durante todo el apartado anterior, vimos un proceso de entrenamiento dónde algunos de los hiperparámetros permanecían inmutables. Mientras que algunos de esos hiperparámetros no fueron alterados porque su valor inicial ya era correcto (o, al menos, así lo consideramos), como sería el caso de la función de pérdida o el optimizador; existen otros parámetros que definen parte del comportamiento del modelo y que, aunque tengan su influencia sobre el entrenamiento, definen más su funcionamiento final.

De esta forma, podemos identificar los parámetros *InputSize*, *Stride* y Escala, cómo los parámetros que queremos comentar en este apartado, debido a su peso en la definición del comportamiento del modelo.

En primer lugar, *InputSize* define el tamaño de entrada de la red neuronal. Con respecto al entrenamiento, de los tres mencionados, este es el que mayor importancia en este aspecto. Este hiperparámetro, al definir el tamaño de entrada de la red, acaba repercutiendo el tamaño de los resultados de cada capa y, por ende, el tamaño del *OutputSize*. Además, este parámetro tiene una gran relevancia durante la reconstrucción de la imagen, cuándo nos servimos de una imagen escalada por medios tradicionales para alimentar nuestra red. En este proceso, el *InputSize* determinará los bordes de la información de la imagen que se pierde. Sin tener en cuenta el factor de conversión *Input-Output* (que podemos calcular como

la mitad de la diferencia entre el *InputSize* y el *OutputSize*), perderemos un total de información equivalente a la siguiente función: $InfoLost = (Height\%InputSize) * Width + (Width\%InputSize) * Height$. Los parámetros de la siguiente función representan:

- *Height*: Altura de la imagen.
- *InputSize*: Tamaño de entrada.
- *Width*: Ancho de la imagen.

Con respecto al parámetro *Stride*, simplemente debemos destacar que su importancia está relegada al proceso de obtención del *Dataset*. Este parámetro es responsable de las imágenes que obtengamos, por lo que, una vez le dimos un valor inicial, decidimos dejarlo inmutable. No obstante, podríamos aumentarlo para reducir el número de imágenes, así como realizar el proceso contrario (reducirlo) para aumentar el número de imágenes.

Por último, el parámetro Escala define uno de los puntos más importantes red, la capacidad de nuestra red para incrementar el tamaño de una imagen. En este caso, hemos decidido definirlo como dos porque nos permite obtener mejores resultados de forma más temprana, lo que, para este tipo de proyecto, pensamos que es suficiente. Sin embargo, si lo deseáramos, podríamos preparar una red con la capacidad de trabajar con mayores escalas

6. Resultados y Métodos más Avanzados

Durante este capítulo, discutiremos los resultados alcanzados en el capítulo anterior. Además, comentaremos la evolución de *SRCNN* en un nuevo tipo de red, *FSRCNN* [30] (*Fast Super Resolution Convolutional Network*, lo que en castellano significa red neuronal convolucional de super resolución rápida).

6.1. Resultados Obtenidos

Ya hemos visto lo que nuestro modelo (el último presentado en el capítulo anterior) es capaz de hacer. Sin embargo, solo hemos mostrado los resultados para una imagen y no hemos comentado otros aspectos relativos a los resultados (más allá del *PSNR*).

De esta forma, comenzaremos esta sección mostrando los resultados de aplicar nuestro modelo sobre una nueva imagen (apreciable a continuación).

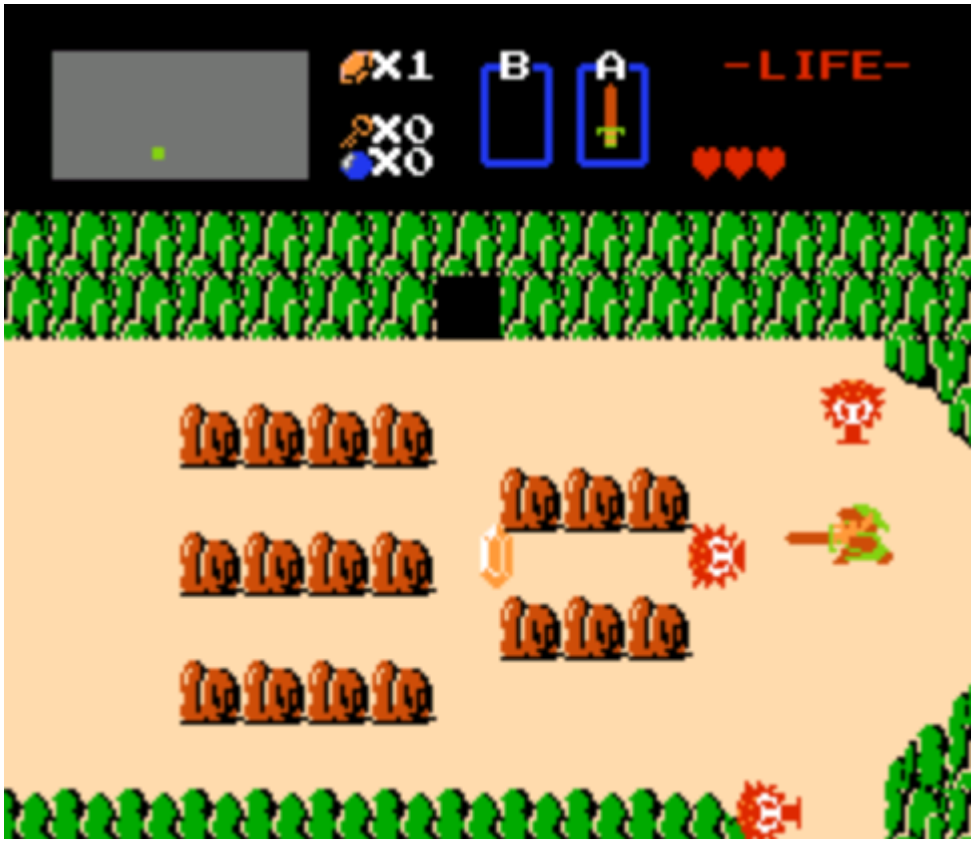

Figura 48: Imagen obtenida mediante el modelo final

Partiendo de la imagen anterior, hemos aplicado nuestro modelo final, obteniendo la ilustración representada en la Figura 49. Sin embargo, debemos señalar que, dado que estamos trabajando con videojuegos antiguos, las imágenes originales no cuentan con una gran calidad, por lo que, en muchas ocasiones, nuestros resultados son mejores, sin necesidad de tener que aumentar la resolución.

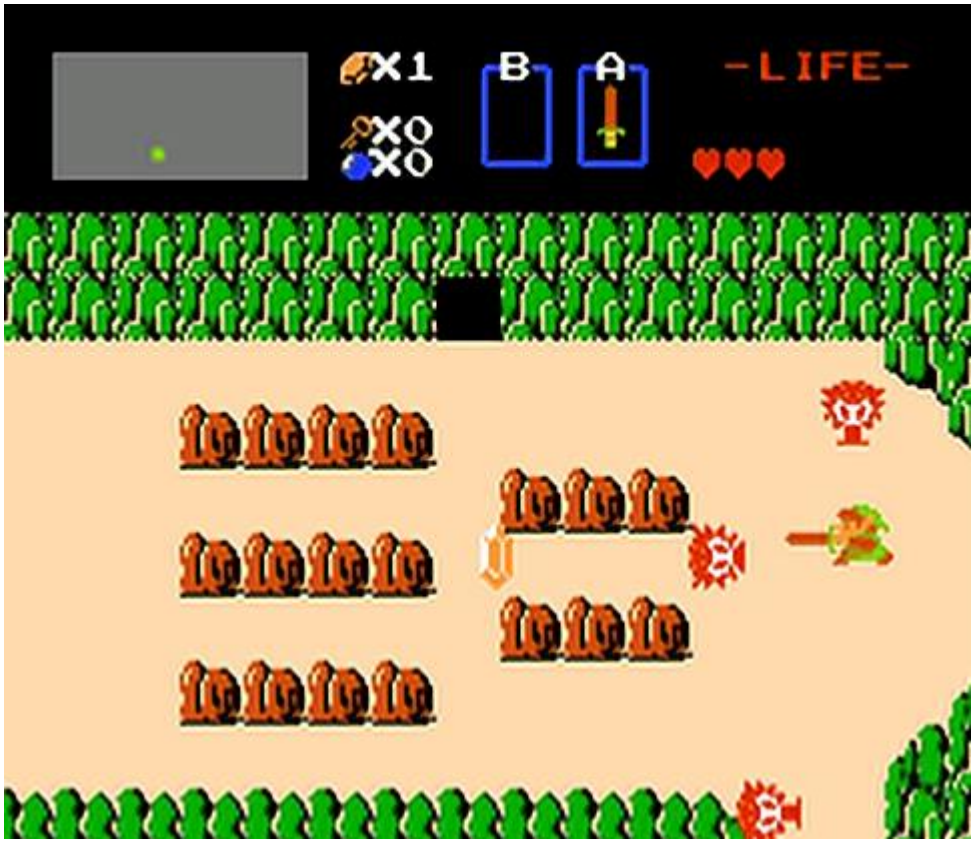

Figura 49: Imagen obtenida mediante el modelo final

No obstante, para evitar valernos únicamente de nuestra palabra y nuestro criterio, hemos realizado un pequeño experimento. En este experimento (realizado sobre diez personas), les pasábamos cinco pares de imágenes (la original y otra con nuestro modelo aplicado sobre ella). Esto nos supone un total de cincuenta casos distintos, mas, debemos añadir, que cada enfrenamiento se resolvió de forma favorable para nuestras imágenes.

Además, hemos calculado la capacidad de nuestro modelo para procesar información en tiempo real. Para esto, hemos hecho que procese una imagen (de 256x240 píxeles, tamaño similar al que podían tener juegos antiguos) y hemos calculado cuanto tiempo ha tardado en procesarla en total. De esta forma, hemos obtenido resultados muy próximos a una unidad de segundo.

6.2. Fast Super Resolution Convolutional Neuronal Network

Durante este apartado, comentaremos una arquitectura de *Deep Learning* muy relacionado con este proyecto: *FSRCNN* [30], evolución directa de la que hemos implementado (*SRCNN* [12]). En primer lugar, mostraremos una imagen (facilitada por el equipo que ha desarrollado ambas arquitecturas), dónde se comparan ambas arquitecturas.

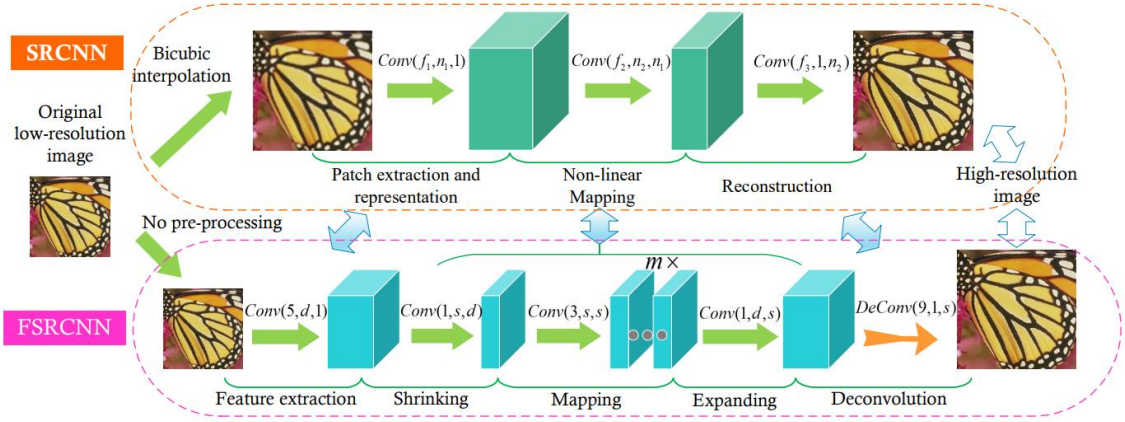

Figura 50: Comparación entre SRCNN y FSRCNN

En esta nueva arquitectura, podemos encontrar parecidos con la arquitectura *SRCNN*. Sin embargo, para que quede todo más claro, comentaremos las fases en las que se encuentra dividida *FSRCNN*:

- Extracción de Características: Esta primera fase es similar a la primera fase del modelo *SRCNN*. El objetivo de esta etapa es extraer una serie de filtros que puedan proporcionarnos información sobre la imagen (cómo aprender filtros relacionados con la obtención de bordes).
- Encogimiento: Para mejorar el coste computacional, una vez aprendidos los primeros filtros sobre la imagen de entrada, se reduce el tamaño, lo que la vuelva más rápida.
- Mapeo: El objetivo de esta fase es transformar información representada en un vector unidimensional a información bidimensional. Además, contaremos con tantas capas cómo necesitemos; sin embargo, aunque un mayor número de capas suele mejorar los resultados, también reduce su velocidad.
- Expansión: Es una fase opuesta a la segunda fase. En esta ocasión, queremos revertir el proceso, pues ya hemos conseguido mejorar el tiempo en la fase anterior, pero no queremos sacrificar calidad.

- Deconvolución: Será el proceso por el cual obtengamos la imagen final. En este caso, se usa la operación contraria a la convolución, la deconvolución.

No obstante, aún no hemos comentado uno de los aspectos más interesantes sobre este modelo. Para reescalar una imagen, no usamos una imagen obtenida mediante algún método tradicional (como la ya tan comentada interpolación bicúbica). En su lugar, podemos trabajar directamente sobre la imagen original.

Con respecto a esta arquitectura, hemos decidido implementarla, desarrollando el código necesario para cumplir todas sus necesidades (entrenamiento y pruebas, entre otros). Sin embargo, solo hemos realizado una fase de entrenamiento.

A continuación, mostraremos datos relativos al entrenamiento realizado para esta nueva arquitectura; no obstante, antes de continuar, debemos aclarar que esta no es la arquitectura principal de nuestro proyecto, por lo que no vamos a ser tan exhaustivos. En su lugar, nos centraremos en los resultados obtenidos; por tanto, primero, mostraremos la gráfica obtenida.

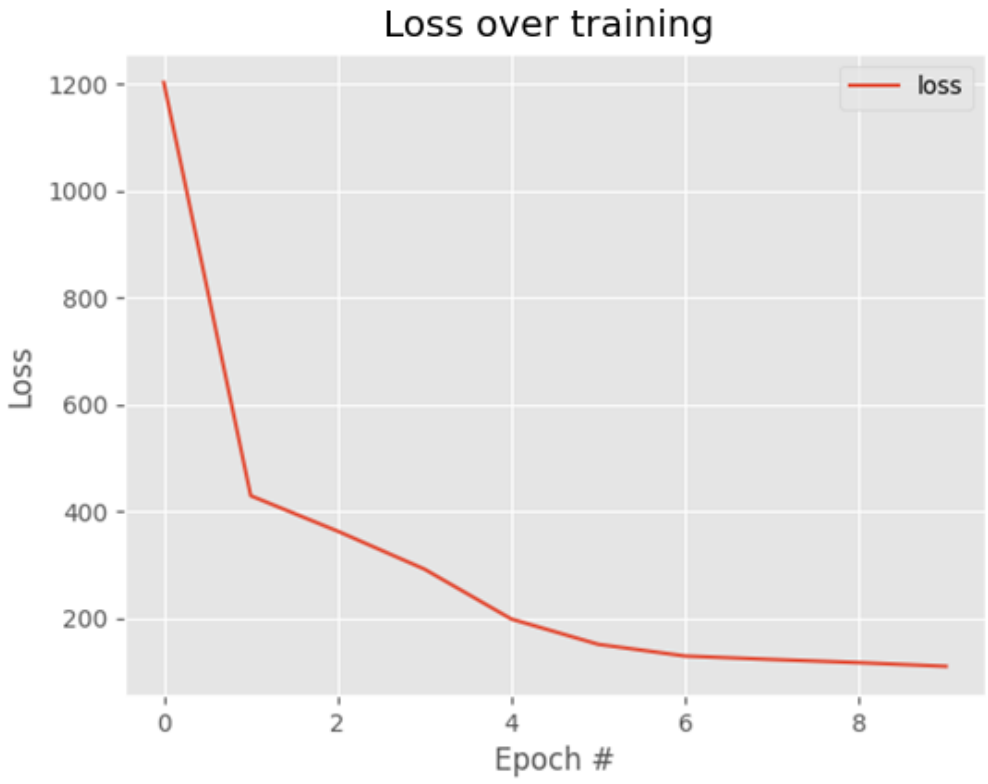

Figura 51: Gráfica de la evolución de la pérdida para FSRCNN

Tras esto, continuaremos con este proceso de análisis de resultados, por lo que mostraremos el resultado de nuestra red sobre la misma imagen que usamos anteriormente (la misma que se aprecia en la Figura 35).

Figura 52: Imagen obtenida mediante FSRCNN

Nuestro último comentario sobre esta red será que, tal y como hicimos en el apartado anterior, hemos calculado su capacidad para trabajar con imágenes en tiempo real. En esta ocasión, los resultados obtenidos han sido mejores, los cuales nos muestran que podríamos procesar unas doce imágenes por segundo.

7. Conclusiones y líneas de futuro

En este último capítulo, comentaremos aquellos aspectos finales que podemos deducir del proyecto realizado en el resto del documento. Además, comentaremos aquellos pasos que nos permitirán mejorar nuestro proyecto en un futuro.

7.1. Conclusiones

Durante este proyecto, hemos sido capaces de desarrollar un modelo de *Deep Learning* que aumenta la resolución de imágenes de forma más precisa que métodos tradicionales. Sin embargo, aún no hemos comentado los usos reales que podemos darle a nuestra red neuronal. Principalmente, pensamos que los puntos más importantes a considerar sobre las aplicaciones del modelo *SRCNN* [12] son los siguientes:

- Reconstrucción de videojuegos antiguos. Este es el objetivo sobre el que se construyó nuestro proyecto, ayudar a abaratar los costes de producción de versiones mejoradas de juegos antiguos. Los resultados obtenidos no nos reportarían grandes saltos gráficos, pero podrían ser suficientes para que los gráficos fueran mucho más nítidos y atractivos para los jugadores.
- Configuración de Resolución. Pensamos que este proyecto también puede servir para el desarrollo de juegos modernos, principalmente estudios con presupuestos más reducidos. Sobre el arte del juego, se podría usar este algoritmo para adaptarlo a distintas resoluciones. Además, creemos que, si se entrenase la red sobre imágenes pertenecientes al juego que compartieran una coherencia visual ente ellas, se podría mejorar los resultados (específicos para un juego concreto).
- Ejecución en Tiempo Real. Este es uno de los puntos más importantes a destacar, pues desarrollar una aplicación que ejecutase juegos con baja resolución y, en tiempo real, nos devolviera gráficos mejorados sería un uso muy interesante para este proyecto. Infortunadamente, el modelo *SRCNN* no sería capaz de llevar a cabo esta tarea, pero su evolución, que también hemos comentado en este proyecto, *FSRCNN* [30], sí podría ser una herramienta interesante para este caso.

Con esto, habríamos visto los principales puntos de aplicación de nuestro proyecto. Sin embargo, debemos también considerar aquellos otros modelos más avanzados y los resultados que ofrecían. No obstante, el modelo con el que hemos trabajado nos permite

disfrutar de una simpleza mayor, por lo que puede ser más sencillo de adoptar para proyectos más modestos.

Para finalizar este apartado de conclusiones, comentaremos que podríamos haber preparado nuestra red para trabajar sobre escalas más grandes, que, en los casos comentados, seguramente sean las situaciones principales.

7.2. Líneas de futuro

Para terminar este documento, indicaremos una serie de pautas para el futuro, que nos permitan seguir desarrollando este proyecto.

En primer lugar, continuar entrenando y estudiando nuestro modelo de *SRCNN* nos permitirá mejorar los resultados obtenidos. Por otro lado, podríamos comenzar con el desarrollo de programas que implementen nuestra red para darle un uso más profesional.

Además, nos ha quedado pendiente intentar resolver el problema relativo a la perdida de información. No obstante, algunas posibles situaciones serían adaptar el algoritmo de reconstrucción para que se aplicase sobre los bordes (aunque sobrescribiera información de las filas y columnas adyacentes), expandir la imagen para que pudiera ajustarse de forma perfecta al tamaño de entrada de la red o sustituir los valores de los bordes por los resultados obtenidos mediante otro algoritmo de reescalado.

Por último, haremos un comentario desde la perspectiva académica. Cómo estudiantes, este Trabajo Fin de Máster nos ha permitido obtener un mayor conocimiento del área de la Ingeniería Informática del *Deep Learning*; sin embargo, no solo nos ha permitido obtener un conocimiento básico y simple; en su lugar, hemos podido enfocar el conocimiento adquirido para que se ajuste a un problema que queríamos tratar y que, además, hemos podido relacionar con la temática de este Máster, los videojuegos.

De esta forma, espero poder seguir incrementado mi conocimiento en este campo, con el objetivo de poder desarrollar más proyectos de inteligencia artificial que estén enfocados en los videojuegos.

Bibliografía

[1] Square Enix. (1997). Final Fantasy VII. Obtenido de https://square-enix-games.com/
[2] Square Enix. (2020). Final Fantasy VII Remake. Obtenido de https://square-enix-games.com/
[3] Nintendo. (2011). The Legend of Zelda: Skyward Sword. Obtenido de https://www.nintendo.es/
[4] Nintendo. (2021). The Legend of Zelda: Skyward Sword HD. Obtenido de https://www.nintendo.es/
[5] Entertainment software association. (2021). Entertainment software association. Obtenido de https://www.theesa.com/resource/2021-essential-facts-about-the-video-game-industry/
[6] Nintendo. (2017). The Legend of Zelda: Breath of the Wild. Obtenido de https://www.nintendo.es/
[7] Intelligent Systems. (1990). Fire Emblem: Ankoku Ryū to Hikari no Tsurugi. Obtenido de https://www.intsys.co.jp/english/
[8] Entertainment software association. (2019). Entertainment software association. Obtenido de https://www.theesa.com/resource/essential-facts-about-the-computer-and-video-game-industry-2019/
[9] Sherrel, L. (2013). Waterfall Model, Dordrecht: Springer.
[10] Chollet, F. (2018). Deep Learning with Python. Ed. Manning.
[11] Courville, A., Goodfellow, I., & Bengio, Y. (2016). Deep Learning. Cambridge, Massachusetts: The MIT Press.
[12] Dong, C., Loy, C. C., Tang, X., & He, K. (2015). Image Super-Resolution Using Deep Convolutional Networks. arXiv.org.
[13] Python Software Foundation. (2022). Python. Obtenido de https://www.python.org/
[14] Chollet, F. (2022). Keras. Obtenido de https://keras.io/
[15] Google LLC. (2022). TensorFlow. Obtenido de https://www.tensorflow.org/?hl=es-419
[16] Intel Corporation. (2022). Open CV. Obtenido de https://opencv.org/
[17] Oliphant, T. (2022). NumPy. Obtenido de https://numpy.org/
[18] Hunter, J. (2022). Matplotlib. Obtenido de https://matplotlib.org/
[19] JetBrains (2022). PyCharm. Obtenido de https://www.jetbrains.com/es-es/pycharm/
[20] Gitlab Inc. (2022). GitLab. Obtenido de https://about.gitlab.com/
[21] Atlassian (2022). SourceTree. Obtenido de https://www.sourcetreeapp.com/

[22] Rosenblatt, F. (1958). The Perceptron: A Probabilistic Model for Information Storage and Organization. Psychological Review.
[23] DeFelipe, J. (1998). Cajal. MIT Press
[24] Minsky, M., & Papert, S. (1969). Perceptrons: an introduction to computational geometry. MIT Press.
[25] LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep Learning. Nature.
[26] LeCun, Y. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE.
[27] LeCun, Y. & Bengio, Y. (1995). Convolutional networks for images, speech, and time series. The handbook of brain theory and neural networks.
[28] Goodfellow, I., & et al. (2014) Generative Adversarial Nets.papers.nips.cc.
[29] Gatys, L. A., Ecker, A., & Bethge, M. (2015). A Neural Algorithm of Artistic Style. arXiv.org.
[30] Dong, C., Loy, C. C., & Tang, X. (2015). Accelerating the Super-Resolution Convolutional Neural Network. arXiv.org.
[31] Universidad de Montreal. (2022). Theano. Obtenido de https://theano-pymc.readthedocs.io/en/latest/
[32] Leding, C., & et al. (2017). Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network. arXiv.org.
[33] He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep Residual Learning for Image Recognition. arXiv.org.
[34] Google LLC. (2022). Google Imagenes. Obtenido de: https://www.google.es/imghp?hl=es.
[35] The HDF5 Group. (2022). The HDF5 Library and File Format. Obtenido de: https://www.hdfgroup.org/solutions/hdf5/.
[36] Ba, J., & Kingma, D. (2014). Adam: A Method for Stochastic Optimization. arXiv.
[37] H5PY Organization. (2022). H5PY. Obtenido de: https://www.h5py.org/.