Escuela Politécnica Superior de Jaén

Grado en Ingeniería Informática

UNIVERSIDAD DE JAÉN
Escuela Politécnica Superior de Jaén

Trabajo Fin de Grado

Procesamiento de imágenes de foto-trampeo con técnicas de aprendizaje profundo para el descarte automático de imágenes vacías

Alumno: David de la Rosa de la Rosa

Tutor: Francisco Charte Ojeda

María José del Jesus Díaz

Dpto: Informática

UNIVERSIDAD DE JAÉN

D./Dª Francisco Charte Ojeda y D./Dª María José del Jesus Díaz, tutor(es) del Trabajo Fin de Grado titulado: **Procesamiento de imágenes de foto-trampeo con técnicas de aprendizaje profundo para el descarte automático de imágenes vacías**, que presenta David de la Rosa de la Rosa, autoriza(n) su presentación para defensa y evaluación en la Escuela Politécnica Superior de Jaén.

Jaén, julio de 2022

El estudiante

DE LA ROSA DE LA ROSA DAVID - 77647640P
Firmado digitalmente por DE LA ROSA DE LA ROSA DAVID - 77647640P
Fecha: 2022.07.01 11:56:04 +02'00'

David de la Rosa de la Rosa

Los tutores

CHARTE OJEDA FRANCISCO - 25996290B
Firmado digitalmente por CHARTE OJEDA FRANCISCO - 25996290B
Fecha: 2022.07.01 12:22:46 +02'00'

Francisco Charte Ojeda

DEL JESUS DIAZ MARIA JOSE - 26485651R
Firmado digitalmente por DEL JESUS DIAZ MARIA JOSE - 26485651R
Fecha: 2022.07.01 12:48:01 +02'00'

María José del Jesus Díaz

Agradecimientos

Escribir estas líneas significa dejar atrás una de las etapas más importantes de mi vida. Me gustaría reservar este pequeño espacio en la memoria para agradecer a todas las personas que han contribuido a que haya logrado acabar con éxito estos cuatro años en el grado.

En primer lugar, agradecérselo todo a mi familia. Ellos siempre han estado ahí en los momentos más difíciles y han sabido animarme a no tirar la toalla y seguir adelante. Son la base de este trabajo.

Por supuesto, dar las gracias a todos los compañeros y amigos que han hecho que estos años no solo sean los más importantes, sino también los mejores años de mi vida. Nunca se olvidarán las risas tomándonos un café o los nervios justo antes de empezar un examen.

Por último, y como no podría ser de otra forma, agradecer enormemente a Francisco Charte y María José, mis tutores de TFG, todo el tiempo dedicado a guiarme y aconsejarme en la realización del trabajo.

A todos vosotros, gracias.

Tabla de contenidos

Lista de figuras

Lista de tablas

Capítulo 1

INTRODUCCIÓN

En este capítulo se planteará un acercamiento inicial al problema a resolver, explicando en primer lugar qué es el fototrampeo y los problemas que puede conllevar el tratamiento de las imágenes obtenidas en entornos naturales con dispositivos de fototrampeo. Seguidamente se detallará la estructura de la memoria de este Trabajo de Fin de Grado. Por último, se mostrará la lista de acrónimos usados a lo largo de la memoria.

1.1. Planteamiento del problema

La preservación del entorno natural es un aspecto de vital importancia para la supervivencia de una gran multitud de especies. Sin embargo, el estudio de estos ecosistemas no es una tarea trivial. Se requieren profesionales que sean capaces de reconocer señales que indiquen la presencia de animales y suficiente tiempo como para lograr comprender su comportamiento. Además, muchas de las zonas están en lugares remotos de difícil acceso, añadiendo una dificultad adicional a esta tarea. Sin embargo, a pesar de todos los problemas mencionados, la monitorización constante de la fauna presente sigue siendo fundamental.

Es habitual que esta labor se apoye en fotografías tomadas en aquellos entornos que se quieran analizar. Gracias a un análisis de estas imágenes, es posible realizar un seguimiento continuo de la fauna de una forma mucho más sencilla, permitiendo conocer su distribución y comportamiento.

Para la obtención de las imágenes en entornos naturales es muy común utilizar dispositivos de fototrampeo. Una cámara de fototrampeo [1] [2], como la que se mues-

tra en la figura 1.1, es una herramienta para la vigilancia y estudio del entorno salvaje desde un punto de instalación fijo, permitiendo capturar imágenes diurnas y nocturnas sin interferir en el comportamiento animal. Las cámaras se instalan normalmente en árboles, donde pasan inadvertidas, de forma que los animales puedan acercarse a ellas con total confianza. Estas cámaras suelen incorporar un sensor de movimiento. Cuando se activa el sensor, idealmente cuando un animal pasa cerca de la cámara, esta captura una ráfaga de fotografías que posteriormente serán almacenadas en una base de datos. La ventaja de hacer uso de estos dispositivos está clara: es mucho más sencillo fotografiar animales en su entorno con una pequeña y discreta cámara que trasladando al lugar un equipo de fotografía que probablemente hará que los animales se alejen.

Figura 1.1: Cámara de fototrampeo colocada en un árbol

Una vez se tienen todas las fotografías capturadas y almacenadas, el siguiente paso es el procesamiento y análisis de las mismas. En este punto se plantean diferentes retos, como la detección de animales en las imágenes o la identificación del tipo de animal presente. Es en este momento donde aparece el problema de las imágenes vacías. En ocasiones el sensor de movimiento se activa pero no aparece ningún animal en la imagen, ya sea porque la cámara ha captado el movimiento de una simple rama o piedra o porque el animal haya pasado muy rápido y la cámara no ha tenido tiempo suficiente para capturarlo.

Se pueden plantear diferentes formas para abordar el problema. Por un lado tenemos el descarte manual de esas fotografías. Esta técnica podría ser viable en caso de mantener una base de datos con pocas imágenes. Sin embargo, cuando el número de

fotografías aumenta hasta las decenas o cientos de miles, este trabajo comienza a ser complejo y en ocasiones inviable.

Por otro lado contamos con la posibilidad de descartar automáticamente las imágenes vacías. La idea se basa en ceder este trabajo a un ordenador capaz de diferenciar entre las imágenes útiles, aquellas que han captado la imagen de un animal, de las imágenes vacías. Los profesionales del ámbito señalan como una tarea muy relevante el descarte automático de imágenes vacías, ya que esto permitiría agilizar en gran medida el trabajo posterior, centrándose únicamente en aquellas fotografías relevantes, es decir, fotografías en las que aparecen animales.

Con esta motivación, se planteará y desarrollará un sistema de descarte automático de imágenes vacías haciendo uso de técnicas basadas en Inteligencia Artificial, concretamente *Deep Learning* ya que ha demostrado ser de gran utilidad en tareas de análisis de imágenes.

1.2. Estructura de la memoria

A continuación se presenta el contenido de los capítulos de la memoria, ofreciendo una visión general de cada uno.

Capítulo 1. Introducción

Capítulo actual. Se presenta una primera aproximación al trabajo que se desarrollará durante el proyecto, analizando tanto el problema a resolver como las técnicas que se aplicarán. También se muestra el glosario de acrónimos usados en la memoria.

Capítulo 2. Antecedentes

En este capítulo se abordará el tipo de técnica presentada para resolver el problema propuesto. Se analizará qué es el *Deep Learning*, un breve recorrido por su historia y los tipos principales de redes neuronales que se utilizan actualmente.

Capítulo 3. Objetivos

Como su propio nombre indica, en este capítulo se detallarán los objetivos del TFG, desgranando el objetivo principal en varios objetivos específicos agrupados por temáticas.

Capítulo 4. Materiales y métodos

Tras concretar el problema que se aborda en el TFG y el tipo de técnica en la que nos basaremos, pasamos a especificar los materiales y métodos que se utilizarán para resolver el problema. Se describe el conjunto de fotografías disponibles para alimentar los modelos y el preprocesamiento aplicado, las herramientas de programación y la elección y entrenamiento de los modelos concretos basados en redes neuronales.

Capítulo 5. Resultados

Una vez definidas las herramientas, pasamos a presentar los resultados obtenidos. Se analizarán y mostrarán los resultados con ayuda de herramientas gráficas que faciliten su comprensión y se comprobará la viabilidad de los métodos propuestos.

Capítulo 6. Conclusiones

Finalmente, en este capítulo se presentarán las conclusiones extraídas tras la finalización del proyecto, indicando qué y cómo se ha hecho y algunas propuestas para ampliar el proyecto en el futuro.

1.3. Glosario de términos

En esta sección se recogen los acrónimos utilizados a lo largo de la memoria del TFG. Se incluye el acrónimo y la traducción al español y al inglés.

Acrónimo	Significado (Español)	Significado (Inglés)
AE	Autocodificador	Autoencoder
AI	Inteligencia Artificial	Artificial Intelligence
BBDD	Base de datos	Database
CNN	Red neuronal convolucional	Convolutional Neural Network
DL	Aprendizaje Profundo	Deep Learning
FN	Falso negativo	False Negative
FP	Falso positivo	False Positive
GAN	Red neuronal generativa adversaria	Generative Adversarial Network
MAE	Error absoluto medio	Mean Absolute Error
ML	Aprendizaje Automático	Machine Learning
MLP	Perceptrón multicapa	Multi Layer Perceptron
MSE	Error cuadrático medio	Mean Square Error
RAE	Autoencoder robusto	Robust Autoencoder
RNN	Red neuronal recurrente	Recurrent Neural Network
SSIM	Similaridad estructural	Structural Similarity
TFG	Trabajo de Fin de Grado	Degree Final Project
TN	Verdadero negativo	True Negative
TP	Verdadero positivo	True Positive

Tabla. 1.1: Lista de acrónimos utilizados en la memoria.

Capítulo 2

FUNDAMENTOS Y ANTECEDENTES

En este capítulo se estudiarán los fundamentos de las técnicas que se aplicarán durante el desarrollo de este TFG, centradas en DL. Además, se estudiarán diversos antecedentes para conocer cómo se ha resuelto este problema y qué técnicas se han utilizado.

2.1. Contexto

En esta sección se expondrá todo lo necesario para conocer de forma general qué es el DL. Detallaremos dónde se encuentra dentro del área de la Informática para luego hacer un breve repaso por su historia, desde sus comienzos hasta la actualidad. Seguidamente, mostraremos el funcionamiento básico de una red neuronal, especificando su arquitectura más básica. Por último, estudiaremos el funcionamiento de redes neuronales avanzadas.

Antes de pasar a explicar qué es el DL, es necesario definir dos conceptos previos muy relacionados: AI y ML

2.1.1. Qué es la Inteligencia Artificial

La AI es una rama del conocimiento dentro de la Informática que se centra en el diseño de sistemas inteligentes, entendiendo estos como aquellos que toman decisiones racionales sobre su entorno para lograr un fin determinado.

Existen numerosas definiciones de AI. Una de ellas la encontramos en el informe [3], donde se define de la siguiente forma:

"*Artificial intelligence (AI) systems are software (and possibly also hardware) systems designed by humans that, given a complex goal, act in the physical or digital dimension by perceiving their environment through data acquisition, interpreting the collected structured or unstructured data, reasoning on the knowledge, or processing the information, derived from this data and deciding the best action(s) to take to achieve the given goal. AI systems can either use symbolic rules or learn a numeric model, and they can also adapt their behaviour by analysing how the environment is affected by their previous actions.*"

En las definiciones disponibles, hay que saber diferenciar entre la AI específica y la AI general (ver referencia [4]).

AI específica es un tipo de AI con funcionalidad limitada. Se refiere al uso de algoritmos avanzados capaces de resolver un problema específico o realizar ciertas tareas en un entorno predefinido sin que esto conlleve el uso de las habilidades cognitivas humanas en su conjunto. Un algoritmo de este tipo solo podrá resolver el problema para el que ha sido entrenado, sin salirse de su marco de trabajo. Un buen ejemplo de este tipo de AI podrían ser los asistentes de voz como Alexa o Siri. Usan AI pero solo funcionarán para una serie de trabajos específicos.

A pesar de esto, el hecho de ser denominada en ocasiones como "AI débil" no implica que sea inútil, ya que ha demostrado con creces ser de mucha utilidad en problemas actuales.

Por su lado, **AI general** es una forma de AI que demuestra una inteligencia humana. En la actualidad, esta acepción de AI se asocia a sistemas con la capacidad para afrontar cualquier tipo de tareas que pueda hacer un ser humano, no solo uno específico como se mencionaba en AI específica. Las máquinas con este tipo de inteligencia son capaces de actuar por sí mismas, tomando decisiones sin necesidad de que un humano las haya programado de forma directa. Un nivel superior a esta AI, denominado super-inteligencia artificial, queda definido por el filósofo Nick Bostrom como "cualquier inteligencia que supere ampliamente el rendimiento cognitivo de cualquiera de nosotros en todos los ámbitos de interés" [5]. En la actualidad, tanto la AI general como la super-inteligencia se pueden ver únicamente en la ciencia-ficción. Toda la AI desarrollada hasta el momento entra dentro del grupo AI específica.

Encontramos una gran variedad de campos dentro de la AI, como visión por ordenador o procesamiento del lenguaje natural, entre otros [3]. Para el desarrollo del TFG, nos centraremos en uno de los principales campos de la AI, el ML.

2.1.2. Qué es el *Machine Learning*

El ML es un campo de la AI que da a los ordenadores o programas la capacidad de aprender sin ser explícitamente programados para ello. Tom Mitchell [6] define el concepto:

"*A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E*"

En general, el ML permite que un programa informático realice tareas sin necesidad de que se especifiquen las reglas a seguir para cumplir su objetivo, completándolas de forma autónoma en base a prueba y error, tomando como entrada la información que el programador le proporcione y teniendo a su disposición una medida de rendimiento.

Dentro del ML podemos encontrar algoritmos que simulan el proceso de aprendizaje de distintas formas (ver referencia [7]). Por lo general se pueden agrupar en tres categorías bien diferenciadas:

- **Aprendizaje supervisado**

  La principal característica de esta categoría es que los datos que usamos para el entrenamiento están etiquetados con la solución correcta, permitiendo comparar la solución aportada por el algoritmo con el valor de salida esperado.

  Este tipo de aprendizaje se puede aplicar a varios tipos de problemas.

  Por un lado encontramos problemas de clasificación, donde el objetivo será establecer una correspondencia entre los datos de entrada y una salida discreta en forma de grupos predefinidos o clases. Durante el entrenamiento, a partir de ejemplos con ciertos atributos se extraerán patrones que permitan predecir la pertenencia de un objeto nuevo a una de las posibles clases.

  Por otro lado esta misma metodología se puede aplicar a problemas de regresión, en los que se pretende establecer una correspondencia entre los datos de entrada y una salida con valores continuos. Podemos incluir también en este grupo problemas de predicción de series temporales, es decir, predecir el comportamiento de una variable continua a lo largo del tiempo.

- **Aprendizaje no supervisado**

  En el aprendizaje no supervisado, a diferencia del anterior los datos de entrenamiento no incluyen etiquetas, por lo que será el propio algoritmo quien intente clasificar la información.

  Esta metodología de aprendizaje es ideal para resolución de problemas de *clustering*, en los cuales se busca agrupar instancias en base a sus características, creando grupos o *clusters* de instancias similares.

  Además, podemos aplicar esta técnica en tareas de descubrimiento de asociaciones, donde se busca encontrar relaciones entre variables de un conjunto de datos. Suele ser utilizado en el análisis de mercado, buscando relaciones entre los productos (los usuarios que compran el producto X suelen comprar también el producto Y).

- **Aprendizaje por refuerzo**

  En este grupo los datos no cuentan con una etiqueta, pero tampoco se pueden considerar aprendizaje no supervisado puro. En este caso se recompensan las decisiones correctas del algoritmo (reforzando ese comportamiento para el futuro) y se penalizan las malas decisiones.

  Este tipo de aprendizaje es ampliamente utilizado en el desarrollo de AI en videojuegos. En un videojuego encontramos un entorno, el propio mundo donde se desarrolla, y un agente sobre el cual queremos aplicar AI. En lugar de entrenar al modelo con miles y miles de ejemplos de buenas y malas acciones del agente, como se haría si se aplica aprendizaje supervisado, le damos libertad para realizar acciones en un principio aleatorias. Como resultado de esas acciones, el agente será penalizado o recompensado y por tanto propiciando que su comportamiento se base en acciones correctas.

Por otro lado, para que todo este proceso sea posible, debemos “alimentar” al modelo con datos. Un correcto entrenamiento del mismo requiere realizar una partición previa del conjunto original de datos, dividiéndolo en tres partes:

- **Datos de entrenamiento**.

  Este será el conjunto de datos que emplearemos durante el entrenamiento del modelo para que ajuste sus parámetros, optimizando los resultados.

- **Datos de validación**.

  Es un subconjunto de los datos de entrenamiento. Se usará para ajustar y optimizar los parámetros y/o hiperparámetros usados durante el entrenamiento del modelo. Estos parámetros serán dependientes del tipo de modelo que utilicemos.

- **Datos de test**.

  En ocasiones el modelo funciona muy bien con los datos de entrenamiento pero es incapaz de generalizar, es decir, si le proporcionamos una entrada distinta a cualquiera del entrenamiento es muy probable que nos de un resultado erróneo. Esto puede ser provocado por el denominado *overfitting*, el sobre-entrenamiento del modelo, donde este memoriza los datos de entrenamiento, haciendo que sea incapaz de generalizar el conocimiento a entradas diferentes. Otra posible causa es el *underfitting*, es decir, proporcionar al modelo un conjunto de entrenamiento demasiado pequeño, impidiendo un correcto aprendizaje de los patrones.

  El objetivo de la partición de test es en definitiva una estimación honesta de la capacidad de respuesta del modelo desarrollado ante nuevos datos.

El ML se basa en diferentes algoritmos para resolver los problemas que se proponen. Es importante recalcar que no existe un algoritmo perfecto capaz de resolver todos los problemas, sino que los programadores deberán escoger y desarrollar uno de ellos en base al contexto del problema y la naturaleza de los datos. En la referencia [8], se describen los principales algoritmos y modelos que encontramos dentro del ML, algunos de los cuales se mencionan a continuación:

- **Árboles de decisión**

  Un árbol de decisión es un modelo que representa elecciones y resultados en forma de árbol. Cada nodo interno representa una comprobación o *test* sobre un atributo, mientras que cada rama hace referencia a uno de los posibles resultados del *test*. La decisión de que la entrada pertenezca a una clase u otra se tomará en los nodo hoja.

- **Máquina de vectores de soporte**

  El objetivo del algoritmo máquinas de vectores de soporte es encontrar un plano en un espacio n-dimensional que sea capaz de separar los datos en sus clases correspondientes. En caso de que exista más de un plano capaz de conseguir esto, se buscará el plano óptimo, aquel que maximice la distancia hasta el borde o margen de cada clase, minimizando por tanto el error de clasificación.

- **Redes neuronales**

  Las redes neuronales son algoritmos que se esfuerzan por reconocer las relaciones en un conjunto de datos a través de un proceso que imita la forma en que funciona el cerebro humano. Este concepto se enmarca dentro de uno de los campos del ML, el DL, del que hablaremos en el siguiente apartado.

2.1.3. Qué es el *Deep Learning*

Una vez explicado qué es AI y ML, podemos definir el concepto de DL [9]. Este se considera una rama del ML que se basa en el uso de redes neuronales artificiales. En español, el término se traduce como aprendizaje profundo, haciendo referencia a las múltiples capas de procesamiento que forman las redes neuronales. Cada una de estas capas estará especializada en la identificación de características específicas, tomando como entrada las salidas de la capa previa y aprendiendo desde patrones simples en las primeras capas hasta patrones muy complejos en capas superiores basados en la información de las capas inferiores.

De esta forma, la relación entre AI, ML y DL quedaría tal y como se aprecia en la figura 2.1

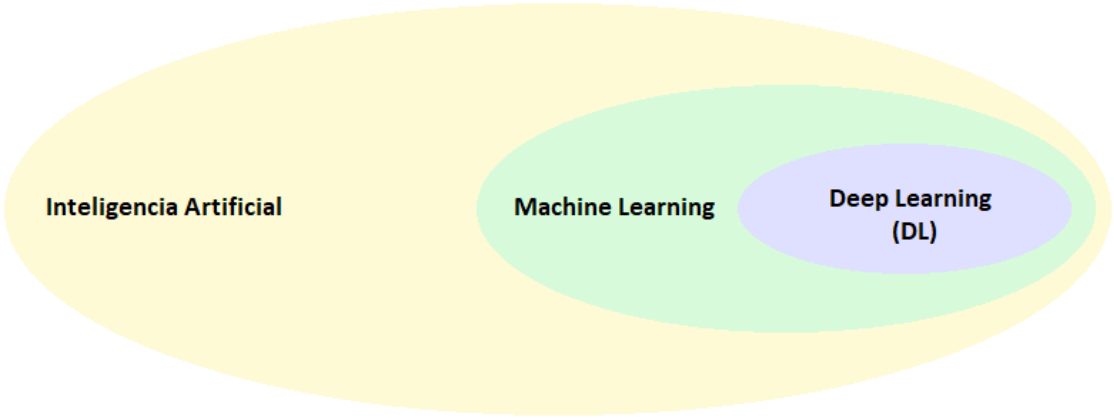


Figura 2.1: Esquema que muestra la relación entre AI, ML y DL
[10]

El modelo más común de DL hace uso del aprendizaje supervisado para realizar tareas de clasificación o detección de objetos entre muchas otras. Sin embargo también se pueden plantear variantes de aprendizaje no supervisado, realizando por ejemplo tareas de *clustering*.

Su funcionamiento específico y arquitecturas más conocidas se estudiarán en la secciones 2.1.7 y 2.2.

2.1.4. Evolución de las redes neuronales desde sus orígenes hasta hoy

Una vez explicado qué es el DL y dónde se encuadra dentro del marco del conocimiento, es interesante conocer cuáles son sus orígenes y cuál ha sido el camino recorrido que ha hecho que el DL sea una técnica muy conocida en la actualidad. En esta sección haremos un breve repaso por la historia del DL [1] (ver figura 2.2).

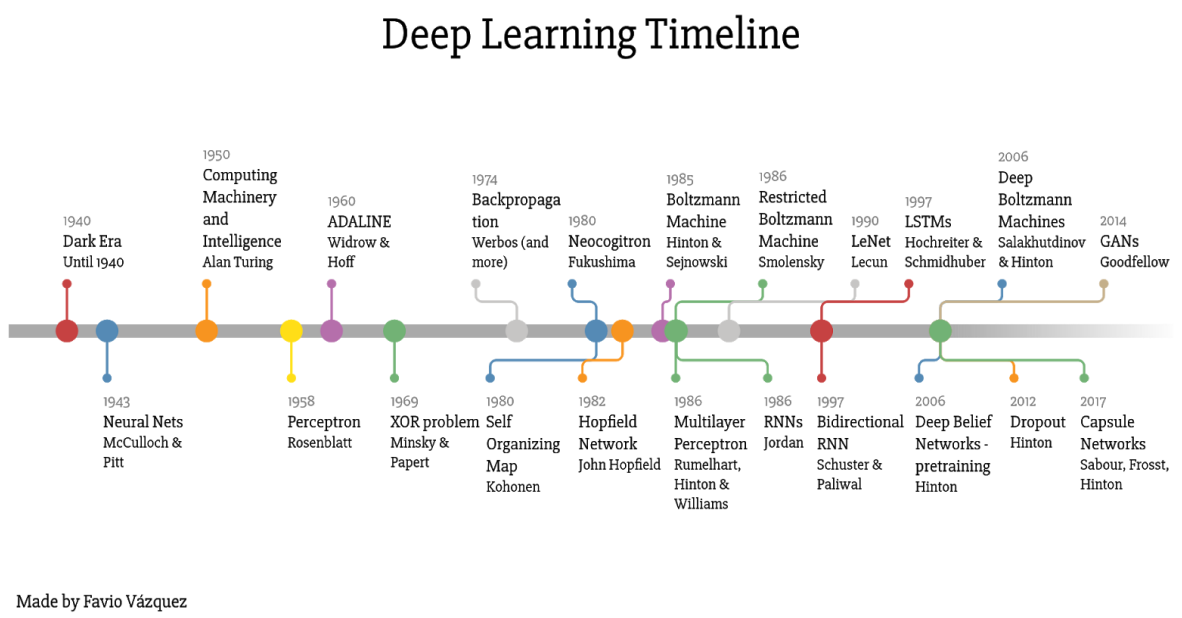


Figura 2.2: Línea del tiempo del Deep Learning desde 1940 hasta 2017 [11]

McCulloch y Walter Pitts (1943)

Un buen punto de partida son los estudios sobre el aprendizaje biológico que llevaron a cabo McCulloch y Walter Pitts en el año 1943, *A logical calculus of the ideas immanent in nervous activity* [12]. En ellos, desarrollaron un modelo matemático, comúnmente denominado *McCulloch-Pitts neuron* que toma como idea fundamental las redes neuronales presentes en el cerebro humano. Así, proporcionaron una forma de describir las funciones cerebrales y demostraron que elementos aparentemente simples conectados entre sí en una red neuronal pueden tener un gran poder computacional. Sin embargo, el artículo no recibió toda la atención que merecía en su momento.

Donald Hebb (1949)

[1] Es importante aclarar que algunos de los modelos básicos que se detallan en esta sección, como el perceptrón o la llamada *McCulloch-Pitts neuron*, no eran denominados DL. Es cierto que son considerados elementos que propiciaron el origen del DL, aunque este término fuera acuñado en una época más moderna.

Donald Hebb introdujo lo que hoy conocemos como “el peso” de una neurona. En el libro *The Organisation of Behaviour: a neuropsychological theory* [13] se propuso la llamada Regla de Hebb. Esta regla o teoría se puede resumir como “las células que se activan unidas, permanecen unidas”. Cuando una neurona tiende a activar a otra, el peso o la importancia de esa conexión aumenta. Esta idea propiciaría el algoritmo básico de aprendizaje mediante redes neuronales artificiales.

Frank Rosenblatt (1958)

El doctor Frank Rosenblatt es considerado uno de los investigadores más relevantes de este área, conocido sobre todo por el desarrollo del algoritmo perceptrón [14], un clasificador binario. Aunque lo describiremos de forma mucho más detallada en la sección 2.1.5, en general se puede definir como una función capaz de decidir si un vector de entrada pertenece o no a una clase específica.

En un principio la idea del perceptrón resultó sumamente atractiva a nivel académico, pero pronto se probó que no podría ser entrenado para conocer otros tipos de patrones, lo que provocó un estancamiento en el desarrollo de nuevas técnicas.

David Rumelhart (1986)

David Everett Rumelhart fue un psicólogo estadounidense que centró su trabajo en el análisis matemático cognitivo y la inteligencia artificial simbólica. Entre sus trabajos destaca el artículo *Backpropagation: The Basic Theory* ([15]). En él se introduce uno de los algoritmos más utilizados en la actualidad en el entrenamiento de redes neuronales, permitiendo calcular los pesos de las neuronas. Su funcionamiento básico, aunque se detallará más adelante, consiste en tomar el error de las salidas de la red neuronal para propagarlo hacia atrás, detectando cuánto error ha aportado cada neurona al error total y ajustando sus pesos para minimizarlo.

Actualidad

En la actualidad, el desarrollo de ordenadores capaces de procesar mucha más información, el uso de GPU (herramienta óptima para el desarrollo en DL) y el aumento exponencial de bases de datos listas para utilizar como entrada a redes neuronales han hecho que el DL sea una herramienta cada vez más usual a la hora de resolver todo tipo de problemas, tal y como demuestra el libro [16]. Algunas áreas donde destaca son:

- Visión artificial, con el uso de redes neuronales convolucionales capaces de clasificar imágenes con una precisión muy alta. Destacan aplicaciones del DL en

campos de medicina, por ejemplo en el reconocimiento y clasificación de tumores en radiografías [17], o en el área del reconocimiento facial [18].

- Procesamiento del lenguaje natural. Usando técnicas DL, el ordenador es capaz de identificar y procesar el lenguaje humano [19].
- Gaming. Hace escasos 5 años veíamos cómo el programa de Google, *AlphaGo* [20], fue capaz de derrotar al campeón mundial de Go, el popular juego de mesa.

Como se ha expuesto, el DL es una tecnología que, a pesar de llevar poco tiempo con nosotros, ha demostrado ser de gran utilidad para dar solución a problemas que resultaban inviables de resolver con la aplicación de otras técnicas.

2.1.5. Arquitectura básica de un modelo neuronal

Una vez conocemos a nivel general qué es el DL y para qué sirve, podemos centrarnos en su interior; cómo se forma una arquitectura basada en DL. Para ello, estudiaremos primeramente el perceptrón, la unidad mínima de procesamiento en estos algoritmos.

Como suele ocurrir en el comportamiento de estructuras avanzadas, la complejidad de estos sistemas emerge del trabajo conjunto de numerosas estructuras mucho más simples. Es nuestro caso, el perceptrón se puede ver como una neurona artificial básica, la estructura más simple dentro de una red neuronal.

El funcionamiento del perceptrón es bastante sencillo. En la figura 2.3 se resume su comportamiento.

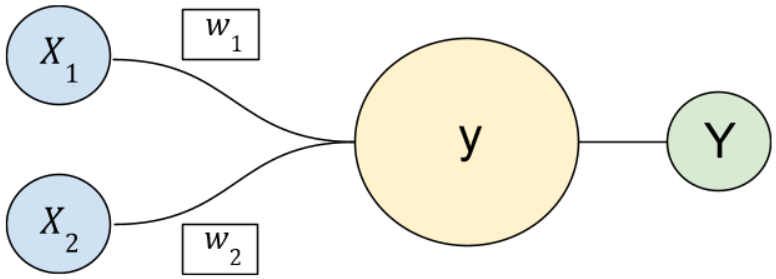


Figura 2.3: Esquema básico del perceptrón. X_1 y X_2 representan las entradas de la neurona, w_1 y w_2 indican el peso que tendrá esa conexión en el resultado final. y hace referencia al resultado de la neurona antes de aplicar la función de activación, mientras que Y hace referencia al resultado final, una vez aplicada la función de activación

La neurona parte de un conjunto de entradas, representadas como X_i. Cada una de estas conexiones de entrada tiene asociado un peso, w_i, con el que podremos

definir con cuánta intensidad afectará cada variable de entrada a la neurona. Con estos valores, el perceptrón realizará una suma ponderada de las entradas y los pesos, tal y como se muestra en la ecuación 2.1:

$$y = w_1 x_1 + w_2 x_2 \tag{2.1}$$

Además de esto, se incluye un sesgo, representado como b, que nos dará cierto control sobre la salida del perceptrón para adecuarlo a nuestro problema en base al umbral que tengamos, representado como θ. De esta forma, la operación del perceptrón quedaría tal y como muestra la ecuación 2.2:

$$y = w_1 x_1 + w_2 x_2 + b \tag{2.2}$$

Tras conocer el resultado de la neurona, lo compararemos con el umbral establecido. Según si el resultado es mayor o menor que el umbral, el perceptrón se activará o no, atendiendo a la función 2.3.

$$f(y) = \begin{cases} 1 & si \quad y \geq \theta \\ 0 & \text{en otro caso} \end{cases} \tag{2.3}$$

De esta forma, una única neurona es esencialmente un modelo de regresión lineal que nos permite separar el espacio n-dimensional en dos regiones, la región en la que la neurona se activa y la región en la que no se activa. Sin embargo, como se suele decir, la unión hace la fuerza. Así, si unimos varias neuronas o perceptrones, conseguiremos crear un modelo mucho más complejo que nos permitirá resolver otros tipos de problemas.

2.1.6. Funciones de activación

Para completar la definición de las neuronas artificiales es necesario mencionar las llamadas funciones de activación. Las funciones de activación son funciones matemáticas ligadas a cada neurona. El objetivo principal de las mismas es introducir la no-linealidad en nuestro modelo.

Tal y como vimos en la sección anterior (2.1.5), una neurona realiza una suma ponderada de sus entradas y pesos más un sesgo, siendo esta una operación lineal. El

problema surge tras descubrir que la composición de muchas funciones lineales da como resultado una función lineal. Si aplicamos esta idea a nuestra red, la composición de muchas neuronas que realizan operaciones lineales dará como resultado otra función lineal, haciendo que todas las capas intermedias resulten de poca utilidad. De forma gráfica, una red neuronal proporciona una solución en un espacio n-dimensional. En ocasiones, esta solución se podrá representar mediante ecuaciones lineales. Sin embargo, en la gran mayoría de problemas reales, dicha solución no será tan simple ya que tendrá fronteras más complejas e imposibles de representar con este tipo de ecuaciones. Será en estos casos donde es imprescindible incluir la no-linealidad en el modelo.

El problema se solventa haciendo uso de las funciones de activación. Al escoger una función de activación no lineal estamos introduciendo la no-linealidad que necesitamos para nuestra red, de forma que podamos concatenar las operaciones de cada una de las neuronas y conseguir modelos más complejos. Así, una neurona realizará de forma general la operación mostrada en la ecuación 2.4, siendo F la función de activación.

$$y = F(\sum(w_i x_i) + b) \tag{2.4}$$

En el ejemplo de la sección anterior se mencionaba que la neurona se activaba o no dependiendo de si el resultado superaba cierto umbral. En realidad, estábamos introduciendo uno de los tipos de funciones de activación, la función de activación escalonada. Sin embargo podemos encontrar numerosos tipos de funciones además de esta. Algunas de ellas se describen a continuación:

- **Sigmoide**

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.5}$$

  Como vemos en su representación gráfica (figura 2.4), normaliza las entradas a un intervalo [0,1] en las salidas, con centro en 0.5. Uno de los problemas que surgen durante el uso de esta función de activación es la fuga del gradiente. Cuando el valor x es muy grande o muy pequeño, la derivada de la función tiende a 0. Si tenemos muchas capas en la red, el gradiente que se propaga hacia atrás durante la aplicación del algoritmo *backpropagation* será cada vez más cercano a 0, propiciando un ratio de aprendizaje bajo en las primeras capas.

- **Tangente hiperbólica**

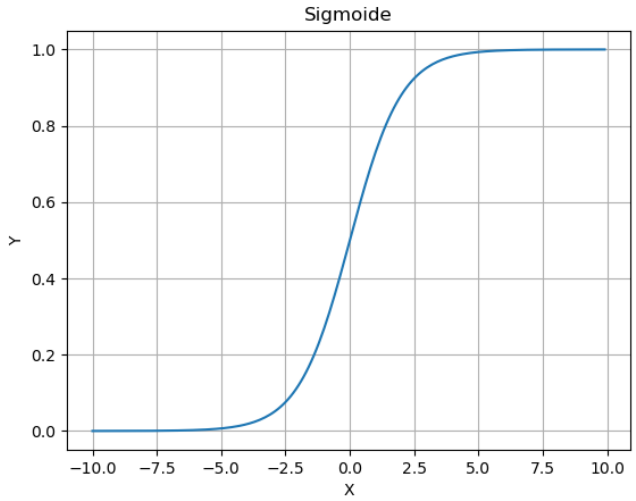


Figura 2.4: Función de activación sigmoide

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.6}$$

Esta función tiene una forma similar a la función sigmoide. En este caso, está centrada en 0 con un intervalo [-1,1]. Su derivada tiene un rango en el eje Y mayor que la función sigmoide, aunque el problema de la fuga del gradiente sigue existiendo. Su representación gráfica queda reflejada en la figura 2.5

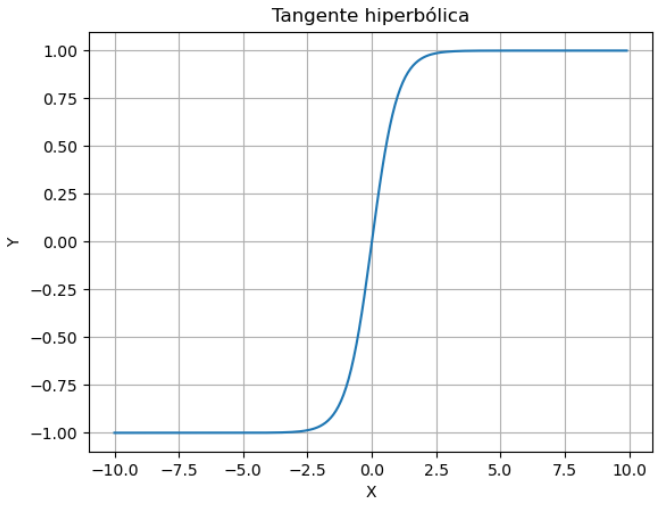


Figura 2.5: Función de activación tangente hiperbólica

- **ReLU**

$$f(x) = max(0, x) \tag{2.7}$$

La función ReLU es una de las más utilizadas en DL. Tiene un cálculo eficiente y el problema de la fuga del gradiente es mucho menor que las anteriores.

Esta función también provoca algunos problemas. Como vemos en la gráfica 2.6, es posible que durante el entrenamiento las neuronas se queden estancadas en la parte negativa de la función, haciendo que nunca se activen y devuelvan siempre 0, haciendo que no aporten nada relevante al modelo. A este problema se le denomina ReLU moribundo.

Por otro lado, el hecho de que la función no sea derivable en $x = 0$ puede provocar problemas durante el proceso de optimización de parámetros de la red neuronal (el cual se describe en el siguiente apartado) ya que el cálculo del gradiente implica calcular la derivada de la función de activación. Lo deseable es que estas funciones sean derivables.

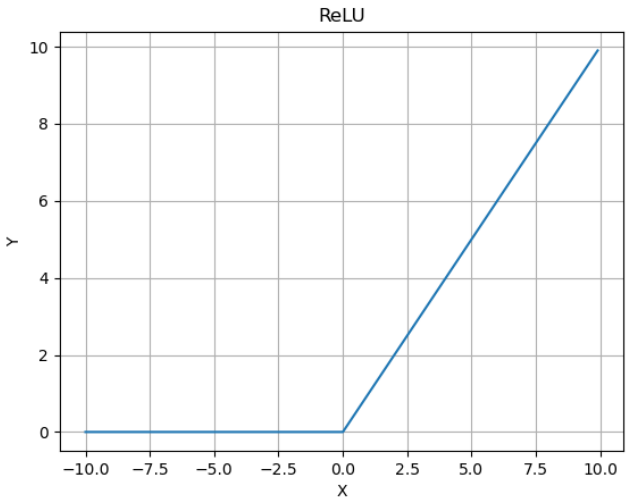


Figura 2.6: Función de activación ReLU

- **Leaky ReLU**

$$f(x) = max(\alpha * x, x) \tag{2.8}$$

Es una modificación de la función ReLU que busca resolver el problema del ReLU moribundo. Alpha se establece con un valor bajo (normalmente $\alpha = 0{,}01$), de forma que la derivada en la parte negativa de la función no sea 0 y evitando por tanto que las neuronas mueran. La representación gráfica para $\alpha = 0{,}1$ se muestra en la figura 2.7.

- **Softmax**

Esta función de activación se utiliza normalmente en la capa de salida de modelos de clasificación de múltiples clases. La salida se representa como un vector

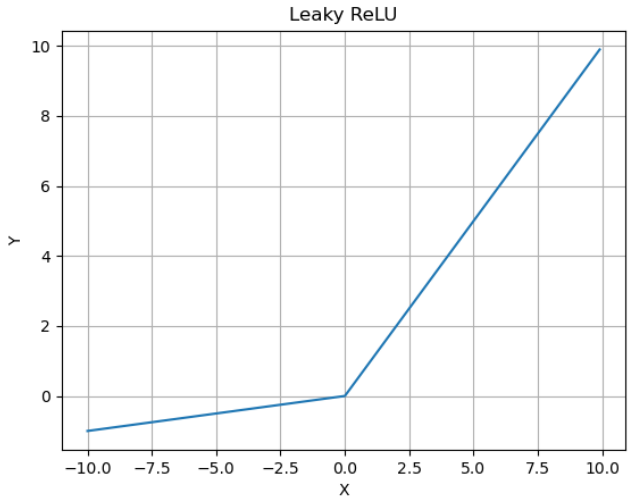


Figura 2.7: Función de activación Leaky ReLU

de tantos valores como clases se quieran clasificar. La sumatoria de estos valores será 1, haciendo que cada valor se pueda utilizar como probabilidades de que la entrada proporcionada pertenezca a la clase correspondiente.

2.1.7. Funcionamiento general de redes neuronales

Hoy en día, la totalidad de redes neuronales que nos encontramos están formadas por múltiples capas de neuronas unidas entre sí tal y como se aprecia en la figura 2.8.

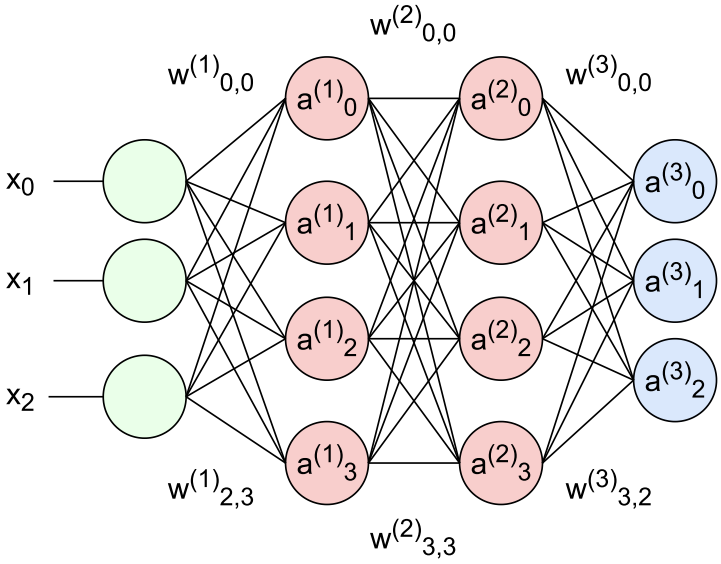


Figura 2.8: Esquema de una red neuronal con los parámetros implicados (entradas x, pesos w y valor de cada neurona a)

Podemos identificar tres grupos básicos de capas. Por un lado tenemos la capa de entrada, representada en la imagen con el color verde, que recogerá las entradas que le proporcionemos a la red neuronal. La última capa, de color azul, nos dará las variables de salida de la red neuronal. Entre estas dos capas, encontramos las llamadas capas ocultas, donde se procesarán los datos de entrada, dando lugar a la salida de la red. Las primeras capas ocultas aprenderán conceptos y patrones básicos que serán procesados por las siguientes capas, elaborando conocimiento cada vez más complejo. Esta estructura es conocida como *Multi-Layer Perceptron*.

Entonces, dada esta estructura, ¿cómo se entrena una arquitectura de este tipo? ¿Cómo aprende una red neuronal? Para responder a estas preguntas, analizaremos las tres fases principales del proceso de aprendizaje de una red neuronal.

1. ***Forwardpropagation***

   Este es el primer paso en el aprendizaje de una red neuronal. Como su propio nombre indica, hace referencia a la fase en la que los datos de entrada son procesados por la red neuronal hacia adelante, desde la entrada a la salida capa a capa. En cada una, las neuronas aplicarán las transformaciones correspondientes a la información hasta llegar a la salida de la red, donde se obtiene el resultado.

   En general, en esta fase la red hará una predicción tomando como base las variables de entrada.

   Cada perceptrón tomará como entrada las salidas de los perceptrones de la capa anterior. De esta forma, visualizando la red mostrada en la figura 2.8 y su notación, el valor de un perceptrón vendrá dado por la expresión 2.9.

$$a_k^{(c)} = f(\sum_{i=0}^{n} w_{i,k} * a_i^{(c-1)} + b_k^c) \tag{2.9}$$

   donde:

   - $a_k^{(c)}$ es el valor del perceptrón que buscamos calcular (capa c y posición k).
   - f es la función de activación del perceptrón.
   - n es el número de perceptrones de la capa anterior que afectan al actual.
   - $w_{n,k}$ es el peso asociado a cada conexión con la capa anterior.
   - $a_n^{(c-1)}$ representa el valor de los perceptrones de la capa anterior.
   - b_k^c es el sesgo del perceptrón.

Este proceso de cálculo del valor de cada perceptrón se puede agilizar haciendo uso del álgebra lineal.

Los valores de una capa c de perceptrones se pueden representar como se muestra en la matriz 2.10.

$$A^{(c)} = \begin{bmatrix} a_0^{(c)} \\ a_1^{(c)} \\ \vdots \\ a_n^{(c)} \end{bmatrix} \tag{2.10}$$

De la misma forma, se pueden establecer los valores de sesgo de cada perceptrón en una capa concreta (2.11).

$$B^{(c)} = \begin{bmatrix} b_0^{(c)} \\ b_1^{(c)} \\ \vdots \\ b_n^{(c)} \end{bmatrix} \tag{2.11}$$

Por último, queda por representar los pesos de cada conexión entre las neuronas de la capa actual con las neuronas de la capa anterior. Esto se refleja en la matriz 2.12, donde k representa el número de neuronas de la capa actual y n el número de neuronas de la capa anterior.

$$W^c = \begin{bmatrix} w_{0,0} & w_{1,0} & \cdots & w_{n,0} \\ w_{0,1} & w_{1,1} & \cdots & w_{n,1} \\ \vdots & \vdots & \ddots & \vdots \\ w_{0,k} & w_{1,k} & \cdots & w_{n,k} \end{bmatrix} \tag{2.12}$$

Con estas definiciones, podemos realizar el cálculo de los valores de los perceptrones de una capa tal y como se muestra en la ecuación 2.13.

$$A^{(c)} = f(W^c * A^{(c-1)} + B^c) \tag{2.13}$$

Siguiendo esta metodología capa tras capa, en la última capa la red conseguirá hacer una predicción en base a la entrada dada.

2. **Cálculo de la función de pérdida o *loss***

   Una vez tenemos el resultado de la red neuronal, debemos comprobar cuán acertada o no es la predicción. La función de pérdida será la encargada de evaluar la desviación entre las predicciones realizadas por la red neuronal frente al resultado que la red debería mostrar. Como es lógico, buscamos minimizar esta función, es decir, que el resultado de la red se parezca lo máximo posible a los valores reales.

   Existen numerosos tipos de funciones de pérdida que podemos usar para nuestro modelo. Una función de este tipo para problemas de regresión podría ser MSE, definida en la expresión 2.14, donde y representa el resultado esperado, $\hat{y}$ el resultado de la red y n el número de predicciones de nuestro modelo.

$$MSE = \frac{1}{n}\sum_{i}^{n}(y_i - \hat{y}_i)^2 \tag{2.14}$$

3. ***Backpropagation***

   En este paso entran en juego dos algoritmos clave: gradiente descendiente [21] y *backpropagation* [15].

   El gradiente descendente es un algoritmo de optimización iterativo capaz de encontrar el valor de las variables de una función que minimicen la misma. Para lograr aplicar este algoritmo debemos calcular el gradiente de la función, que no es más que un vector formado por las derivadas parciales de la función que nos indica la dirección en la que la pendiente asciende de forma máxima en el punto donde se aplique. En nuestro caso nos interesa el negativo del gradiente ya que buscamos encontrar el mínimo de la función. Un factor clave es la llamada ratio de aprendizaje. Este valor es usado para escalar el gradiente y controlar cuánto cambiamos cada variable en la dirección del gradiente en cada iteración. Este proceso se repite de forma iterativa hasta conseguir el mínimo de la función o alcanzar cualquier condición de parada que impongamos.

   Relacionando este concepto con redes neuronales, la función a minimizar será la función de pérdida, mientras que los parámetros a optimizar serán los propios parámetros de cada neurona de la red neuronal, tanto los pesos W como el sesgo o *bias*, b. De esta forma obtendremos los parámetros óptimos de cada neurona para minimizar el error que comete la red neuronal a la hora de procesar las entradas.

   Por su lado, *backpropagation* es el algoritmo que nos permite calcular el gradiente de la función de pérdida con respecto a las variables del modelo. Calcular el gradiente en una red neuronal compleja no es una tarea trivial, ya que la función

de pérdida estará compuesta por miles de parámetros (los pesos y sesgos de cada neurona) y resultará muy complicado calcular la derivada parcial de la función de pérdida con respecto a cada uno de estos parámetros, representada en la ecuación 2.15, donde w_i es el peso de cada una de las neuronas de la red.

$$\frac{\partial C}{\partial w_i} \tag{2.15}$$

El algoritmo *backpropagation* resuelve este problema. Consiste en propagar el error hacia atrás, comenzando a calcular las derivadas parciales con respecto a los parámetros de la última capa, valores que se pueden obtener de una forma mucho más sencilla en comparación con las primeras capas. Con las derivadas de la última capa, podemos calcular fácilmente el resto de derivadas parciales de la capa anterior. De forma iterativa, esta técnica se aplicaría al resto de capas, permitiendo descubrir cuál ha sido la aportación de error de cada neurona al resultado final de la red.

En resumen, para optimizar los parámetros de nuestra red en la fase de entrenamiento usaremos el algoritmo del gradiente descendente que, a su vez, hará uso del algoritmo *backpropagation* para calcular el gradiente de la función de pérdida.

2.2. Modelos más conocidos en *Deep Learning*

A raíz del modelo básico visto en la sección anterior, han surgido nuevas arquitecturas de redes neuronales. En esta sección nos centraremos en explicar algunas de ellas, quedándonos con las características principales y usos de cada una.

Es importante recalcar que estas no son las arquitecturas más modernas de DL en la actualidad. Tomando como base estas arquitecturas que se presentarán, han surgido de nuevo muchos otros modelos más complejos, como podrían ser las redes LSTM (*Long-Short Term Memory*) en el caso de las RNN. En esta sección nos quedaremos únicamente con los modelos básicos de estas arquitecturas.

2.2.1. *Deep Feed Forward Network*

La arquitectura *Deep Feed Forward Network*, también denominada *Multi-layer Perceptron* [22], se refiere a una arquitectura de red neuronal en la que se unen múltiples

capas de perceptrones, con una estructura similar a la de la figura 2.8 a la que se añaden numerosas capas ocultas.

Una de las características principales de este tipo de redes es la ausencia de ciclos, es decir, un perceptrón solo recibirá la información de salida de las neuronas de la capa anterior como sus entradas. La información se dirige "hacia adelante" desde la capa de entrada hasta la salida, de ahí el nombre de la red.

Este tipo de esquema de red neuronal es la base de muchas otras redes más avanzadas como las que se verán a lo largo de la sección.

Usos de una *Deep Feed Forward Network*

El uso de la arquitectura *Deep Feed Forward* proviene de la necesidad de ampliar la capacidad de los perceptrones para resolver problemas. Haciendo uso de un único perceptrón podríamos solventar problemas de regresión simples, pero al añadir más perceptrones, tanto creando nuevas capas de neuronas como ampliando cada una de ellas, podríamos modelar información más compleja.

A pesar de esto, como ya se ha indicado existen otros tipos de redes que podremos aplicar a problemas más específicos como el análisis de imágenes.

2.2.2. Redes neuronales convolucionales

Las CNN [23] son un tipo de red neuronal que está diseñado específicamente para tratar imágenes de forma mucho más eficiente que en una red neuronal tradicional.

Su funcionamiento se basa en múltiples capas de filtros convolucionales que se aplican a la imagen de entrada. La convolución se define como un proceso en el cual un núcleo, denominado *kernel*, recorre la superficie de la imagen. Para cada posición se calculará el producto escalar entre los valores del núcleo y los valores de la imagen que se ven afectados en esa posición. Tras recorrer la imagen completa habrá generado una nueva matriz de salida, como se ve en la figura 2.9, que representará la siguiente capa de la red. La lógica del proceso se encuentra en el filtro a utilizar. Según el filtro que usemos podremos obtener un mapa de características que nos permita identificar distintos tipos de patrones en la imagen.

Normalmente en cada capa no se usa un único filtro, sino que se hace uso de varios de ellos de forma que se puedan identificar un mayor número de patrones.

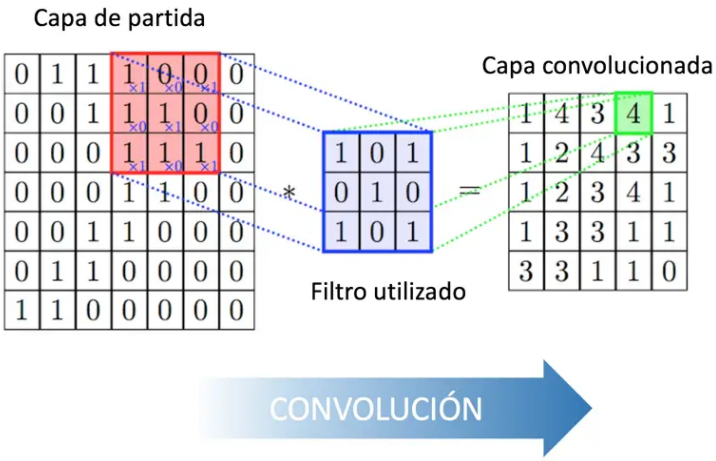


Figura 2.9: Aplicación de un filtro en una red neuronal convolucional. A partir de los datos de la capa de partida y el filtro, se obtiene el resultado en la capa convolucionada.
[24]

Así, el objetivo del entrenamiento de una red neuronal de este tipo será encontrar los mejores parámetros para cada filtro a utilizar, además de un valor de sesgo por cada filtro.

Sin embargo, surge un nuevo problema: el tamaño de las imágenes filtradas. Si mantenemos su tamaño original durante todo el recorrido de la red, el número de parámetros a optimizar pronto se dispararía. La idea, por tanto, es simplificar la información recogida por la capa convolucional, quedándonos con los aspectos más relevantes. Para ello podremos usar diferentes técnicas. Dos de las más importantes son el *max-pooling* y *average-pooling*. Básicamente consisten en dividir la imagen en sectores y, para cada sector, quedarnos con el mayor valor en caso de *max-pooling* o con el valor medio en caso de *average-pooling*, tal y como se aprecia en la figura 2.10.

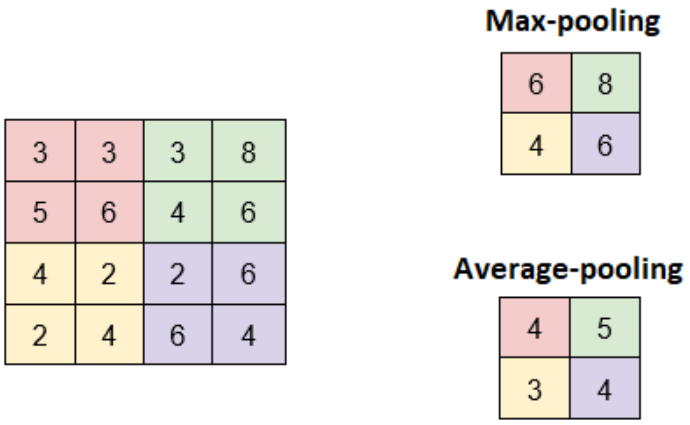


Figura 2.10: Aplicación de *max-pooling* y *average-pooling* en una matriz de enteros.

El último paso es tomar la capa oculta final y conectarla a una capa de neuronas densamente conectadas. Esta capa será la encargada de dar el resultado final a partir

de la imagen original. Por ejemplo, si buscamos resolver un problema de clasificación podríamos introducir una capa con función de activación *softmax*.

Usos de una CNN

Las redes neuronales convolucionales han demostrado tener una gran utilidad en el campo de análisis de imágenes, como clasificación o detección de objetos. El preprocesamiento que se requiere en una CNN es mucho menor que el que se requeriría en un algoritmo tradicional, donde los filtros se especifican y ajustan de forma manual, ya que una CNN es capaz de aprender esos filtros y optimizarlos de forma automática.

2.2.3. Redes siamesas

Las redes neuronales siamesas [25] son un tipo de arquitectura que contiene dos o más subredes idénticas, es decir, con la misma configuración, parámetros y pesos. Son usadas para comparar dos o más entradas en base a la diferencia de los *feature vector* resultantes, vectores que simplifican las entradas eliminando aspectos redundantes o innecesarios.

El funcionamiento de estas redes se puede deducir fácilmente. Cada entrada se introduce en una subred distinta. Si las entradas que se entregan a la red son parecidas, entonces sus *feature vector* serán muy similares. Si las entradas son diferentes, sus *feature vector* también lo serán. De esta forma, midiendo la diferencia entre ambos *feature vector* y estableciendo un umbral adecuado, podremos determinar si las entradas se parecen o no.

Una de las principales ventajas de esta arquitectura frente a otras, como una red convolucional simple, radica en el número de datos necesarios para entrenar al modelo. Supongamos que nos encontramos ante un problema de clasificación entre varias clases. En un modelo CNN serán necesarias numerosas imágenes de entrenamiento de cada clase con el fin de que el modelo aprenda las características de cada una. En un modelo de redes siamesas se usa la técnica denominada *One Shot Learning*. Este método plantea convertir el problema de clasificación en un problema de estimación de diferencia. La idea será entrenar al modelo para calcular la diferencia entre dos imágenes. De esta forma, simplemente teniendo una imagen de cada una de las clases que busquemos clasificar, la red será capaz de asignar la clase que más se parezca a la imagen de entrada.

Por tanto, esta técnica hace que sea necesario un conjunto de datos mucho más pequeño que si usáramos una CNN. Además, su flexibilidad es mucho mayor ya que si introducimos una clase nueva, simplemente necesitaremos una imagen que la identifique, no volver a entrenar el modelo.

Usos de una red siamesa

Las redes siamesas pueden ser útiles en cualquier área donde sea necesario comparar dos o más elementos, ya sean imágenes u otros tipos de datos diferentes.

Un campo interesante donde se han usado redes de este tipo es en la verificación de firmas. La red *SigNet* [26] sigue una estructura de este tipo para reconocer si una firma es auténtica o no.

De la misma forma, una red siamesa puede ser útil en el reconocimiento facial. Con una simple imagen de las caras de las personas a detectar, el modelo podrá reconocer e identificarlas en diferentes imágenes.

2.2.4. Redes neuronales recurrentes

Las RNN [27] son un tipo especial de red neuronal artificial adaptada para trabajar con datos de muestras que no son independientes entre sí.

En una red neuronal tradicional los datos con los que se alimenta son independientes, por ejemplo un conjunto de imágenes etiquetadas. Sin embargo existen problemas en los que los datos sí son dependientes entre sí, como la predicción de la temperatura que hará en unas horas; los valores de temperatura dependen, entre otros aspectos, de los valores de temperatura registrados anteriormente. Es para estos problemas donde entran en juego las RNN. Se distinguen por ser capaces de "memorizar", ya que toman información de entradas anteriores para influenciar en la salida actual.

Esto se puede apreciar en la figura 2.11, que representa el esquema de una RNN. Cualquier salida y_t estará influenciada por su entrada anterior x_{t-1}, además de la entrada actual x_t.

Usos de una RNN

Una RNN puede ser útil en cualquier problema donde se trabaja con series temporales, es decir, observaciones obtenidas a través de diferentes medidas en el tiempo, o problemas de predicción en las que los valores a predecir dependen de los valores que se han dado anteriormente. Una posible aplicación se encuentra en el campo del

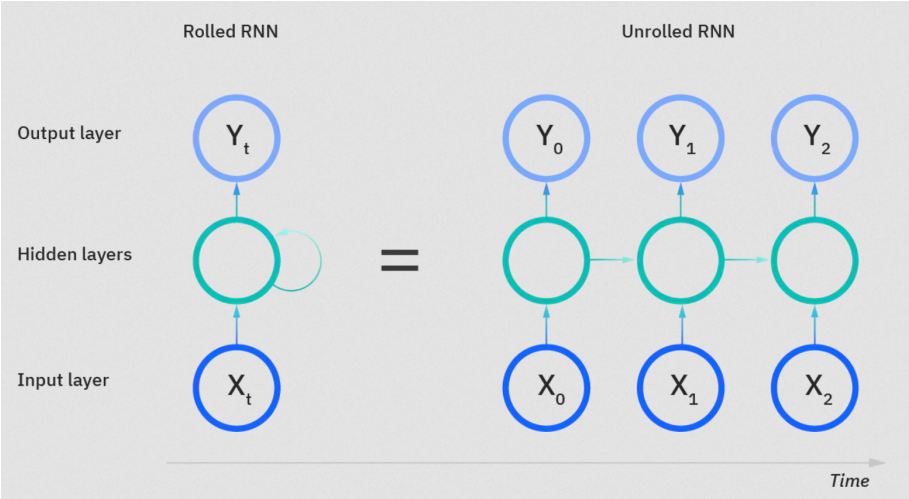


Figura 2.11: Esquema básico de una RNN. X representan las entradas, Y las salidas [28]

procesamiento del lenguaje natural, donde si buscamos predecir la siguiente palabra en una frase será importante conocer el resto de palabras que se dan y su orden.

2.2.5. Redes neuronales generativas adversarias

Las GAN [29] son redes DL capaces de generar nuevo contenido en base a la entrada que se proporcione, ya sea en forma de imágenes, vídeo, voz o texto.

Para esto, hacen uso de dos modelos de red neuronal:

- **Generador:** se encarga de crear el nuevo contenido en base al dominio del problema.
- **Discriminador:** estimará la probabilidad de que un ejemplo provenga del conjunto de entrenamiento en lugar de que haya sido producido por el generador, es decir, clasificará las entradas como verdaderas si provienen del conjunto de entrenamiento o falsas si dictamina que han sido creadas por el generador.

Ambos modelos serán entrenados a la vez. El generador creará un conjunto de ejemplos y estos, junto a los datos de entrenamiento reales, serán dados al discriminador, que los clasificará como verdaderos o falsos. Tras esto se aplicará el algoritmo *Backpropagation* y se actualizará el discriminador, tomando en cuenta los resultados obtenidos para mejorar su eficacia en la clasificación de las siguientes iteraciones. Por su lado, el generador también será actualizado basándonos en si los datos generados

han sido capaces de engañar o no al discriminador haciéndose pasar por verdaderos. Todo este proceso se puede apreciar en la figura 2.12.

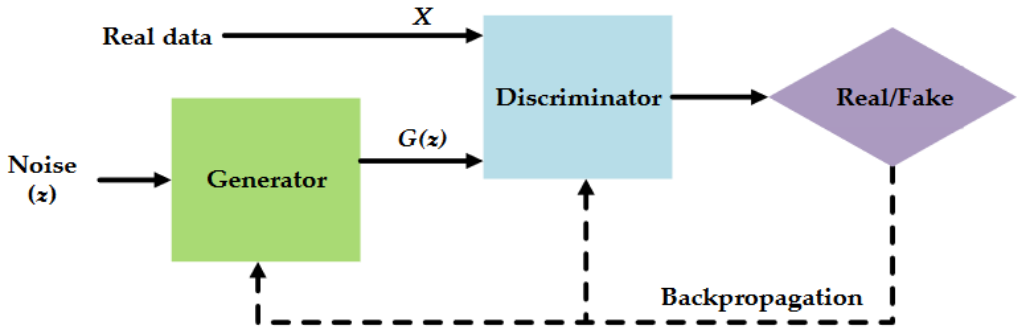


Figura 2.12: Esquema de una GAN. El discriminador estima la diferencia entre la muestra real X y la generada por el generador, G. Con esa diferencia, se ajustan los parámetros tanto del generador (para crear mejores imágenes) como del discriminador (para aprender a diferenciar una imagen real de una falsa).

[30]

Usos de una GAN

Las GAN o redes derivadas de esta se han utilizado en múltiples ocasiones para la generación de contenido. Un buen ejemplo de ello son las *Deep Fakes*, la creación por ordenador de caras falsas que parecen reales. También destacan trabajos en los que se demuestra cómo un modelo de este tipo es capaz de generar dibujos realistas a partir de simples bocetos [31].

Otro aspecto interesante en el uso de estas redes se encuentra en el campo de compresión de vídeo. Nvidia ha desarrollado una técnica de compresión de vídeo [32] que permite ahorrar ancho de banda al mismo tiempo que llega a mejorar la calidad de la imagen en videoconferencias.

2.3. *Autoencoders*

En esta sección nos enfocaremos de forma directa en otro tipo de red neuronal muy utilizado hoy en día, los AE. Se detallará qué es un AE, los tipos que encontramos según su funcionamiento interno y posibles aplicaciones. Esta arquitectura será el pilar fundamental del sistema que se propondrá más adelante para afrontar el problema ya descrito, razón por la cual se dedica una sección completa a explicar su funcionamiento de forma más detallada.

2.3.1. Estructura general de un autoencoder

Un AE [33] es un tipo de red neuronal con una estructura simétrica, donde la capa central de la red representa una codificación de la capa de entrada. El objetivo de un AE será reconstruir las entradas en las salidas bajo ciertas condiciones, buscando que la salida no sea una copia directa de la entrada.

En el interior de un AE podemos encontrar los elementos mostrados en la figura 2.13. En la parte izquierda se encuentra la entrada del AE, x. Esta es codificada en y haciendo uso del codificador, denominado como f. Seguidamente, se vuelve a decodificar por medio de g para obtener la salida, r.

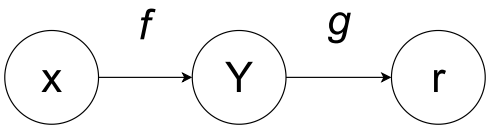


Figura 2.13: Estructura general de un AE. x representa la entrada. Mediante el codificador f se obtiene su representación codificada y, la cual se vuelve a decodificar con g, obteniendo la salida r.
[33]

Esta estructura descrita será trasladada a una red neuronal tal y como se muestra en la figura 2.14, siendo x y r las capas de entrada y salida de la red, y la capa de codificación y f y g capas adicionales del AE encargadas de la codificación y decodificación.

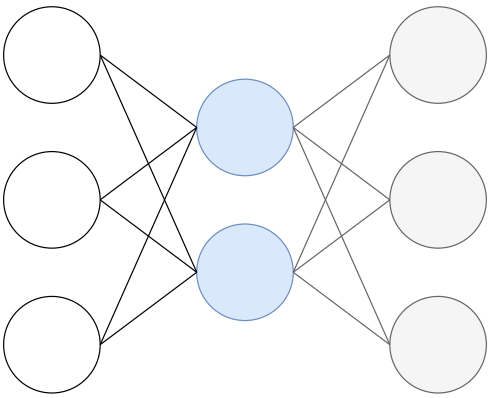


Figura 2.14: Estructura general de un AE plasmada en una red neuronal. La capa de la izquierda representa la entrada, la capa central el estado codificado y la capa derecha la salida del AE.

2.3.2. Tipos de autoencoders

Existen varias formas de clasificar un AE. En este apartado nos centraremos en la clasificación según el tamaño de la capa de codificación y su funcionamiento interno.

En relación a la capa de codificación podemos encontrar dos variantes de AE. Por un lado, hallamos capas de codificación de menor tamaño que la entrada, forzando al modelo a aprender a reconstruir la entrada a partir de una representación más compacta. Por otro lado, existen variaciones en las que la capa de codificación es de mayor tamaño en relación a la entrada.

Enfocándonos en el funcionamiento interno del AE, aparecen numerosas clasificaciones diferentes centradas en distintos aspectos como regularización, tolerancia al ruido o modelos generativos. En este caso abarcaremos tres de ellas, *denoising AE*, *Sparse AE* y AE convolucional. El resto de arquitecturas se pueden estudiar en el artículo referenciado al inicio de la sección.

- ***Denoising AE***

  Una aproximación distinta a la idea original de que las salidas sean idénticas a las entradas es modificar ligeramente este aspecto para aumentar la tolerancia al ruido.

  Un *denoising AE* tendrá una estructura interna idéntica a cualquier otro tipo de AE. La principal diferencia radica en el proceso de entrenamiento. En este paso se generarán entradas ruidosas, es decir, parcialmente destruidas. Sin embargo se seguirán manteniendo los datos de entrada originales como objetivos de las salidas, obligando al AE a reconstruir los datos de forma correcta sin importar el ruido de entrada.

  La principal ventaja de hacer uso de este tipo de AE es que serán mucho más robustos frente a entradas corruptas, permitiendo una mayor tolerancia al ruido.

- ***Robust AE***

  Tal y como hemos analizado, una de las técnicas para lograr que un AE sea capaz de ser más tolerante al ruido es incluir ruido a las entradas del AE, buscando que en la reconstrucción se elimine automáticamente el ruido. Sin embargo, esta no es la única aproximación disponible.

  Una alternativa es modificar la función de pérdida y adecuarla para minimizar el error de reconstrucción a la vez que incrementa la capacidad de soportar el ruido de entrada. Un RAE sigue esta vía. Gracias al uso de una función de pérdida basada en *correntropy* (definida en [34]) se consigue mantener la calidad de los resultados ante entradas ruidosas.

  Dicha función de pérdida queda definida en las ecuaciones 2.16 y 2.17, siendo y_{pred} la predicción del AE, y_{true} el valor real y σ un valor constante que representa el tamaño del *kernel*.

$$Loss = \sum kernel(y_{pred} - y_{true}) \tag{2.16}$$

$$kernel(\alpha) = \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{\alpha^2}{2\sigma^2}) \tag{2.17}$$

Correntropy establece la probabilidad de que dos variables aleatorias sean iguales. Por tanto, un RAE deberá intentar maximizar dicha función o minimizar el negativo de la función.

- ***Sparse AE***

  Una de las cosas a evitar en el comportamiento de un AE es que el modelo copie directamente la entrada en la salida. En una estructura básica de AE se tiene, normalmente, una capa de codificación más pequeña que la capa de entrada de la red, lo que permite que el modelo no se base simplemente en copiar la entrada en la salida, sino en reconstruir la codificación de la entrada. En un *sparse AE* se propone una solución distinta para solventar este problema sin la necesidad de que la capa de codificación sea de menor tamaño, permitiendo que pueda ser de igual o mayor tamaño que la capa de entrada.

  La idea es construir una función de pérdida que, además de penalizar la diferencia entre la entrada y la salida, sancione la activación de neuronas. Así, por cada observación provocaremos que la red aprenda un modelo de codificación y decodificación basado en la activación de unas pocas neuronas, tal y como se muestra en la figura 2.15. Como resultado, habremos limitado la capacidad de la red de memorizar los datos de entrada sin reducir el tamaño de la red y por tanto su capacidad para aprender características de los datos.

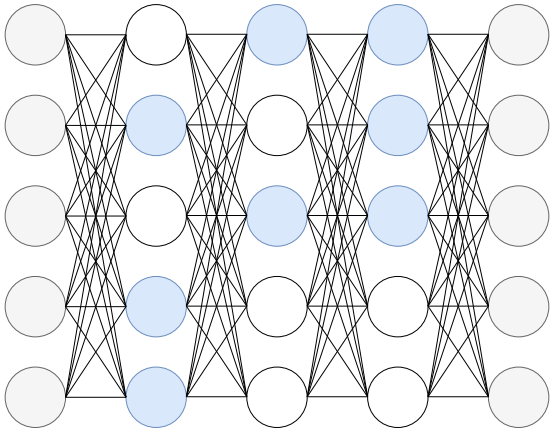


Figura 2.15: Estructura interna de un *Sparse AE*. Las capas grises de los extremos representan las capas de entrada y salida. En las capas ocultas, las neuronas azules representan neuronas que se han activado durante una observación concreta.

- **AE convolucional**

  Un AE básico trabaja con entradas de una dimensión, un vector de datos. Sin embargo, existen otros tipos de datos que no tienen una única dimensión. Es el caso de las imágenes, datos formados por dos o tres dimensiones en caso de incluir más de un canal de color.

  Para solventar este problema pueden surgir varias aproximaciones. Una posible solución sería transformar la imagen de entrada, que en última instancia es una matriz de valores, a un vector, de forma que pueda ser tratada por el AE como cualquier otro tipo de dato.

  Por otro lado, podemos centrarnos en las diferencias que había entre una *Deep Feed Forward Network* y una CNN. Esta última, como se analizó en la sección 2.2.2, conseguía un mejor tratamiento de imágenes al usar filtros sobre las mismas, ya que lograba guardar la coherencia espacial de cada imagen. Siguiendo esta aproximación, surgen los AE convolucionales.

  Un AE convolucional no es más que un AE especializado en el tratamiento de imágenes. Para lograr su objetivo, utiliza capas convolucionales idénticas a las vistas en las CNN. Además, se aplicarán capas *Max Pooling* y *Up Sampling*. El *Max Pooling* es una técnica ya definida en la figura 2.10 que permite reducir el tamaño de las capas convolucionales reduciendo la información que pasa a la siguiente capa. Por su lado, la técnica *Up Sampling* produce justo el efecto contrario al incrementar la información que pasa a la capa posterior, algo deseable en el bloque de decodificación en un AE.

  La estructura simétrica de los AE se mantiene intacta. En la figura 2.16 se muestra un esquema de un AE convolucional donde intervienen capas de convolución, *Max Pooling* y *Up Sampling* [2].

2.3.3. Usos de un autoencoder

La arquitectura de un AE puede ser aprovechada para resolver multitud de problemas. En este caso nos centraremos en tres de ellos: compresión de datos, visualización de datos y clasificación.

- **Compresión de datos**

  Una aplicación lógica de un AE se basa en la compresión de grandes cantidades de datos. En un lado codificamos y comprimimos los datos, obteniendo la capa

[2] La visualización del modelo se ha conseguido utilizando la herramienta Net2Vis [35].

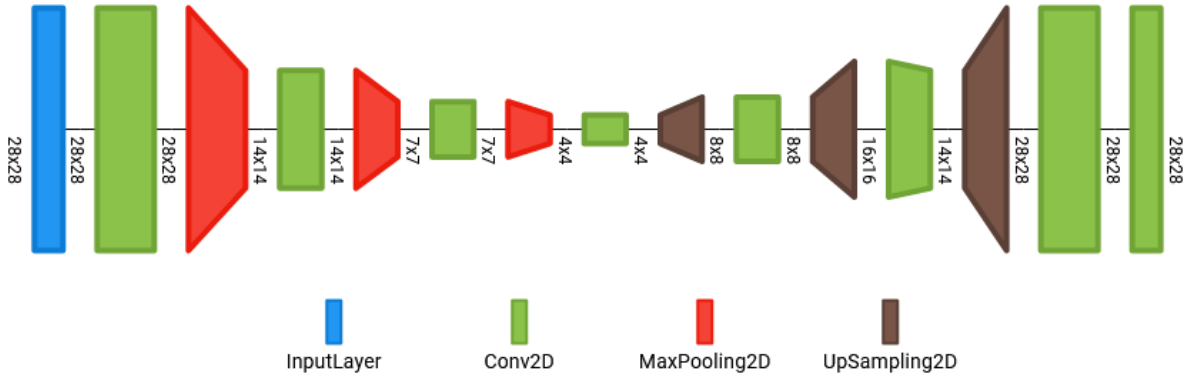


Figura 2.16: Arquitectura de un modelo AE convolucional. Aparecen cuatro tipos de capas: capa de entrada (InputLayer), capas convolucionales (Conv2D), capas de *max pooling* (MaxPooling2D) para reducir el tamaño de las salidas de las capas convolucionales y capas *up sampling* (UpSampling2D) para incrementar el tamaño de las salidas de las capas convolucionales.

de codificación mencionada. En otro lado, teniendo el bloque de decodificación del AE, volvemos a reconstruir los datos a su estado inicial.

Sin embargo, irremediablemente durante este proceso de compresión y descompresión utilizando un AE se producirán pérdidas en los datos, por lo que esta técnica solo será válida en aquellos ámbitos donde la compresión con pérdida sea aceptable.

Normalmente las imágenes son el tipo de dato utilizado para la compresión de datos. A pesar de esto, esta técnica se puede aplicar sobre otros tipos de información. Por ejemplo, en el artículo [36] se presenta un modelo DL basado en AE para la compresión de lenguaje escrito con el objetivo de acortar una oración manteniendo su significado.

- **Visualización de datos**

  La mayoría de problemas actuales toman como entrada datos n-dimensionales. La visualización de estos datos podría ser de utilidad para facilitar la interpretación del problema, pero cuando los datos están formados por más de tres dimensiones la representación de los mismos se vuelve compleja.

  Haciendo uso de un AE se podría resolver este problema. Como se ha descrito, un AE reproduce las entradas en las salidas obteniendo durante el proceso una representación compacta de los datos de entrada. Si el AE se configura adecuadamente, dicha representación podrá estar representada en dos o tres dimensiones, permitiendo una visualización más cómoda. En [37] se muestra un ejemplo de reducción de dimensionalidad de los datos.

- **Clasificación**

  En este ámbito podemos apreciar varias aproximaciones. Una de ellas es utilizar el AE como complemento para un clasificador estándar basado en otras técnicas. El AE podría realizar un preprocesamiento de los datos para reducir la dimensión de los mismos manteniendo las características más importantes, facilitando la posterior tarea del clasificador.

  Por otra parte, se pueden plantear soluciones donde el AE sea el centro de la clasificación. Su funcionamiento es similar a sistemas de detección de anomalías. Pongamos por caso el propio tema del TFG, detectar la presencia de animales en una cámara de fototrampeo. Si entrenamos a un AE para que sea capaz de reconstruir imágenes vacías, es decir, entrenamos al modelo con imágenes de este tipo, la futura reconstrucción de imágenes en las que aparezcan animales tendrá una variación (o error) mayor, algo que podemos cuantificar para detectar la presencia o ausencia de animales. Esta es la idea sobre la que gira la propuesta desarrollada en este TFG.

2.4. Antecedentes del problema

En esta sección se hará un breve recorrido sobre las técnicas ya desarrolladas para solucionar el problema que se analizará en el TFG, la detección automática de imágenes vacías procedentes de dispositivos de fototrampeo frente a aquellas que sí contienen animales.

Principalmente encontramos dos aproximaciones distintas en la resolución del problema: técnicas de visión artificial y técnicas basadas en redes neuronales.

En cuanto al uso de técnicas de visión artificial, encontramos el estudio [38]. Dicho estudio se basa en que cada vez que la cámara detecta una actividad, se obtendrá una serie de imágenes en un corto periodo de tiempo. Estas imágenes tendrán el mismo fondo, pero se encontrarán diferencias entre ellas en caso de que haya un animal en movimiento. También se tiene en cuenta la existencia de niebla en el ambiente. La niebla provocará diferencias entre las imágenes, pero no afectará a los bordes. En base a este razonamiento, en caso de que exista niebla se aplicará el algoritmo Sobel [39] para la detección de bordes y se analizarán los cambios en las imágenes resultantes.

Por otro lado encontramos técnicas basadas en redes neuronales. Concretamente, en el estudio [40] se usa una red CNN con arquitectura *ResNet-18* y un conjunto de

entrenamiento de 3.367.383 imágenes. La red era capaz de diferenciar entre imágenes con animales y imágenes vacías, además de realizar una clasificación del tipo de animal que encuentra.

Como última referencia, encontramos un TFG presentado en la Universidad de Jaén por Jesús Enrique Cartas Rascón [41]. En él se abordaba el problema para la detección de imágenes vacías mediante dos métodos: una aproximación con AE, reconstruyendo las imágenes y manteniendo la premisa de que las imágenes vacías tendrían menor error que las imágenes con animales, y otra aproximación mediante técnicas de visión por computador para la detección de bordes.

Capítulo 3

OBJETIVOS

En este breve capítulo se expondrán los principales objetivos del TFG, avanzando desde el objetivo principal hasta desembocar en objetivos más específicos, detallando cada uno de ellos.

3.1. Objetivo principal

El objetivo global de este trabajo será desarrollar un sistema software capaz de procesar imágenes de fototrampeo para el descarte automático de imágenes vacías (imágenes que no contienen ningún animal) haciendo uso de técnicas de aprendizaje profundo (DL).

3.2. Objetivos específicos

A continuación se presentarán todos los objetivos específicos que se buscan alcanzar con la realización de este TFG.

Adquisición de conocimiento

Una de las principales metas que se buscan durante la realización de un TFG es la adquisición de conocimiento por parte del estudiante, que permita complementar y ampliar toda la información vista a lo largo del grado. En este caso, con un TFG basado en DL, se ampliarán los conceptos vistos en asignaturas como Inteligencia

artificial, Procesamiento de información visual o Minería de datos. Concretamente, se persiguen los siguientes objetivos:

- Obtener una visión general del campo de Ciencia de datos y los métodos esenciales de Minería de datos. Entre las áreas sobre las que se trabajarán destacan:

    1. Creación y limpieza de un *dataset*. Los datos con los que se trabajan en un proyecto de Ciencia de datos deben ajustarse en una fase de preprocesamiento, eliminando inconsistencias, datos incompletos o datos erróneos. El objetivo será crear un *dataset* unificado y listo para ser utilizado.
    2. Desarrollo de algoritmos clave en ciencia de datos que permitan resolver el problema de detección de imágenes vacías propuesto, en este caso algoritmos de *clustering* y DL.
    3. Experimentación y análisis e interpretación de resultados, una de las fases clave de cualquier proyecto experimental.

- Obtener una visión general del campo de DL. Se explorarán técnicas basadas en el uso de redes neuronales artificiales para el tratamiento de datos. Concretamente se estudiarán modelos de redes neuronales centrados en problemas de clasificación.

Formación de un *dataset* de imágenes

Los datos con los que se trabajarán, en este caso imágenes de fototrampeo, provienen de fuentes de datos distintas y pueden no tener el formato adecuado para su procesamiento en redes neuronales (modelo de color, tamaño, proporción altura-anchura, etc), requiriendo una fase previa de limpieza y preprocesamiento. Debido a esto, uno de los principales objetivos será la creación de un *dataset* de imágenes. Concretamente, se completarán los siguientes objetivos:

1. Recopilar un conjunto de imágenes de fototrampeo.
2. Etiquetar de forma manual el conjunto de imágenes en las que aparecen imágenes de animales. Se trabajará con técnicas de aprendizaje supervisado, por lo que el conjunto de datos debe estar etiquetado.
3. Llevar a cabo un preprocesamiento y limpieza sobre el *dataset*, eliminando cualquier inconsistencia presente en el conjunto de imágenes y unificando el formato de todas ellas.

4. Creación y separación del conjunto de entrenamiento, que será utilizado para el entrenamiento de los modelos implementados, y del conjunto de *test*, con el que se verificarán los resultados del modelo.

Estudio previo y experimentación

Para lograr vencer el problema, será necesario hacer uso de un conjunto de herramientas con las que trabajar. En la actualidad existen numerosas formas de trabajar con redes neuronales, por lo que se deberá decidir cuáles de todas ellas utilizar. Los objetivos que se persiguen son:

1. Estudiar distintos tipos de redes neuronales existentes y escoger el que mejor se adapte al problema, haciendo hincapié en aquellos modelos específicos para el análisis de imágenes.
2. Estudiar diversos *frameworks* y librerías centradas en DL y elegir una de ellas para la implementación de los modelos.
3. Desarrollar los modelos de aprendizaje profundo adecuados para la detección de imágenes que contienen animales.

Evaluación del modelo

El último objetivo que se presenta es la evaluación del modelo DL desarrollado.

Este objetivo consistirá en analizar e interpretar los resultados obtenidos tras la aplicación del modelo, extrayendo y detallando las conclusiones a las que se han llegado tras la finalización del proyecto.

Capítulo 4

MATERIALES Y MÉTODOS

En este capítulo se definirán los materiales con los que contamos para el desarrollo del TFG y el preprocesamiento de los mismos para facilitar su utilización. Además, se concretarán las herramientas que se usarán para el desarrollo del modelo DL.

4.1. Estudio inicial de las fuentes de datos

El primer aspecto a tener en cuenta son los datos con los que alimentaremos el modelo. En nuestro caso trabajaremos con un conjunto de fotografías obtenidas con dispositivos de fototrampeo instalados en diferentes parques naturales de España. Estas imágenes pertenecen a WWF/Adena [1], una organización mundial dedicada a la conservación de la naturaleza.

La BBDD está compuesta por dos fuentes de datos que siguen una estructura similar: varios grupos de imágenes separados en carpetas y un fichero en formato CSV con la información de cada archivo de imagen (nombre del fichero y especie del animal que aparece, entre otros datos).

La primera de ellas, a la que denominaremos a partir de ahora como BBDD-1, contiene cuatro grupos de carpetas, *Lynx pardinus*, *Meles meles*, *Oryctolagus cuniculus* y *Vulpex vulpex*. Cada carpeta tiene en su interior imágenes de la especie animal que se indica en el propio título, haciendo referencia a linces, tejones, conejos y zorros respectivamente.

[1] La página principal de la organización en España se puede visitar en el siguiente enlace: https://www.wwf.es/

La segunda fuente de datos, a la que llamaremos BBDD-2, contiene 18 subcarpetas nombradas como *L1, L2, ... , L9* y *N1, N2, ... , N9*. Estos nombres hacen referencia a diferentes localizaciones. Así, cada carpeta contendrá fotografías capturadas en una localización concreta diferente de las del resto de carpetas.

En conjunto, las dos fuentes de datos contienen un total de 96 372 imágenes. Tras una inspección visual, podemos apreciar la existencia de varios tipos de fotografías. En algunas de ellas aparecen animales, otras se encuentran vacías y, en menor cantidad, hay imágenes en las que aparecen carteles, humanos o vehículos.

4.2. Análisis exploratorio de las fuentes de datos

Tras un estudio inicial de las fuentes de datos disponibles, procedemos a realizar una inspección más profunda de las mismas, analizando el formato de las imágenes y su contenido.

4.2.1. Información sobre las imágenes

Las imágenes han sido realizadas con dos cámaras distintas:

- **Modelo UV562**: UOV como fabricante de la cámara.
- **Modelo SG560VB**: BolyMedia como fabricante de la cámara.

A pesar de haber sido fotografiadas con diferentes cámaras, todas las imágenes analizadas poseen las mismas características. El formato de las imágenes es JPG, con unas dimensiones comunes de 2 560 x 1 920 píxeles (ancho x alto) en el espacio sRGB. Se muestra una imagen de ejemplo en la figura 4.1.

4.2.2. Análisis de las subcarpetas

Como ya se ha descrito, tanto BBDD-1 como BBDD-2 están formadas por un conjunto de subcarpetas, donde se encuentran las imágenes. En este punto se analizará el contenido de las dos fuentes de datos.

Figura 4.1: Imagen aleatoria extraída de la BBDD

Con este objetivo en mente, se han desarrollado dos *scripts* escritos en Python, cada uno centrado en una de las dos fuentes de datos. Su funcionamiento es sencillo, recorreremos cada fuente de datos y registramos todas las imágenes que se encuentren. Para cada imagen, se realizará una búsqueda en el fichero CSV correspondiente y se clasificará según haya animales o no en la fotografía.

Las estadísticas recogidas tras ejecutar ambos programas se recogen en la tabla 4.1 [2].

4.2.3. Objetivos del preprocesamiento

Tras analizar las fuentes de datos disponibles hemos creído conveniente llevar a cabo una fase de preprocesamiento de los datos.

El objetivo del preprocesamiento de la BBDD será reorganizar y unificar todos los archivos en varios grupos:

- **Animales:** Grupo donde se encontrarán todas las imágenes en las que aparezcan animales.

[2] En la tabla, se entiende por BBDD a la unión de BBDD-1 y BBDD-2

Carpeta	Total	Con animales	Sin animales
L1	8 438	533	7 905
L2	8 419	688	7 731
L3	7 858	265	7 593
L4	2 595	1 086	1 509
L5	635	364	271
L6	4 314	1 480	2 834
L7	2 633	1 136	1 497
L8	2 504	958	1 546
L9	657	132	525
N1	1 173	32	1 141
N2	1 616	267	1 349
N3	347	171	176
N4	402	113	289
N5	1 392	32	1 360
N6	210	129	81
N7	2 156	222	1 934
N8	2 105	78	2 027
N9	881	539	342
Lynx pardinus	41 652	41 652	0
Meles meles	1 151	1 151	0
Oryctolagus cuniculus	2 189	2 189	0
Vulpex vulpex	3 045	3 045	0
BBDD completa	96 372	56 262	40 110

Tabla. 4.1: Número de imágenes en el estado inicial de la BBDD. No se han tenido en cuenta imágenes irrelevantes, como fotografías a carteles, o imágenes que no aparecen en los archivos CSV

- **Vacío:** Grupo donde se encontrarán todas las imágenes en las que no aparezcan animales.

- **Descartes:** Grupo donde se encontrarán las imágenes que no sirven para el problema. Se incluyen fotografías realizadas manualmente a carteles u otros identificadores, además de imágenes que durante el proceso de preprocesamiento no puedan ser tratadas correctamente debido a corrupciones en las mismas o la imposibilidad de clasificarlas en una de las dos categorías por no existir una

referencia a la imagen en los archivos CSV. Estas imágenes no se tendrán en cuenta.

4.3. Preprocesamiento de los datos

En esta sección se analizará de forma detallada el preprocesamiento de las fuentes de datos implicadas. Constará de tres fases: selección del tamaño de las imágenes, unificación de las fuentes de datos y eliminación de metadatos asociados a las imágenes.

4.3.1. Selección del tamaño de las imágenes

El objetivo del modelo será utilizar todas las imágenes de entrenamiento disponibles con el fin de aprender patrones de las mismas y, en el ámbito de este proyecto, saber diferenciar entre aquellas en las que aparecen animales y las que no. Como es lógico, las imágenes deben tener una calidad lo suficientemente aceptable como para que se distingan los animales. A simple vista podría parecer una buena idea utilizar imágenes con la mayor calidad posible, así el modelo podría identificar y diferenciar cada elemento de la imagen. Sin embargo, surgen varios problemas.

Por un lado la memoria limitada del sistema. El ordenador deberá ser capaz de procesar muchas de estas imágenes de forma simultánea y, en caso de que estas sean demasiado grandes, puede faltar memoria en el equipo.

Por otro lado aparece el problema del tiempo de ejecución. Entrenar a un modelo DL, con la optimización de los miles de parámetros que conlleva, puede implicar mucho tiempo. Este tiempo de ejecución aumenta a mayor calidad de imagen se use.

Las imágenes originales que encontramos en las fuentes de datos poseen un tamaño de 2 560 x 1 920 píxeles. En este punto haremos un breve estudio para identificar qué tamaño podría ser adecuado para nuestro problema.

Cabe recalcar que la manera óptima de llevar a cabo esta tarea es hacer las pruebas utilizando el propio modelo y comprobando cómo se comporta ante cada tamaño de imagen. Sin embargo, debido a la falta de recursos y tiempo, creemos que una inspección visual de las imágenes será suficiente.

Para hacer el estudio se realizará un análisis visual sobre un subconjunto aleatorio de imágenes tras aplicar un escalado sobre la imagen a un ancho de 512, 384, 256 y 192 píxeles, manteniendo la proporción de la imagen original. En la figura 4.2 se puede observar el escalado de una de las imágenes utilizadas.

(a) Tamaño 512 píxeles (ancho)

(b) Tamaño 384 píxeles (ancho)

(c) Tamaño 256 píxeles (ancho)

(d) Tamaño 192 píxeles (ancho)

Figura 4.2: Estudio sobre el tamaño de las imágenes. Se muestra una misma imagen modificando el ancho de la misma a 512(a), 384(b), 256(c) y 192(d) píxeles. La proporción $alto\ x\ ancho$ se mantiene invariable.

En las imágenes presentadas se puede apreciar cómo la diferencia no es demasiado notable entra las dos primeras (tamaño 512 y 384 píxeles), manteniendo una calidad aceptable en ambas. Sin embargo, en las dos imágenes de la parte inferior (tamaño 256 y 192 píxeles) sí se aprecia una diferencia de calidad mayor.

En base a estos resultados y el breve análisis, decidimos que se utilizarán imágenes con tamaño 384 píxeles de ancho. En cuanto a la dimensión referente a la altura, se mantendrá la proporción inicial. Así, quedará una altura de 288 píxeles.

4.3.2. Unificación de las fuentes de datos

El resultado de la unificación de las dos fuentes de datos será una única BBDD con dos carpetas, Animales y Vacío, que contendrán el tipo de imágenes según su identificador, imágenes con animales e imágenes vacías, respectivamente. Además, como ya se ha seleccionado el tamaño de las imágenes, se escalarán todos los elementos de la BBDD a un tamaño común, 384x288 píxeles (ancho y alto, respectivamente).

Para esta tarea vuelvo a utilizar los *scripts* en Python usados en la sección anterior. En este caso, se modifican para que en caso de analizar una imagen con un animal en su interior se mueva a la carpeta Animales, mientras que si la imagen está vacía se moverá a la carpeta Vacío. En este *script* se incluyen también los procesos de escalado de las imágenes, con lo que las imágenes finales tendrán el tamaño adecuado.

Por otro lado, se eliminarán aquellas imágenes que formen parte del conjunto Descartes (ver sección 4.2.3).

El resultado después del preprocesamiento se puede ver en la tabla 4.2.

Carpeta	Nº Imágenes	Porcentaje respecto al total
Animales	55 070	59.50
Vacio	37 503	40.50
Total	92 573	100.00

Tabla. 4.2: Estado de la BBDD tras el preprocesamiento

4.3.3. Eliminación de metadatos asociados a las imágenes

Todas las imágenes analizadas tienen en la parte inferior una barra en la que se muestran distintos parámetros de la imagen como la fecha y hora en la que se tomó. Esta barra debería ser eliminada, reduciendo así el ruido se entrada de la red neuronal.

Con este objetivo, se ha desarrollado un *script* capaz de recortar el borde inferior de la imagen y guardar el resultado. El problema que surge es dónde establecemos el umbral de corte de forma que se recorte toda la barra de información pero no se pierdan demasiados detalles de la propia imagen. El tamaño de la barra no es común a todas las imágenes, sino que varía entre unas y otras. Tras un análisis visual del dataset, se ha comprobado que modificando el tamaño de la imagen de 384x288 a 384x256 se consiguen eliminar todas las barras de información. En la figura 4.3 se

muestra un ejemplo. Este recorte se aplicará sobre todo el conjunto de imágenes disponibles.

Figura 4.3: Comparación de imágenes antes de aplicar el recorte (izquierda) y después de aplicar el recorte (derecha).

4.4. Elección de las herramientas para el desarrollo del modelo

Tras realizar el preprocesamiento de las fuentes de datos, el siguiente paso lógico será decidir las herramientas que se utilizarán en el desarrollo del proyecto. En esta sección se discutirán posibles alternativas, mostrando sus ventajas e inconvenientes para finalmente determinar cuáles de ellas serán escogidas.

4.4.1. Lenguaje de programación

Hoy en día existen numerosos lenguajes de programación para el desarrollo de proyectos informáticos. Saber cuál de ellos escoger puede resultar en una tarea compleja. En este apartado nos centraremos en discutir el lenguaje de programación que se empleará para el desarrollo de modelos DL.

Los lenguajes estudiados para esta tarea serán Python, R y Java. Estos lenguajes han sido escogidos en base a dos criterios fundamentales. Por un lado, la popularidad general de estos lenguajes de programación para el desarrollo ML y DL frente al resto. Por otro lado, mi experiencia en estos lenguajes, superior en comparación a otros lenguajes de programación, como podría ser Julia. Una vez decididas las mejores opciones, se determinará cuál de ellas se escogerá teniendo en cuenta diferentes

aspectos, como las características del lenguaje, la popularidad del mismo, la cantidad y calidad de librerías DL y mi experiencia personal en cada uno.

Características propias del lenguaje

Python es un lenguaje de programación a alto nivel ampliamente utilizado en dominios como programación orientada a objetos, procesamiento del lenguaje natural o visualización de datos, entre otros. Sin embargo destaca su uso en áreas centradas en el análisis de datos. Por otro lado, la sintaxis de este lenguaje es muy sencilla y simple, facilitando la legibilidad del mismo.

Java es un lenguaje de programación orientado a objetos. Es uno de los lenguajes de programación más utilizados en la actualidad. A diferencia de Python, es un lenguaje compilado, lo que hace que posea una velocidad de ejecución mayor.

R por su lado es un lenguaje interpretado centrado en el análisis estadístico. Además, cuenta con herramientas poderosas para la visualización de datos.

En este aspecto, el lenguaje de programación que se utilice no es muy determinante, ya que en última instancia el código que se ejecutará será el de la propia librería, independientemente del lenguaje que se use para llamar a sus métodos y funciones.

Popularidad

Como podemos ver en la figura 4.4, Python se alza sobre el resto de lenguajes en puestos de trabajo relacionados con ML o ciencia de datos, seguido de Java y dejando en tercer lugar a R. A pesar de esto, es interesante apreciar cómo los 3 lenguajes principales siguen una tendencia en alza similar.

Librerías centradas en DL

En los tres lenguajes analizados existen librerías específicas para el desarrollo de modelos DL y ML. En Java encontramos librerías como *Deeplearning4j*, centrada en la creación de redes neuronales. Por otro lado, *Keras* y *Tensorflow*, dos de las librerías para la creación de modelos DL más utilizadas, se encuentran en Python y R.

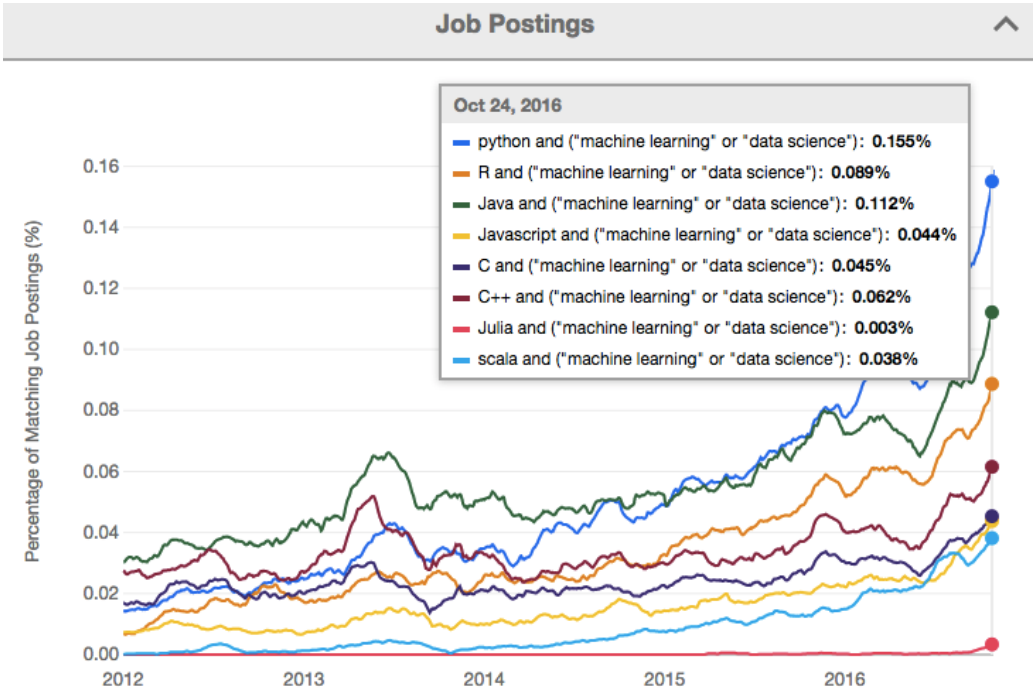


Figura 4.4: Resultado de un estudio sobre los puestos de trabajo en *machine learning* o *data science* y el lenguaje de programación asociado.

[42]

Experiencia personal

Mi experiencia personal está limitada principalmente a los proyectos realizados en el grado de Ingeniería Informática. La experiencia que poseo en Python con respecto a R, lenguaje que apenas he utilizado, es notablemente mayor. Centrándome en Java, es cierto que en estos años he adquirido un conocimiento bastante amplio del lenguaje, aunque teniendo en cuenta que en este último año apenas apenas he escrito programas Java, me siento mucho más cómodo con Python.

Elección del lenguaje

Como se ha analizado, los tres lenguajes de programación son válidos para la tarea a realizar en este TFG. Sin embargo, en base a la popularidad y principalmente a la experiencia y preferencias personales, se ha decidido escoger **Python** como el lenguaje de programación que se utilizará para desarrollar los modelos.

4.4.2. Librería específica para desarrollo *Deep Learning*

La siguiente herramienta a escoger será la librería que se utilizará para el desarrollo de redes neuronales. En la actualidad existen numerosas librerías disponibles especialmente en Python, lenguaje de programación escogido. Sin embargo, muchas de ellas han dejado de estar activas en los últimos años. En la figura 4.5 se puede ver un resumen de los *frameworks* DL y algunas de sus características.

Software	Initial release	Actively developed	Open source	Written in	CUDA support
Wolfram Mathematica	1988	Yes	No	C++, Wolfram Language, CUDA	Yes
MATLAB + Deep Learning Toolbox	1992	Yes	No	C, C++, Java, MATLAB	Yes
Dlib	2002	N/A	Yes	C++	Yes
Torch	2002	No	Yes	C, Lua	Yes
OpenNN	2003	N/A	Yes	C++	Yes
Intel Math Kernel Library	2003	Yes	No		No
Theano	2007	No	Yes	Python	Yes
Neural Designer	2012	N/A	No	C++	No
Caffe	2013	No	Yes	C++	Yes
Deeplearning4j	2014	N/A	Yes	C++, Java	Yes
Intel Data Analytics Acceleration Library	2015	N/A	Yes	C++, Python, Java	No
Apache SINGA	2015	N/A	Yes	C++	Yes
Chainer	2015	No	Yes	Python	Yes
Keras	2015	Yes	Yes	Python	Yes
Apache MXNet	2015	Yes	Yes	Small C++ core library	Yes
TensorFlow	2015	Yes	Yes	C++, Python, CUDA	Yes
BigDL	2016	N/A	Yes	Scala	No
Microsoft Cognitive Toolkit (CNTK)	2016	No	Yes	C++	Yes
PyTorch	2016	Yes	Yes	Python, C, C++, CUDA	Yes
Flux	2017	Yes	Yes	Julia	Yes
PlaidML	2017	Yes	Yes	Python, C++, OpenCL	No

Figura 4.5: Características de librerías para el desarrollo de modelos DL. [43]

Si nos centramos en aquellas escritas en Python, encontramos librerías como *Theano*, *Intel Data Analytics Acceleration Library*, *Chainer*, *Keras*, *Tensorflow*, *Pytorch* y *PlaidML*, aunque hay otras como *Caffe* que, si bien está escrita en C++, también tiene su propia adaptación a Python.

Sin embargo, lo ideal es escoger una librería que esté activa actualmente y que soporte CUDA, lo que permitirá hacer uso de la GPU durante el entrenamiento del modelo, acelerándolo considerablemente. Al establecer estos requisitos, aparecen únicamente tres librerías, *Keras*, *Tensorflow* y *Pytorch*. Si tenemos en cuenta que *Keras* fue integrada dentro de Tensorflow en el año 2017, nos quedan solo las dos opciones restantes.

Analizando las estadísticas referentes al número de búsquedas en Google de cada *framework* en la figura 4.6, podemos ver cómo ambos están prácticamente igualados desde mediados de 2019, haciendo ver que las dos son buenas opciones para la creación de modelos DL, superando *Pytorch* ligeramente a *Tensorflow*.

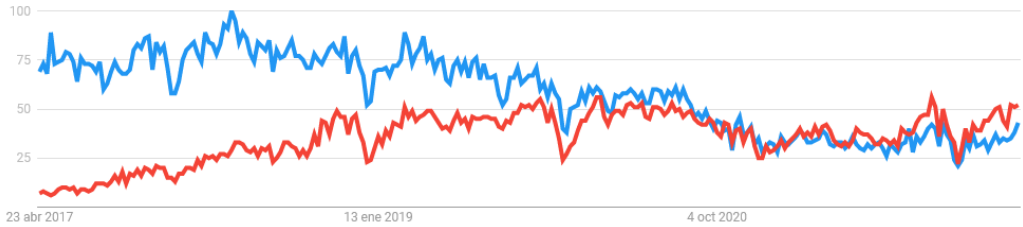


Figura 4.6: Comparación y evolución del número de búsquedas en Google para *Tensorflow* (azul) y *Pytorch* (rojo) desde el 23 de abril de 2017 hasta el día de hoy
[44]

Como ambas opciones siguen estando activas y mantienen una comunidad amplia, se consideran buenas candidatas ya que se podrán conseguir los objetivos planteados con cualquiera de ellas. Para el desarrollo de este TFG, se ha decidido utilizar el *framework Tensorflow*. Esto nos permite también incluir otra de las librerías más usadas para DL, *Keras*. Por tanto, la elección final es la **combinación de *Tensorflow* y *Keras***.

4.5. Estructura del modelo propuesto

En esta sección se analizará la estructura interna del modelo propuesto. Además, se documentará la experimentación realizada hasta llegar al modelo final.

4.5.1. Elección del modelo de red neuronal

El primer paso es la elección del modelo de red neuronal sobre el que se trabajará para la resolución del problema presentado en el TFG.

Partimos del hecho de que el conjunto de datos está formado por imágenes, lo que nos indica que utilizar un modelo de red convolucional es idóneo para el problema. La elección más simple sería escoger una red neuronal convolucional centrada en clasificación que se encargue de determinar si una imagen contiene o no un animal. Sin embargo se abren otras opciones que también pueden ser interesantes.

En [45] se utiliza un AE convolucional para la detección de anomalías en redes, concretamente para la detección temprana de ciberataques. En primer lugar se entrena al AE para reconstruir en las salidas los datos de la red en su estado normal de forma que se minimice el error de reconstrucción. De esta forma, cuando el AE intenta reconstruir datos de la red en el momento de ser atacada, el error de reconstrucción será mucho mayor. Midiendo ese error y estableciendo un umbral son capaces de detectar anomalías en la red: un posible ciberataque. Esta misma lógica se puede aplicar al problema que nos ocupa; entrenamos un AE de forma que se minimice el error de reconstrucción en imágenes vacías, obtenemos el error de reconstrucción de las imágenes de *test* y, en base a un umbral predefinido, decidimos si hay animales (anomalías) o no.

En la literatura aparecen numerosos tipos de AE diferentes. En la sección 2.3 se abordaron algunos de ellos y en [33] se pueden estudiar más en profundidad. El siguiente paso es escoger uno de entre todos los tipos disponibles.

Un AE básico debería ser capaz de realizar correctamente la tarea de reconstrucción de imágenes, pero la elección de un AE más complejo dará presumiblemente mejores resultados. En el artículo [46] se realiza un análisis sobre la reducción de dimensionalidad que llevan a cabo diversos tipo de AE; en lugar de utilizar datos de gran dimensionalidad como entrada para un clasificador, se introduce una tarea previa de reducción de dimensionalidad haciendo uso de AE. Tal y como se demuestra en el artículo, para la mayoría de los casos los RAE (*Robust Autoencoder* o autoencoder robusto) funcionan mejor que el resto, haciendo ver que las características resultantes en la capa de codificación son de mayor calidad. Si este tipo de AE es capaz de realizar una fase de fusión de características de mejor calidad que el resto para tareas de clasificación, es previsible que funcione bien en la tarea original de un AE, la reconstrucción de entradas en las salidas.

En base a todas las razones expuestas, se ha decidido utilizar un RAE para este TFG. Este tipo concreto de AE quedó definido en la sección 2.3.2.

4.5.2. Planteamiento general

En este apartado se mostrará la estructura del software desarrollado, indicando cada una de las partes que lo forman.

El objetivo general del TFG es desarrollar un modelo DL que permita detectar imágenes de dispositivos de fototrampeo que no contengan animales. Para ello, se trabaja

con la siguiente hipótesis inicial: se puede desarrollar un AE (en particular un RAE) cuyo objetivo sea reconstruir con la mayor precisión posible imágenes vacías. Si lo planteamos desde esta perspectiva, cuando se proporcione una imagen con animales al RAE, su reconstrucción será presumiblemente peor que en caso de que la imagen perteneciera al conjunto de imágenes vacías. Midiendo el error de la reconstrucción, podremos determinar la presencia o no de animales en una imagen.

Para conseguir esto, debemos entrenar al RAE únicamente con el conjunto de imágenes vacías. Sin embargo, dentro del grupo de imágenes vacías aparecen imágenes de muchos tipos distintos, imágenes diurnas, nocturnas, brillantes, más oscuras, etc. Por esta razón, se plantea utilizar un conjunto de RAE especializados cada uno en grupos de imágenes con características similares.

El primer paso por tanto es entrenar un modelo de *clustering* que nos permita agrupar imágenes en varios grupos en base a sus características internas, como iluminación, colores u otros aspectos. Este modelo se entrenará haciendo uso de todas las imágenes vacías, utilizando la información del histograma de cada una de ellas.

Una vez formado el modelo de *clustering*, se procederá a la separación en grupos del conjunto de entrenamiento. El modelo habrá sido entrenado para formar N *clusters*, siendo N un número natural, por lo que se acabará separando el conjunto de entrenamiento en N grupos.

Siguiendo el esquema, el siguiente paso es el entrenamiento de los AE haciendo uso exclusivamente de las imágenes vacías pertenecientes al conjunto de entrenamiento. Cada uno de ellos estará especializado en uno de los grupos formados, usando las imágenes de dicho grupo como muestras para el entrenamiento. Estos AE proporcionarán las reconstrucciones de las imágenes originales. Tanto estas reconstrucciones como las imágenes originales serán transferidas al siguiente elemento, el sistema para calcular el error entre imágenes originales y su reconstrucción.

El elemento restante es el clasificador de imágenes. Se encargará de, dado un error obtenido a partir de la diferencia entre la imagen original y la reconstrucción del AE, clasificar la imagen correspondiente en uno de los dos grupos, con animales o vacío. Este modelo será entrenado tanto con imágenes vacías como con imágenes de animales pertenecientes al conjunto de entrenamiento.

Teniendo ya todos los elementos entrenados, podemos pasar al bloque de clasificación de nuevas imágenes. Tomando el modelo de *clustering*, cada nueva imagen se asociará a uno de los N grupos existentes. El AE correspondiente al grupo seleccionado será el encargado de obtener su reconstrucción. Posteriormente, se calculará el

error entre la imagen original y la reconstrucción proporcionada por el AE. Dicho error pasará como entrada al modelo clasificador, que nos dará la clasificación final entre imagen vacía o imagen con animales.

La representación global del sistema se muestra en la figura 4.7

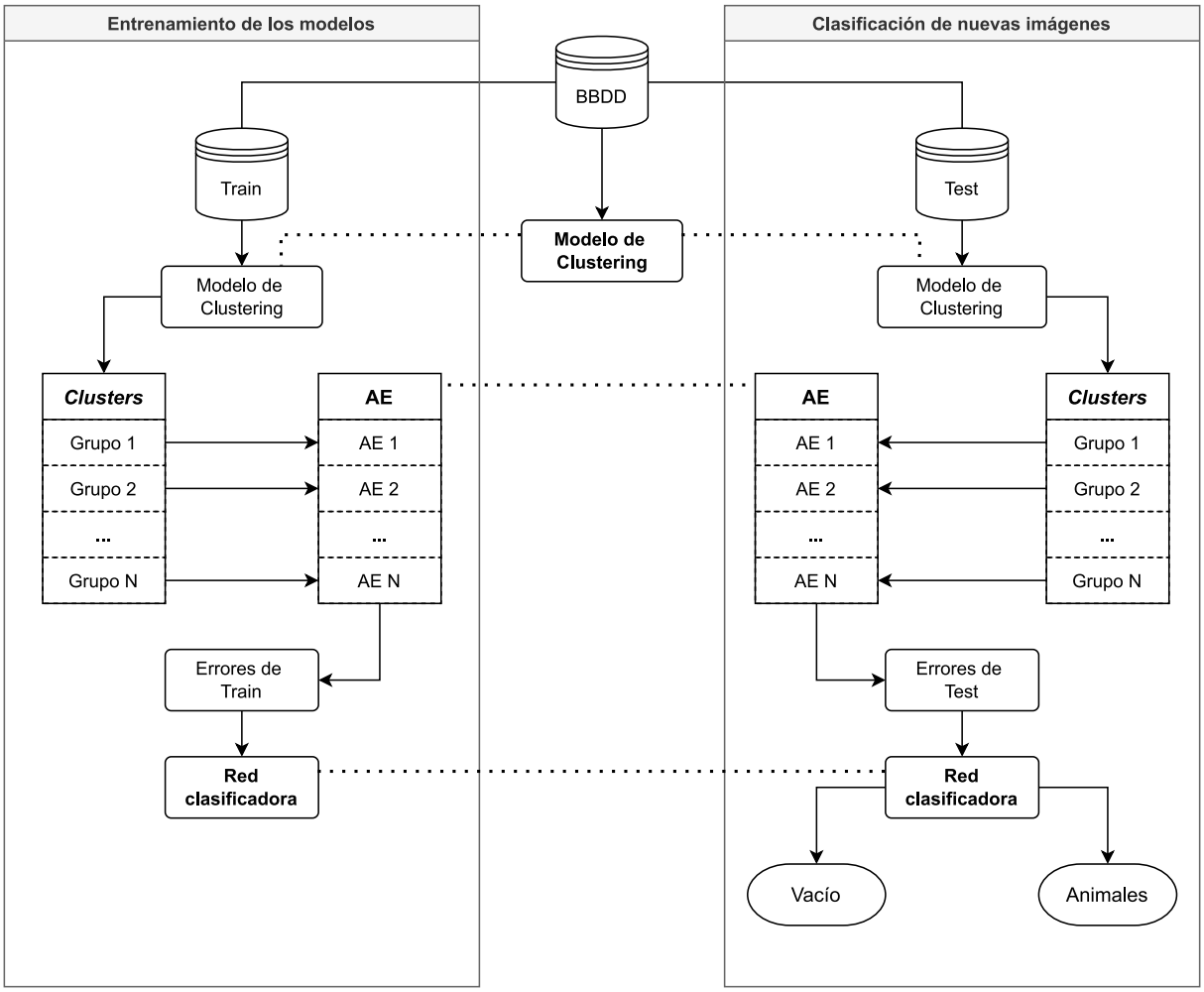


Figura 4.7: Esquema del sistema desarrollado.
En primer lugar, haciendo uso del conjunto completo de imágenes vacías, se entrenará un modelo de *clustering* que permitirá la formación de N *clusters* o grupos de imágenes. Seguidamente, la base de datos se divide en dos particiones: partición de entrenamiento y partición de *test*, sobre las cuales se empleará el modelo de *clustering* para formar los diferentes grupos de imágenes. Cada *cluster* del conjunto de entrenamiento tendrá asociado un AE entrenado con sus propias imágenes. Además, se aprovecharán tanto imágenes vacías como imágenes con animales del conjunto de entrenamiento para entrenar el modelo de red clasificadora, que toma como entrada los errores de reconstrucción.
Por otro lado, cada imagen del conjunto de *test* se pasará por el modelo de *clustering* entrenado previamente asociándola a uno de los AE, que se encargará de obtener el índice de error entre la imagen original y su reconstrucción. La red clasificadora tomará como entrada dicho error para determinar si hay presencia animal o no.

En las siguientes secciones se detallará cómo se ha creado cada uno de estos elementos que componen el sistema completo.

4.6. Implementación y entrenamiento del modelo de *clustering*

En esta sección se detallará todo el proceso de creación del modelo de *clustering*, encargado de separar las imágenes de la BBDD en varios grupos.

4.6.1. Histograma de una imagen

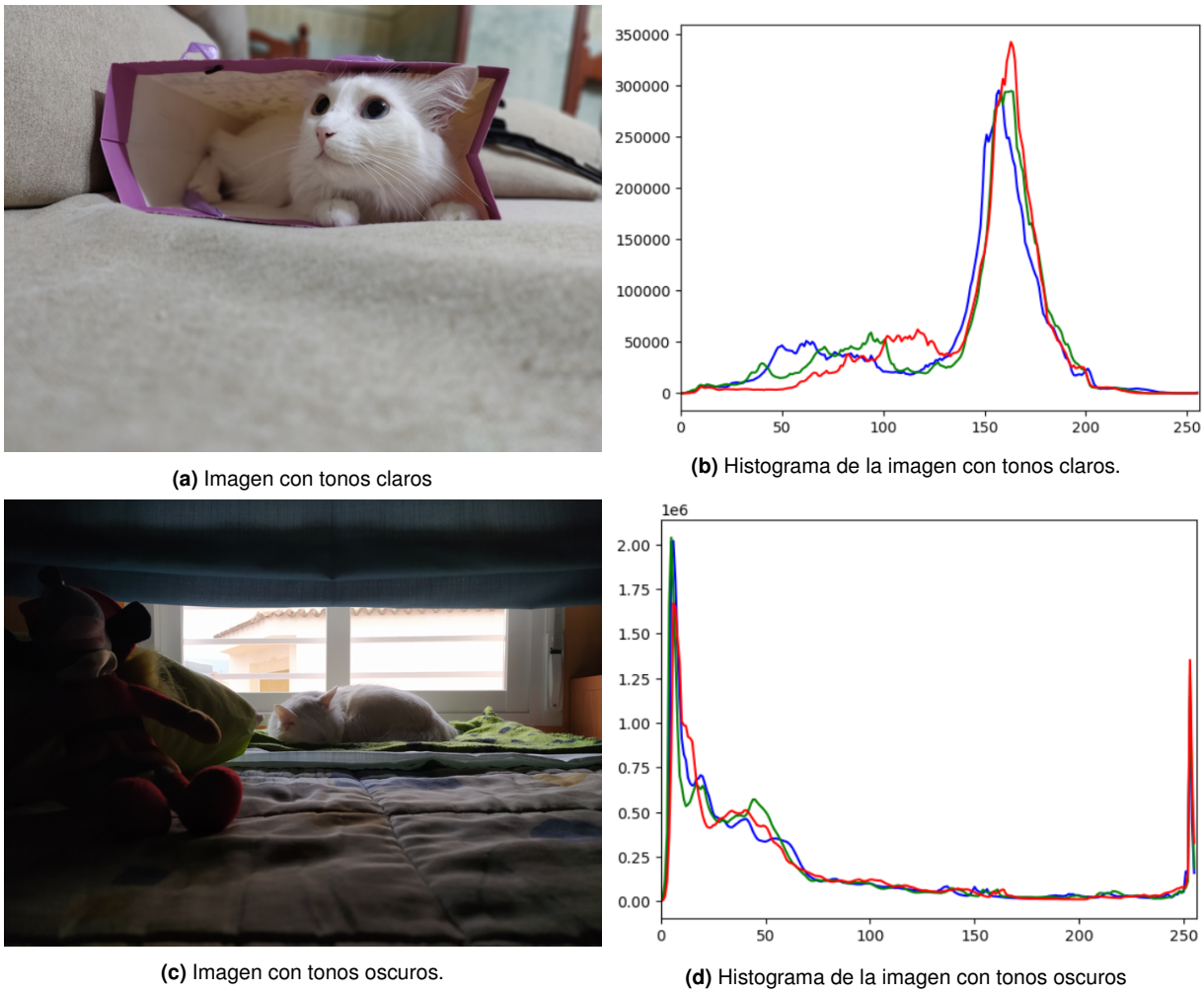
Para comprender cómo se implementa el modelo de *clustering* de imágenes, hay que definir previamente el concepto de histograma de una imagen.

El histograma de una imagen no es más que una representación del número de píxeles en una imagen en función de su intensidad. Los histogramas están formados por cubetas, cada una de ellas representando un cierto rango de intensidad de valores. La formación del histograma se completa analizando todos los píxeles de la imagen y asignando cada uno de ellos a una cubeta en función de su intensidad.

Para comprender mejor su funcionamiento, se analizará el histograma de dos imágenes RGB. Tanto las imágenes como su histograma resultante se pueden analizar en la figura 4.8.

La representación gráfica de los histogramas es sencilla. El eje X representa el número de cubetas que, al tener un tamaño 1, coincide con el rango de valores de píxeles. El eje Y representa el número de píxeles, por lo que una coordenada (x, y) representará el número de píxeles, y, que tiene un determinado valor, x. Cada línea representa uno de los canales de color, *Red* (rojo), *Green* (verde) o *Blue* (azul).

A simple vista se aprecia que la imagen del nivel superior posee tonos mucho más claros frente a la imagen del nivel inferior, en la que predominan tonos más oscuros. Esto se ve claramente reflejado en los histogramas resultantes. En la imagen con tonos oscuros, la mayoría de valores de píxeles se agrupan en la zona derecha del histograma, indicando tonos oscuros cercanos al negro. Ocurre todo lo contrario para la imagen con tonos claros, donde los valores predominantes se sitúan en la mitad derecha del histograma, indicando colores con una gran tonalidad, alejados de los tonos oscuros. El hecho de que los tres canales de color tengan unos valores similares indican una presencia alta del gris.



(a) Imagen con tonos claros

(b) Histograma de la imagen con tonos claros.

(c) Imagen con tonos oscuros.

(d) Histograma de la imagen con tonos oscuros

Figura 4.8: Histogramas de dos imágenes. A la izquierda se muestran las imágenes. A la derecha los histogramas asociados a cada una de ellas. En (b) se aprecia cómo el histograma refleja muchos valores de píxeles cercanos a los valores 150-200, indicando tonos claros. Por el contrario, en (d) comprobamos cómo los valores se agrupan principalmente en la zona izquierda, lo que se traduce en una imagen con menor iluminación y tonos oscuros.

4.6.2. Elección e implementación del modelo de *clustering*

Con el objetivo definido en la sección anterior, se procederá a desarrollar un algoritmo de *clustering* sobre los histogramas.

Previo a la división en grupos de todos los histogramas disponibles, se ha decidido escoger un subconjunto aleatorio de 578 imágenes vacías sobre el que aplicar el *clustering*. De esta forma podremos comprobar de forma más cómoda y rápida si el algoritmo que utilicemos provee unos resultados aceptables o no.

El algoritmo a utilizar será K-Means [47], concretamente la versión de Lloyd [48], un método de agrupamiento que permite la partición de un conjunto de datos de entrada en k grupos, donde cada observación pertenece al grupo más cercano. El algoritmo

original usa la distancia euclídea para medir la distancia entre un valor y cada grupo, aunque existen variaciones en las que se usan otras métricas.

Se han escogido dos implementaciones distintas del algoritmo, la implementación de la librería Scikit-learn [49] y la implementación de la librería NLTK [50]. La principal diferencia entre una y otra radica en la métrica de distancia que se puede utilizar: mientras que en caso de Scikit-learn solo se permite la distancia euclídea, en NLTK se puede especificar cualquier otra métrica.

En los siguientes apartados se expondrán las características, objetivos y resultados de cada experimento.

Parámetros utilizados en cada prueba

Las imágenes escogidas para la realización de los experimentos han sido seleccionadas de forma aleatoria dentro del conjunto de imágenes "Vacío", es decir, en ninguna de ellas aparecen animales. El número de cubetas del histograma usadas para cada dimensión será $1/3$ del rango de valores de dicha dimensión, agrupando así los valores de los píxeles en cubetas de tamaño 3 para la representación del histograma. De esta forma una imagen RGB con rango de píxeles [0,255] pasará a un histograma con un rango de valores [0,85].

Durante la fase de experimentación se analizarán tres aspectos del algoritmo:

1. **Modelo de color**

   Un modelo de color proporciona un modelo matemático abstracto para representar los colores en forma numérica. En nuestro caso se analizará la diferencia entre el modelo RGB (*Red, Green, Blue*) y HSV (*Hue, Saturation, Value*).

   Durante la realización de estos experimentos referentes al modelo de color se utilizará la distancia euclídea como métrica de distancia para el algoritmo. Además, se crearán diez *clusters*.

2. **Métrica de distancia**

   Una vez calculado el modelo de color que se utilizará, pasaremos a realizar experimentos sobre la métrica de distancia. Se analizarán cuatro métricas distintas para calcular la distancia entre dos vectores u y v:

   a) *Euclidean distance* (implementación de Scikit-Learn) (4.1)

$$distance_{u,v} = \sqrt{\sum (u_i - v_i)^2} \tag{4.1}$$

b) *Cosine distance* (4.2)

$$distance_{u,v} = 1 - \frac{u * v}{||u||_2 * ||v||_2} \tag{4.2}$$

c) *Manhattan distance* (4.3)

$$distance_{u,v} = \sum |u_i - v_i| \tag{4.3}$$

d) *Correlation distance* (4.4)

$$distance_{u,v} = 1 - \frac{(u - \overline{u}) * (v - \overline{v})}{||u - \overline{u}||_2 * ||v - \overline{v}||_2} \tag{4.4}$$

El número de *clusters* en estas pruebas seguirá siendo diez, mientras que el modelo de color que se utilizará será el ganador del experimento anterior.

3. **Número de *clusters***

   Tras hallar el modelo de color y métricas adecuadas, el siguiente paso será decidir el número de *clusters* que formará el algoritmo. Se analizarán los cambios al escoger un número de *clusters* entre 1 y 12.

Para cada experimento, se medirá su resultado siguiendo dos criterios:

- La distancia intra-cluster de cada grupo, es decir, la sumatoria de distancias euclídeas de cada elemento del grupo (cada histograma) a su centroide asignado.
- Un análisis visual sobre los grupos que el algoritmo ha creado.

Nota: a partir de este momento, en la memoria nos referiremos a "distancia intra-cluster" como *puntuación*.

Elección del modelo de color

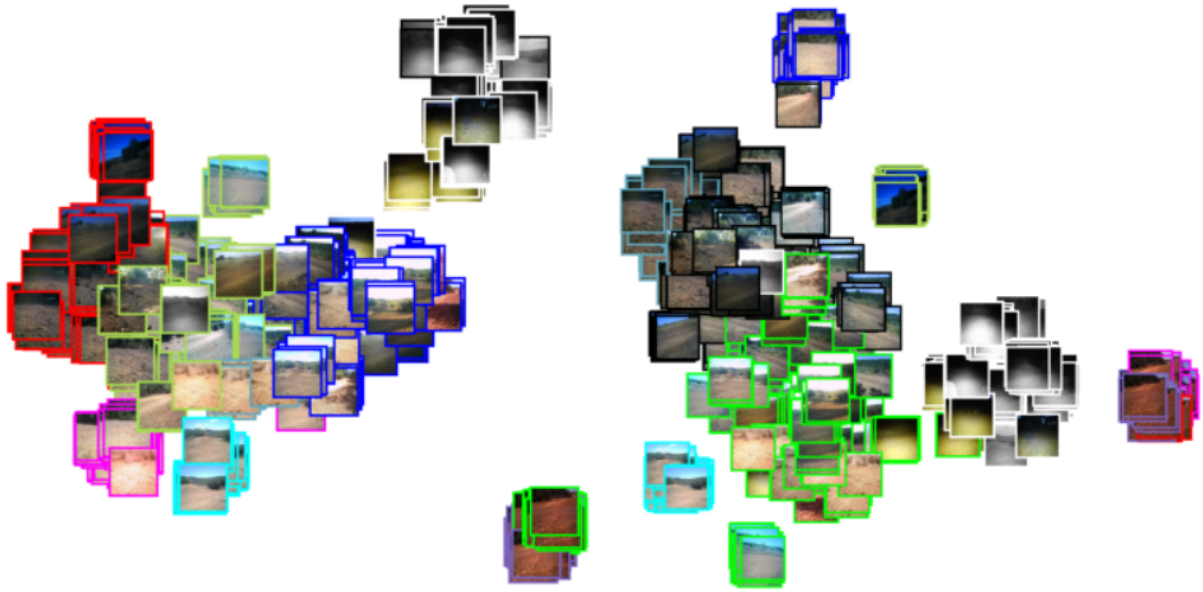
Las puntuaciones obtenidas para cada modelo de color se ven reflejadas en la tabla 4.3.

Para el modelo RGB, se ha obtenido una puntuación de 770.05, mientras que en el caso de HSV la puntuación es de 1157.47.

Modelo de color	Puntuación
RGB	770.05
HSV	1 157.47

Tabla. 4.3: Resultados de los experimentos respecto al modelo de color.

En la figura 4.9 se pueden observar los cambios en el agrupamiento al usar un modelo de color u otro.



(a) Modelo RGB **(b)** Modelo HSV

Figura 4.9: Resultado del *clustering* usando el modelo RGB y HSV. Cada grupo está representado por marcos de distintos colores

A simple vista, podemos observar cómo en cada experimento las imágenes se han agrupado de forma similar, dejando las imágenes nocturnas en un *cluster* apartado del resto. Sin embargo podemos ver algunas diferencias. Por ejemplo, en el modelo HSV las imágenes con tonos rojizos las ha dividido en cuatro grupos, tres de ellos centrados exclusivamente en imágenes con esos tonos (esquina inferior derecha) y otro en el que se agrupan imágenes de otros tipos (parte inferior, con recuadro verde). Esto provoca que el resto de grupos deba ser más grande.

Fijándonos en la puntuación podemos ver cómo la distancia intra-cluster del experimento RGB es notablemente menor que la encontrada en el modelo HSV, indicando una mejor calidad de agrupamiento.

Por último, es destacable que al utilizar el modelo HSV de las imágenes hay que hacer una conversión de las mismas ya que inicialmente se leen en formato RGB. Esto provoca un mayor coste computacional y por tanto un mayor tiempo de ejecución.

En base a estas razones, se decide escoger el modelo RGB como modelo de color.

Elección de la métrica de distancia

En la tabla 4.4 se aprecian los resultados obtenidos para cada métrica de distancia.

Métrica de distancia	Puntuación
Euclidean	770.05
Canberra	820.86
Cosine	856.57
Manhattan	902.32

Tabla. 4.4: Resultados de los experimentos respecto a la métrica de distancia.

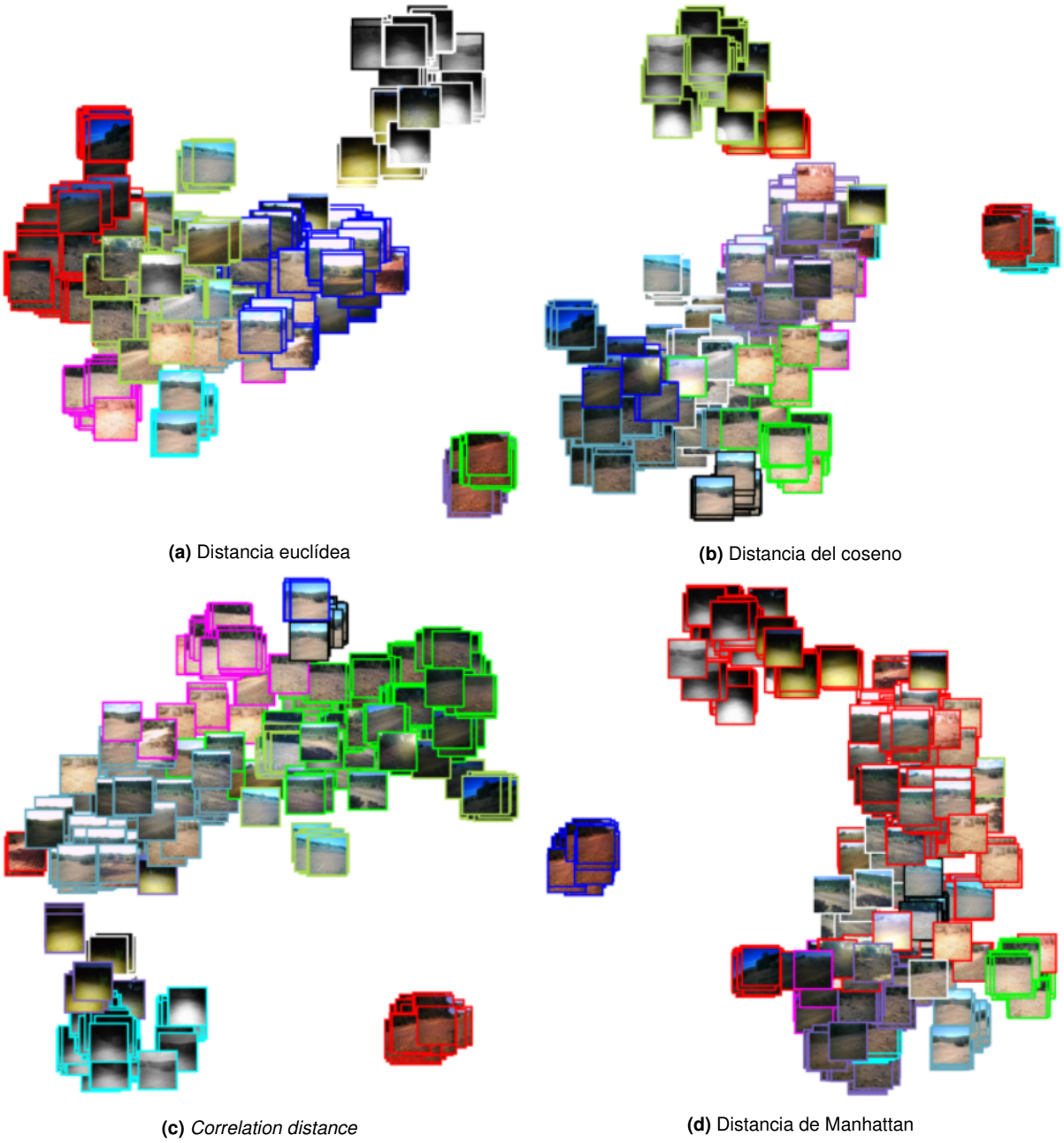
Visualmente podemos apreciar la diferencia entre escoger una métrica u otra en la figura 4.10.

En el caso de utilizar la métrica distancia de Manhattan podemos ver una calidad del agrupamiento notablemente peor que el resto. Encontramos un grupo muy amplio con imágenes de muchos tipos, incluyendo imágenes diurnas y nocturnas (grupo rojo). Igualmente, en la puntuación obtenida se refleja como el peor resultado de los cuatro. Esta métrica de distancia queda descartada.

A nivel visual, no hay una diferencia excesiva en el resto de métricas. Sí se aprecian diferencias más leves, como dividir las imágenes con tonos rojizos en dos grupos (distancia euclídeas y del coseno) o mantenerlas en un único grupo (*correlation distance*). Centrándonos en las imágenes nocturnas, todas las métricas consiguen separarlas del resto, incluso subdividiéndolas en grupos más pequeños. El tamaño medio de los grupos también es muy similar, todos tienen un tamaño parecido.

Fijándonos en la distancia intra-cluster, podemos ver que efectivamente la diferencia no es muy amplia (menos de 80 puntos de diferencia entre la mejor y la peor).

En base a estos resultados, decidimos que la mejor opción es utilizar la distancia euclídea como métrica de distancia, aunque la distancia del coseno y *correlation distance* también habrían sido una buena elección.



(a) Distancia euclídea

(b) Distancia del coseno

(c) *Correlation distance*

(d) Distancia de Manhattan

Figura 4.10: Resultado del *clustering* usando diferentes métricas de distancia. Cada grupo está representado por marcos de distintos colores

Elección del número de *clusters*

Los resultados de la puntuación respecto al número de *clusters* se muestra en la gráfica 4.11.

En la imagen podemos ver una tendencia logarítmica. Principalmente llaman la atención dos puntos. En primer lugar, al formar de dos a cuatro grupos hay una tendencia más pronunciada que en el resto, que se relaja hasta llegar a los seis grupos. Sin embargo hay un cambio brusco de tendencia en el cambio de seis a siete grupos.

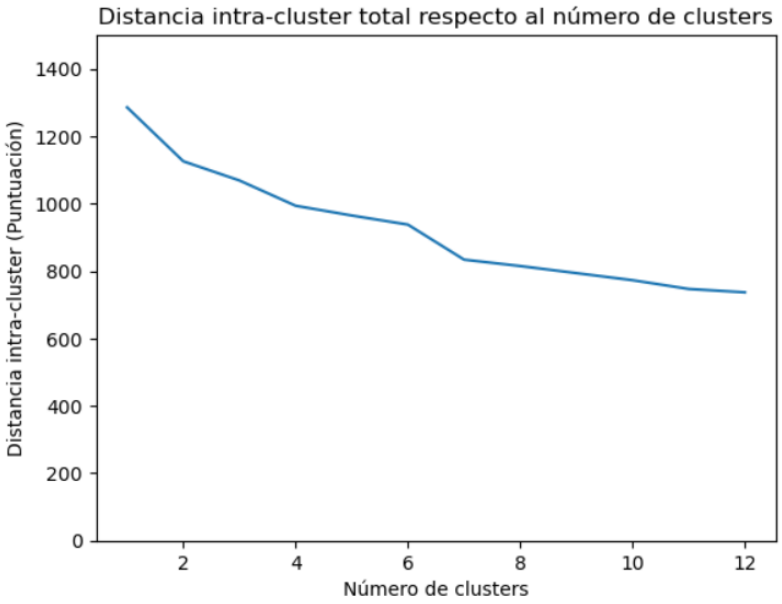


Figura 4.11: Puntuación del *clustering* respecto al número de *clusters* usados

Analizando visualmente los *clusters* formados, este cambio se debe a la separación en grupos distintos de imágenes nocturnas y diurnas. De formar siete grupos en adelante, el cambio es mucho más suave.

En base a los cambios observados, se ha decidido establecer en siete el número de *clusters* a formar.

Resumen del modelo final para el *clustering* de histogramas

Tomando en cuenta todas las consideraciones mencionadas en los apartados anteriores, el modelo de *clustering* se aplicará siguiendo los siguientes aspectos:

1. El algoritmo será K-Means de la implementación de la librería Scikit-learn.
2. El modelo de color de las imágenes será RGB.
3. La métrica de distancia será la distancia euclídea.
4. El número de *clusters* a formar serán 7.

Este modelo de *clustering* será el que utilizaremos para separar en grupos tanto las imágenes de entrenamiento como las imágenes de *test*. Previamente, será entrenado con el conjunto de imágenes vacías.

4.7. Separación del conjunto de entrenamiento y test. *Clustering* sobre todo el conjunto de imágenes.

El siguiente paso es separar el conjunto de datos disponible en tres particiones, entrenamiento, validación y *test*, tal y como se desarrolló en la sección 2.1.2.

La proporción escogida será 60-20-20, es decir, 60 % de imágenes para entrenamiento, 20 % para *test* y 20 % para validación (valores aproximados). Para formar el conjunto de *test* se han escogido 16 000 imágenes aleatorias de la BBDD, formadas a partir de 8 000 imágenes pertenecientes a cada uno de los subconjuntos, Vacío y Animales. A partir de este momento, para el entrenamiento de los modelos se trabajará únicamente con el resto de imágenes vacías, las cuales forman el conjunto de entrenamiento.

Después de hacer esta separación y teniendo ya el modelo de *clustering* definido, pasamos a aplicarlo sobre el conjunto completo de imágenes. El resultado es la formación de 7 grupos de imágenes similares entre sí.

A modo de ejemplo y para tener una idea del tamaño de cada grupo frente al resto, en la tabla 4.5 se muestra el número de imágenes asociadas a cada grupo para el conjunto de entrenamiento (imágenes vacías).

Grupo	Nº Imágenes
0	6 450
1	7 683
2	524
3	13 863
4	218
5	446
6	319

Tabla. 4.5: Grupos formados tras el *clustering* de imágenes vacías sobre el conjunto de entrenamiento y validación.

En este punto queda por crear el conjunto de validación. Estará formado por 8 000 imágenes escogidas aleatoriamente del ya creado conjunto de entrenamiento.

4.8. Implementación y entrenamiento del autoencoder

En esta sección se detallará todo el proceso de creación y entrenamiento del modelo RAE, mostrando la evolución desde la primera aproximación hasta la versión definitiva.

La función de pérdida para el RAE será la especificada en el paquete RUTA [51] en la sección referente a *robust AE*. Dicha función es la misma ya definida en la sección 2.3.2. La única diferencia recae en que se utilizará el negativo de la función. De esta forma durante el entrenamiento se buscará minimizar el resultado.

$$Loss = -\sum kernel(y_{pred} - y_{true}) \tag{4.5}$$

$$kernel(\alpha) = \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{\alpha^2}{2\sigma^2}) \tag{4.6}$$

En cuanto al conjunto de datos que alimentará la red neuronal, teniendo en mente que el objetivo es optimizar el AE para la reconstrucción de imágenes vacías, se emplearán únicamente imágenes de este tipo, tanto para el entrenamiento (conjunto de entrenamiento y validación) como para la evaluación del modelo (conjunto de *test*). A pesar de esto, se harán pruebas introduciendo imágenes con animales para verificar el comportamiento correcto del AE. El número de imágenes de cada conjunto que se utilizará se especifica en la tabla 4.6.

Conjunto	Nº de imágenes
Entrenamiento	21 503
Validación	8 000
Test	8 000

Tabla. 4.6: Número de imágenes pertenecientes al conjunto de entrenamiento, validación y test.

4.8.1. Elección de la arquitectura y parámetros de la red

En este apartado se realizarán los experimentos necesarios para optimizar la arquitectura del AE. Se toma como base la arquitectura del modelo de AE empleado en el TFG de Jesús Enrique [41]. La arquitectura base se ha adaptado para poder mane-

jar imágenes de dimensiones 384x256 píxeles. Además, se ha pasado de un modelo AE básico a un modelo RAE, introduciendo la función de pérdida definida en 2.17.

Arquitectura y parámetros iniciales

A continuación se muestran los parámetros iniciales del AE junto a una breve descripción de cada uno [3].

- **Número de *epochs***: 20

  El número de *epochs* o épocas define el número de veces que el modelo recorrerá el dataset proporcionado, dividiendo el entrenamiento en distintas fases y haciendo más fácil la evaluación periódica del modelo.

- ***Batch size***: 32

  El *batch size* o tamaño de lote es el número de elementos del *dataset* que se procesarán en paralelo. Durante el entrenamiento, los resultados obtenidos al procesar un conjunto de elementos de tamaño especificado en este parámetro resultará en una única actualización del modelo.

- **Optimizador**: Adam

  El optimizador es el elemento encargado de modificar los parámetros del modelo en base al resultado generado en la función de pérdida.

- **Ratio de aprendizaje**: $0{,}001$ (por defecto)

  La ratio de aprendizaje es un parámetro ligado al optimizador. Define cuán rápido se adaptará el modelo al problema, indicando al optimizador el grado de actuación sobre el modelo.

- ***Steps per epoch*** y ***validation steps***: $\frac{1000}{batch\ size}$

  El parámetro *steps per epoch* se refiere al número de iteraciones para que un *epoch* se considere completado, procesando en cada iteración el número de imágenes dado por el parámetro *batch size*. Por su lado, el parámetro *validation steps* cumple la misma función sobre el conjunto de validación, en lugar del conjunto de entrenamiento.

  Por defecto, estos valores quedan establecidos como el número de datos o instancias entre el *batch size* establecido. Sin embargo, en esta primera versión se

[3] Los detalles sobre cada parámetro se pueden encontrar en la propia documentación de Keras https://keras.io/

ha decidido establecer estos parámetros con un valor predefinido, haciendo que en cada *epoch* se procese una parte del *dataset*, no el conjunto completo. Presumiblemente, el sistema será más rápido ya que en cada *epoch* se procesa un menor número de imágenes. Sin embargo, esto puede traducirse en una pérdida de calidad de reconstrucción.

Todos estos parámetros mencionados se analizarán y modificarán en las diferentes versiones que se presentarán en este mismo apartado, optimizando la arquitectura de la red.

La arquitectura básica que se tomará como base se observa en la figura 4.12 [4]. En el extremo izquierdo se aprecia la primera capa de la red, la capa de entrada (*InputLayer*), preparada para admitir imágenes con dimensiones $384x256$. Seguida de ella, aparecen una serie de capas convolucionales (*Conv2D*) y capas de *max-pooling* (*MaxPooling2D*), que se encargarán de procesar y detectar las características esenciales de la imagen y reducir su tamaño, respectivamente. Tras esta fase de codificación, aparece una capa de neuronas densa (*Dense*) representando la entrada en su estado codificado. Seguidamente volvemos a tener una serie de capas convolucionales traspuestas (*Conv2DTranspose*) y capas *up-sampling* (*UpSampling2D*) que llevarán a la capa final, en la que se representará la predicción del AE.

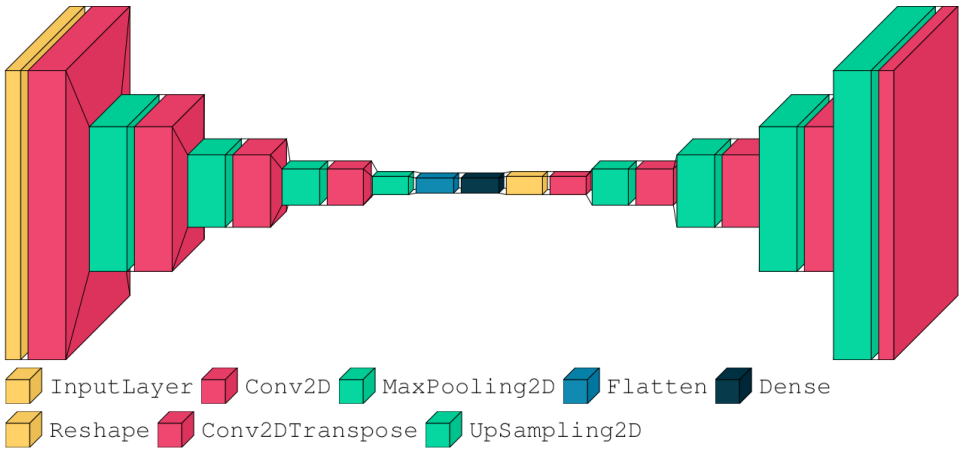


Figura 4.12: Arquitectura original del AE.

Versión 1

En esta primera versión se aplicará la estructura básica mostrada en el apartado anterior. Se trabajará únicamente con las imágenes pertenecientes al *cluster* 0.

[4] La visualización de los modelos se ha conseguido haciendo uso de la librería *VisualKeras* [52]

La evolución de la función de pérdida durante el entrenamiento de la red neuronal se puede observar en la gráfica 4.13. En ella se aprecia cómo a partir de los 7 *epochs* la función se estabiliza a un valor casi constante hasta los 20 *epochs*.

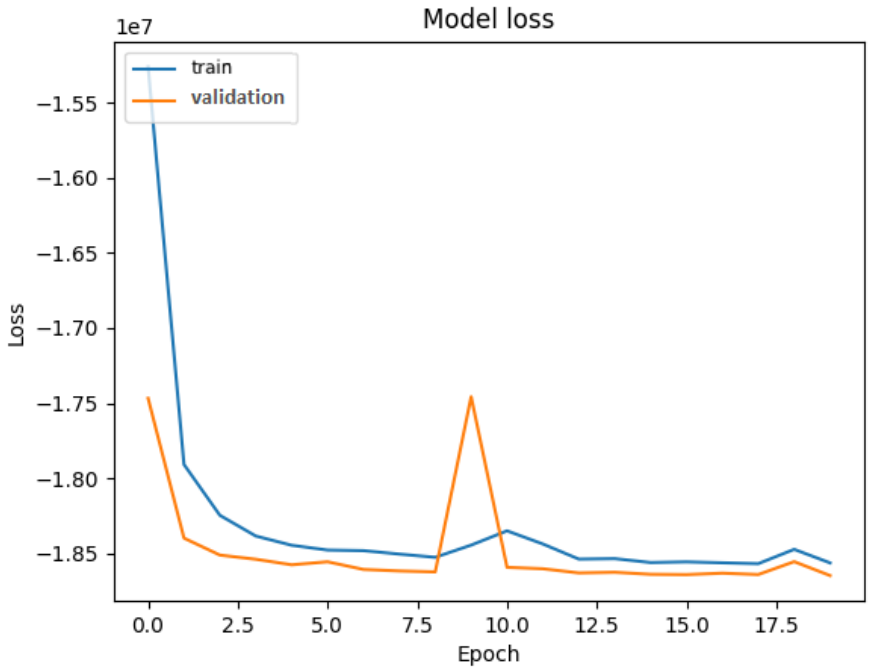


Figura 4.13: Evolución de la función de pérdida en la versión 1 del AE.

En la figura 4.14 se muestra la comparación entre una imagen de ejemplo original y la reconstrucción hecha por el AE. A simple vista podemos ver cómo la reconstrucción es notablemente borrosa. Aunque se pueden distinguir con facilidad las diferentes partes de la imagen como el cielo, la hierba alta o la zona de tierra, los resultados no son tan nítidos como se esperaba. En las siguientes versiones del modelo se intentará mejorar los resultados.

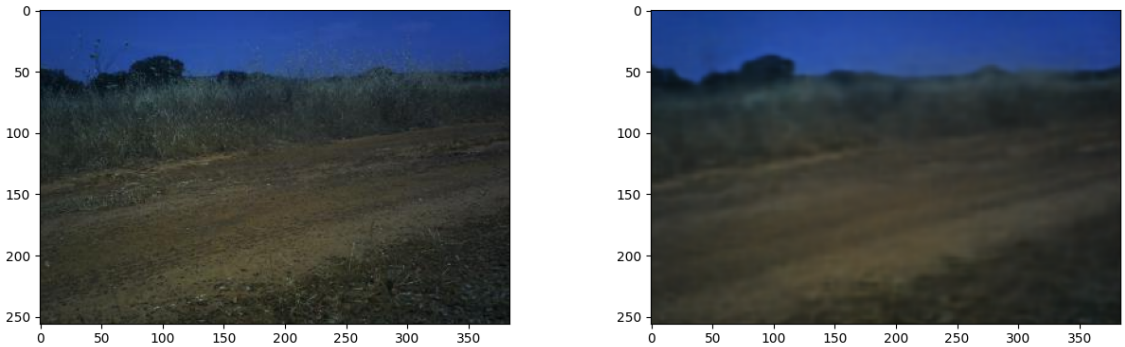


Figura 4.14: Comparación entre la imagen original (izquierda) y la reconstrucción del AE en la versión 1 (derecha).

Versión 2

Después de estudiar los resultados de la primera versión, lo primero que consideramos fue aumentar el tamaño de la red neuronal. Estos se puede conseguir de dos formas distintas: aumentando el número de neuronas de cada capa o aumentando el número de capas.

Para esta segunda versión se optó por añadir una capa más a la red, haciéndola aún más profunda. Dicha capa se añadirá justo antes de la zona central, donde se encuentra la codificación de la entrada. Como buscamos que el AE tenga una estructura simétrica, añadimos también otra capa en el lado de decodificación. De esta forma, la arquitectura queda tal y como se aprecia en la figura 4.15.

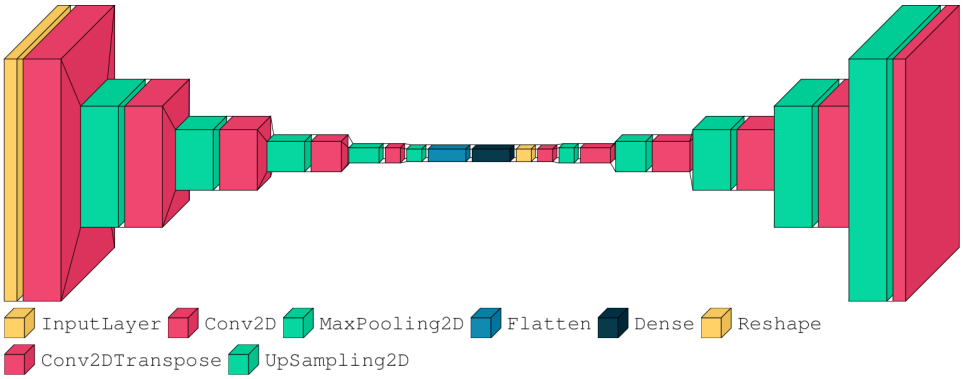


Figura 4.15: Arquitectura original de la versión 2 del AE.

En la gráfica 4.16 se puede ver la evolución de la función de pérdida durante el entrenamiento del modelo. El comportamiento es muy similar a la gráfica de la función de pérdida vista en la primera versión, ya que podemos observar una clara estabilización a partir de los 7 *epochs*. Sin embargo, se aprecia que los valores alcanzados son ligeramente más altos, indicando un peor resultado.

En 4.17 se muestra un ejemplo de reconstrucción de una imagen.

En comparación con la versión anterior, los resultados parecen a simple vista mucho peores, algo que puede parecer ilógico ya que hemos aumentado la capacidad de la red neuronal.

Una de las posibles causas de la mala calidad de la reconstrucción de las imágenes puede estar en el tamaño de la capa de codificación resultante. Al analizar con detalle la arquitectura de la red, podemos ver cómo la capa de codificación posee 2 304 neuronas. En comparación con el tamaño de las imágenes que la red toma como entrada ($384 * 256 = 98304$ valores), el tamaño de la capa de codificación resulta casi insignificante. Al utilizar un AE, conviene que la capa de codificación sea lo suficientemente

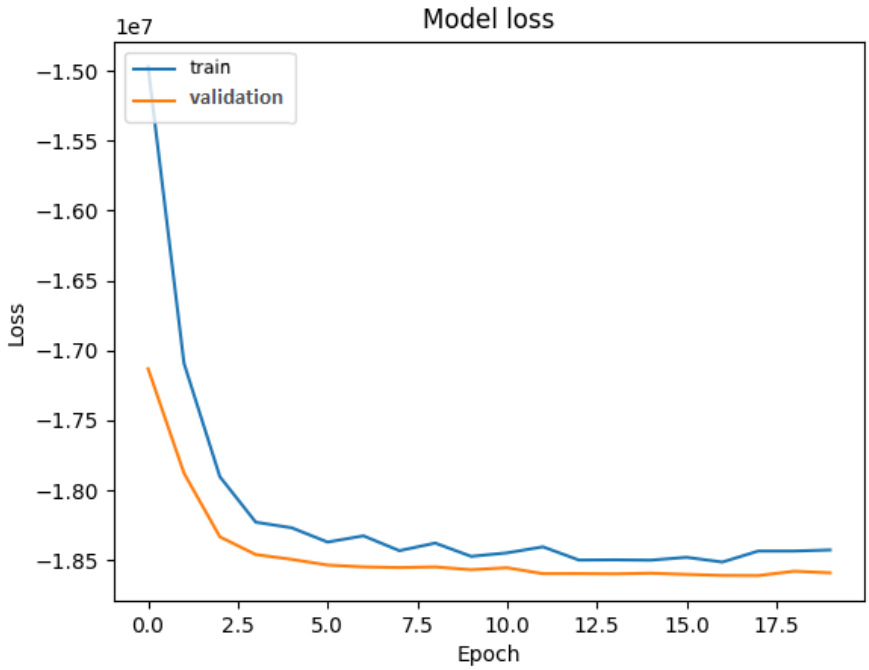


Figura 4.16: Evolución de la función de pérdida en la versión 2 del AE.

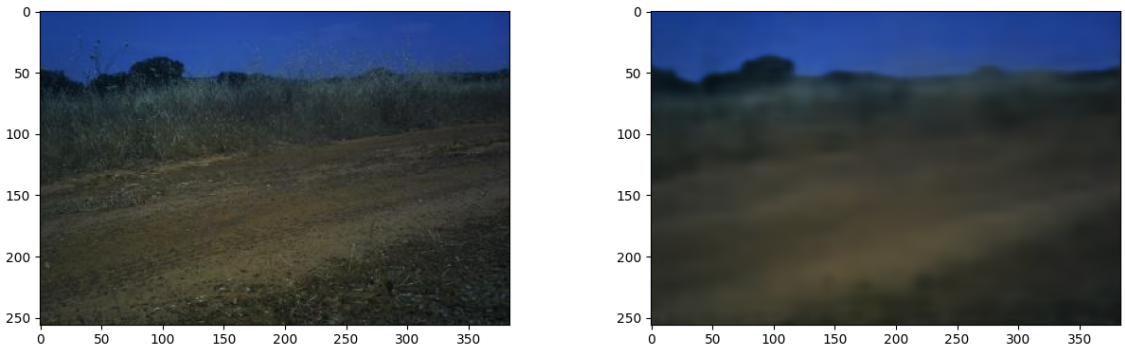


Figura 4.17: Comparación entre la imagen original (izquierda) y la reconstrucción del AE en la versión 2 (derecha).

pequeña como para captar aquellos aspectos más relevantes de la imagen siempre y cuando no se pierdan detalles importantes, algo que ocurre si empleamos una capa de codificación muy pequeña, como parece ser el caso de esta versión.

Después de estudiar estos resultados, decidimos desechar estos cambios, volviendo a la arquitectura original.

Versión 3

Hasta ahora se han utilizado únicamente imágenes pertenecientes al *cluster* 0, siendo todas ellas muy parecidas a las mostradas en las imágenes de ejemplo. Sin embargo, para poder escoger los parámetros adecuados del AE que funcionen bien para todos los *clusters*, parece conveniente escoger un subconjunto de fotografías pertenecientes a todos los *clusters*.

Con este objetivo en mente, se ha decidido escoger un subconjunto representativo de los datos de entrenamiento, validación para la optimización de parámetros del AE formado por el 20 % de las imágenes de cada *cluster*. De la misma forma, se utilizará el 20 % de imágenes del conjunto de test para evaluar el modelo en esta fase y comprobar así si su comportamiento ante nuevas imágenes es correcto. El número concreto de imágenes de cada conjunto se puede analizar en la tabla 4.7.

Conjunto	Nº de imágenes
Entrenamiento	4 297
Validación	1 598
Test	1 598

Tabla. 4.7: Número de imágenes pertenecientes al conjunto de entrenamiento y validación durante la fase de optimización de parámetros del AE. Se incluye también el número de imágenes de *test* que se usarán para evaluar el modelo en esta fase.

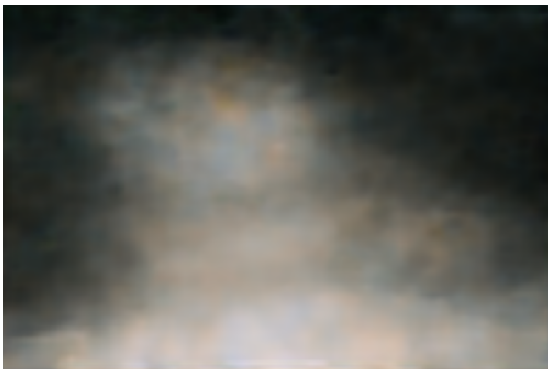
Por otro lado, se han reservado 10 imágenes para el análisis visual del AE. Las imágenes 0, 1, 2 y 8 contienen animales en su interior, el resto son imágenes vacías. Tres de estas imágenes se muestran en la figura 4.18. Sobre estas imágenes se calculará el error cuadrático medio entre las originales y sus respectivas reconstrucciones, definido anteriormente en la ecuación 2.14. De esta forma no solo confiamos únicamente en el análisis visual a la hora de elegir la mejor arquitectura.

Figura 4.18: Imágenes para analizar visualmente la reconstrucción del AE. Se incluyen dos imágenes con animales (extremo derecho e izquierdo) y una vacía (centro)

Tras generar este nuevo conjunto de datos, se procede a comprobar los resultados utilizando una arquitectura de red idéntica a la mostrada en la versión 1. En la figu-

ra 4.19 se presentan 3 imágenes y su reconstrucción haciendo uso de este modelo, mientras que en la tabla 4.8 se muestra el MSE obtenido al comparar las imágenes originales y reconstruidas por el AE.

(a) Original 1



(b) Reconstrucción 1

(c) Original 2

(d) Reconstrucción 2

(e) Original 3

(f) Reconstrucción 3

Figura 4.19: Comparación entre las imágenes originales (izquierda) y la reconstrucción del AE en la versión 3 (derecha).

Como era de esperar, al no haber cambiado la arquitectura del modelo los resultados siguen siendo borrosos, algo que se puede comprobar en las tres imágenes que se incluyen a modo de ejemplo. Además, al incluir imágenes de todos los *clusters* (a diferencia de las primeras dos versiones), el AE presenta resultados algo peores que en la versión 1. La gráfica 4.20 demuestra este hecho.

Sin embargo, podemos analizar cómo se comporta el AE ante diferentes tipos de imágenes. Centrándonos en las reconstrucciones podemos ver una clara diferencia

Imagen	MSE
Imagen 0	2 829.86
Imagen 1	5 408.69
Imagen 2	1 139.09
Imagen 3	172.58
Imagen 4	1 021.96
Imagen 5	169.42
Imagen 6	465.80
Imagen 7	1 358.60
Imagen 8	2 061.53
Imagen 9	2 753.83
Media (vacío)	990.37
Media (animales)	2 575.02
Diferencia	1 584.65

Tabla. 4.8: Error cuadrático medio para las imágenes de prueba de la versión 3.

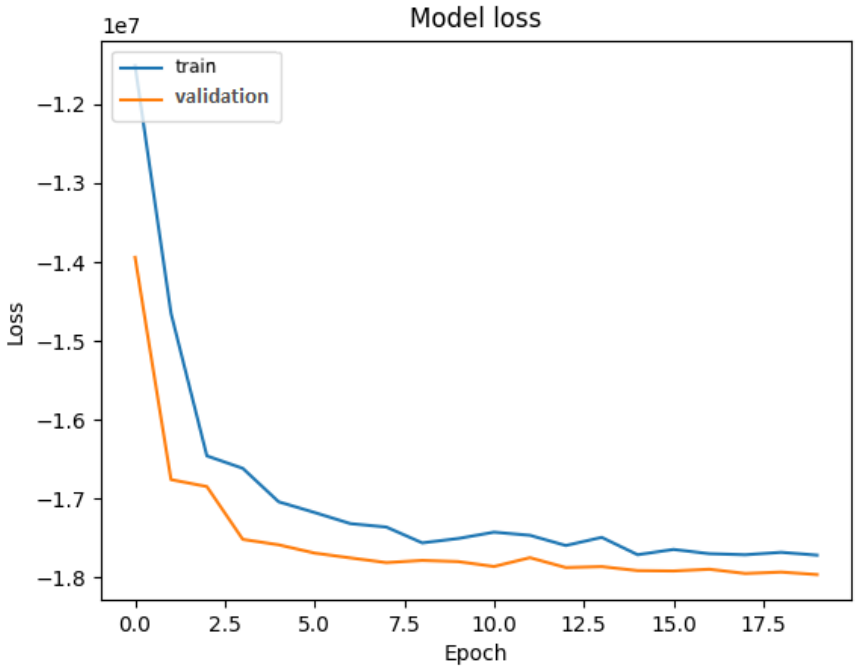


Figura 4.20: Evolución de la función de pérdida en la versión 3 del AE.

entre aquellas imágenes en las que aparecen animales y en las que no aparecen. Mientras que en la segunda imagen la reconstrucción es completa, en las imágenes que contienen animales la reconstrucción es mucho peor, reconstruyendo mal toda

la imagen (imagen 1) o, al menos, la zona donde está el animal (imagen 3). Este es justo el comportamiento que buscamos en el AE: reconstrucción correcta en imágenes vacías e incorrecta en imágenes con animales, lo que resulta en un mayor valor MSE.

Esta versión ha demostrado que el AE se comporta correctamente ante las imágenes que encuentra. Sin embargo permanece el problema de la mala calidad general de reconstrucción, algo que abordaremos en la siguiente versión.

Versión 4

Tras aumentar el número de capas de la red y ver cómo su funcionamiento empeoraba (versión 2), hemos decidido no tocar la propia arquitectura de la red y centrarnos en otros parámetros, en este caso el número de *epochs*.

En las versiones anteriores se ha mantenido un número de 20 *epochs* constante. Teniendo en cuenta el tamaño y cantidad de imágenes que tenemos, aumentar este valor podría dar como resultado una mejora en la calidad de reconstrucción del AE.

Para analizar este hecho, se han realizado pruebas utilizando 200, 150 y 100 epochs.

En la figura 4.21 se puede observar la evolución de la función de pérdida después de 200 *epochs* junto a la evolución de la métrica MSE durante el entrenamiento del modelo. Se incluye también la tabla 4.9, que registra el MSE de las 10 imágenes de prueba [5]. Por último, se muestra una imagen de ejemplo con su posterior reconstrucción tomando 20, 100, 150 y 200 *epochs* en la imagen 4.22.

Tomando como referencia las gráficas de la pérdida del modelo y el valor MSE durante el entrenamiento, podemos ver cómo a partir de las iteraciones 50 - 100 ambos valores siguen disminuyendo de una forma mucho más suave. Por otro lado, centrándonos en el valor MSE de las reconstrucciones para las imágenes de prueba se aprecia una diferencia considerable entre 100 y 150 *epochs* donde por un lado se reduce el error de reconstrucción en imágenes vacías y por otro lado se incrementa en imágenes con animales, aumentando por tanto la distancia entre ambos grupos. Esta diferencia alcanza su máximo en 150 *epochs*.

[5] En los resultados presentados se detectan diferencias entre el valor MSE conseguido durante el entrenamiento (valores muy cercanos a 0) y el obtenido al comparar una imagen original y su reconstrucción (valores que varían entre una centena y varios miles). Durante el entrenamiento, las imágenes han sido normalizadas previamente a valores entre 0 y 1, algo que no se ha aplicado al calcular la diferencia entre una imagen original y su reconstrucción. Por esta razón, los valores MSE son tan dispares.

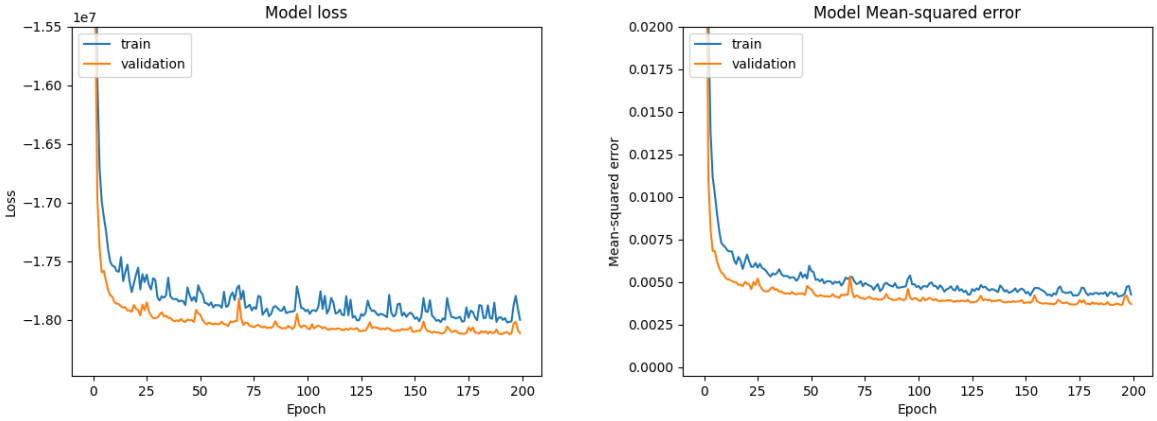


Figura 4.21: Evolución de la función de pérdida y MSE durante el entrenamiento para la versión 4.

Imagen	MSE-200	MSE-150	MSE-100
Imagen 0	2 844.81	3 886.62	3 488.41
Imagen 1	5 395.84	5 667.41	5 213.06
Imagen 2	1 110.42	1 184.70	1 155.86
Imagen 3	133.09	139.09	153.03
Imagen 4	663.49	760.07	815.75
Imagen 5	91.11	99.34	107.07
Imagen 6	332.62	368.11	346.14
Imagen 7	1 164.39	1 202.68	1 167.21
Imagen 8	2 043.66	2 112.62	1 985.03
Imagen 9	2 307.78	2 340.60	2 534.57
Media (vacío)	778.74	818.31	854.02
Media (animales)	2 848.68	3 212.83	2 960.57
Diferencia	2 069.94	2 394.52	2 106.55

Tabla. 4.9: Error cuadrático medio para las imágenes de prueba en la versión 4, con 200, 150 y 100 *epoch*.

Tomando estos resultados, está claro que aumentar el número de *epochs* incrementa considerablemente la calidad de reconstrucción de la imagen. Sin embargo, antes de decidir qué número de *epochs* es el adecuado, estudiaremos cómo se comporta el modelo modificando otros parámetros.

(a) 20 *epochs*

(b) 100 *epochs*

(c) 150 *epochs*

(d) 200 *epochs*

Figura 4.22: Reconstrucción de una imagen a los 20, 100, 150 y 200 *epochs* tras entrenar el modelo de la versión 4.

Versión 5

En esta versión modificaremos el *batch size* del modelo. Si reducimos el *batch size* haremos que el AE procese de forma simultánea un menor número de imágenes en cada *epoch*, aumentando así la capacidad de aprendizaje aunque incrementando también levemente el tiempo de entrenamiento de la red ya que se necesitarán más pasos para completar un *epoch*.

Se harán las mismas pruebas que en la versión anterior tomando 100, 150 y 200 *epochs*, esta vez con un *batch size* de 16 en lugar de 32, y se estudiará si merece la pena rebajar este valor.

La evolución de la función de pérdida y MSE en el entrenamiento del modelo se muestra en la figura 4.23. En la tabla 4.10 se detalla el valor MSE para cada una de las reconstrucciones de las imágenes de prueba. Se incluye también una reconstrucción a los 20, 100, 150 y 200 *epochs* en 4.24, donde se aprecia claramente la evolución del modelo.

La evolución de la pérdida del modelo y el valor MSE durante el entrenamiento es prácticamente idéntico a la versión anterior. En cambio, al analizar los valores MSE

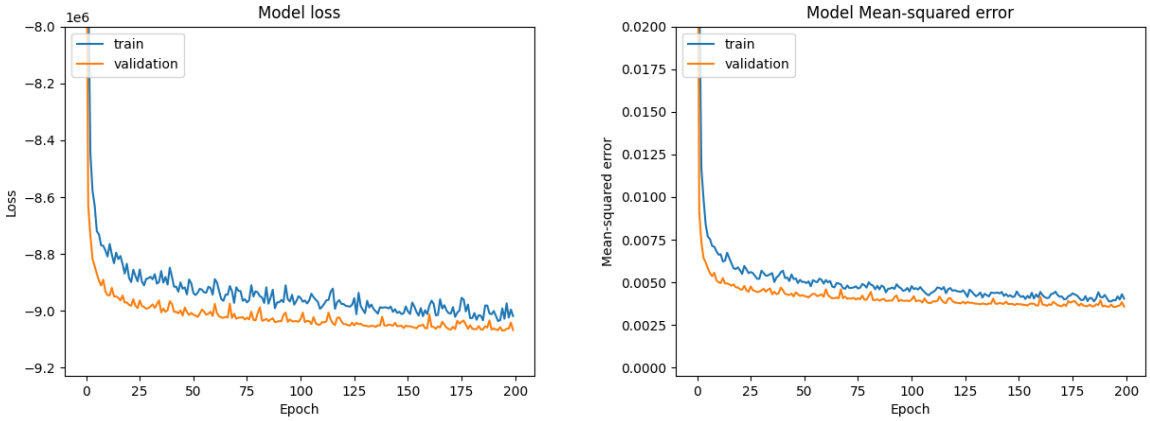


Figura 4.23: Evolución de la función de pérdida y MSE durante el entrenamiento para la versión 5.

Imagen	**MSE-200**	**MSE-150**	**MSE-100**
Imagen 0	3 385.48	3 874.87	3 351.46
Imagen 1	6 161.44	6 066.39	5 256.80
Imagen 2	1 275.01	1 287.64	1 240.27
Imagen 3	108.72	128.62	136.59
Imagen 4	650.03	621.17	782.03
Imagen 5	102.49	101.49	118.53
Imagen 6	354.32	418.40	458.76
Imagen 7	1 083.56	1 118.92	1 094.23
Imagen 8	1 987.73	1 984.65	1 933.29
Imagen 9	2 334.37	2 322.18	2 409.82
Media (vacío)	772.24	785.13	833.32
Media (animales)	3 202.41	3 303.38	2 945.45
Diferencia	2 430.17	2 518.25	2 112.13

Tabla. 4.10: Error cuadrático medio para las imágenes de prueba en la versión 5, con 200, 150 y 100 *epoch*.

para las imágenes de prueba, vemos cómo la distancia entre las imágenes vacías y con animales es superior en cualquiera de los escenarios, con 100, 150 y 200 *epochs*. Al igual que ocurría en la versión anterior, resalta por encima del resto el resultado tras 150 iteraciones. Es destacable que el tiempo de entrenamiento del modelo apenas se ha visto afectado por este cambio.

(a) 20 *epochs*

(b) 100 *epochs*

(c) 150 *epochs*

(d) 200 *epochs*

Figura 4.24: Reconstrucción de una imagen a los 20, 100, 150 y 200 *epochs* tras entrenar el modelo de la versión 5.

Presentados los resultados, podemos concluir que reduciendo el *batch size* a 16 e incrementando la duración del entrenamiento hasta los 150 *epochs* se consiguen los mejores resultados hasta ahora.

Versión 6

A pesar de que las reconstrucciones de la versión anterior parecen ser suficientes para cumplir los objetivos del TFG, se ha creído conveniente hacer más pruebas modificando los valores de los parámetros *steps per epoch* y *validation steps*.

Hasta ahora, ambos valores tenían un valor fijo de $\frac{1000}{batch\ size}$, haciendo que no se recorra por completo el conjunto de entrenamiento ni el conjunto de validación para este caso. En esta nueva versión se eliminarán dichas restricciones, manteniendo los parámetros por defecto de ambos valores, $\frac{Numero\ de\ datos}{batch\ size}$. De esta forma, en cada *epoch* se recorrerá el conjunto de datos al completo.

Como se indicó en la versión anterior, el modelo que mejor resultados proporcionaba tenía como parámetros 150 *epochs* y *batch size* 16. En esta versión se mantendrán dichos parámetros.

La evolución de la métrica MSE y la función de pérdida tras entrenar el nuevo modelo se aprecia en la figura 4.25.

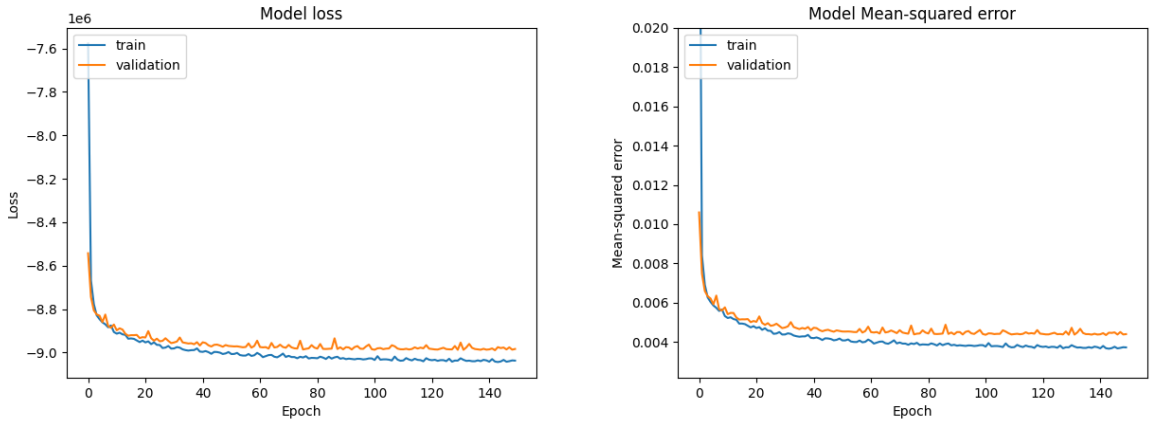


Figura 4.25: Evolución de la función de pérdida y MSE durante el entrenamiento para la versión 6.

Lo primero que llama la atención con respecto a las gráficas mostradas en otras versiones es la relación entre el resultado *train* y validación. Mientras que anteriormente los resultados de validación eran mejores que los de entrenamiento en este caso el comportamiento es inverso; hay peores resultados durante la validación. Realmente, este es el comportamiento esperado en las redes neuronales ya que es lógico que la red aporte mejores resultados con aquellos datos con los que se ha entrenado.

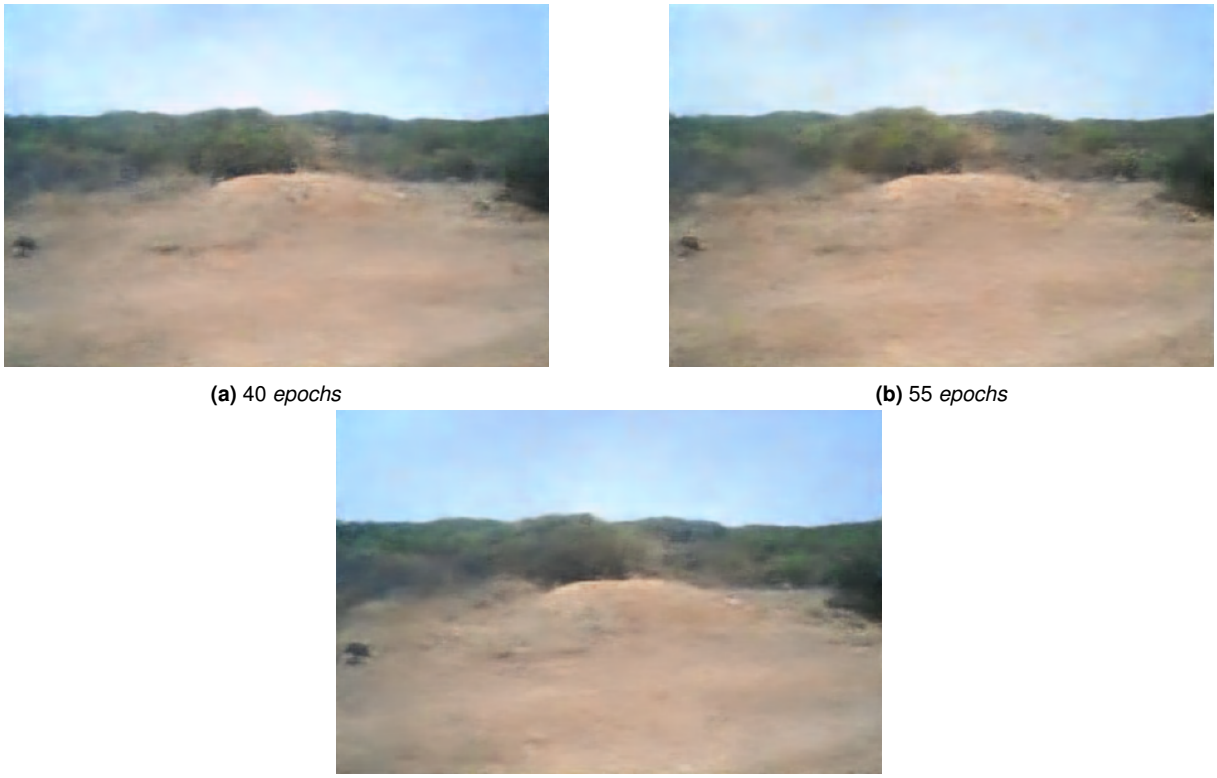
Es también destacable ver cómo los resultados de validación se estabilizan a partir de la iteración 70 mientras que los resultados sobre el conjunto de entrenamiento continúan bajando progresivamente. Este comportamiento se produce cuando el modelo sufre de *overfitting*, es decir, el ajuste excesivo para los datos de entrenamiento, provocando un estancamiento o empeoramiento de los resultados sobre el conjunto de validación. Este hecho demuestra que no es necesario establecer el número de *epochs* más allá de 70 ya que el resultado no mejorará.

En base a todo lo mostrado, es necesario incluir una nueva versión para estudiar el resultado del modelo bajo un número de *epochs* adecuado.

Versión 7

La siguiente iteración consistirá en analizar el modelo de la versión 6 tras 70, 55 y 40 *epochs*, zona a partir de la cual según la gráfica 4.25 los resultados de validación se vuelven casi constantes. Se estudiarán los resultados de cada uno y se decidirá el modelo final.

Los resultados de los modelos se pueden observar en la figura 4.26 y la tabla 4.11



(a) 40 *epochs*

(b) 55 *epochs*

(c) 70 *epochs*

Figura 4.26: Reconstrucción de una imagen a los 40, 55 y 70 *epochs* tras entrenar el modelo de la versión 7.

En cuanto a las imágenes reconstruidas, a simple vista apenas se aprecia diferencia entre los tres modelos a excepción de pequeños detalles de la imagen. Sí es más notoria la disparidad entre los modelos al conseguir el valor MSE de las imágenes de prueba. La mayor distancia entre los grupos vacío y con animales se consigue a los 70 *epochs* con 2279.45, quedando muy cerca el valor 2238.34, obtenido a los 55 *epochs*.

Otro aspecto destacable es el tiempo de entrenamiento del modelo. Para este caso, cada *epoch* requiere entre 55 y 65 segundos para completarse, habiendo así una diferencia de 15 minutos entre cada modelo.

Tras observar la ligera mejoría al utilizar 70 *epochs* y teniendo en cuenta que el tiempo de entrenamiento no se ve demasiado afectado, parece que ajustar de esta forma el parámetro es la mejor opción frente al resto.

La única decisión que queda por tomar es escoger como definitiva esta versión o la versión 5, en la que se modificaban los parámetros *steps per epoch* y *validation steps*. Por un lado, es cierto que en la versión 5 se consiguen unos resultados ligeramente superiores, aumentando la distancia entre el MSE de las imágenes con animales y el MSE de las imágenes vacías. Sin embargo, es destacable que en esta versión el

Imagen	MSE-70	MSE-55	MSE-40
Imagen 0	3 017.43	3 308.32	2 967.00
Imagen 1	5 875.80	5 491.48	5 368.22
Imagen 2	1 120.21	1 268.15	1 027.04
Imagen 3	122.53	127.10	119.07
Imagen 4	565.28	599.24	625.76
Imagen 5	75.07	77.67	107.53
Imagen 6	340.23	440.67	456.32
Imagen 7	1 090.67	1 184.22	1 204.01
Imagen 8	2 135.52	2 005.99	1 940.87
Imagen 9	2 353.01	2 251.95	2 398.06
Media (vacío)	757.79	780.14	818.45
Media (animales)	3 037.24	3 018.48	2 825.78
Diferencia	2 279.45	2 238.34	2 007.33

Tabla. 4.11: Error cuadrático medio para las imágenes de prueba en la versión 7, con 70, 55 y 40 *epoch*.

comportamiento de la red es más “normal”, algo que se ve reflejado en las gráficas 4.25, donde los resultados de entrenamiento superan a los resultados de validación, algo que no ocurría en la versión 5. Además, en la propia documentación de Keras se hace referencia a que el estado normal de los parámetros *steps per epoch* y *validation steps* es su valor por defecto, permitiendo que en cada *epoch* se aproveche al completo todas las instancias de datos disponibles. Únicamente se recomienda ajustar estos parámetros cuando se ha introducido una fase de *data augmentation* a los datos existentes, es decir, el incremento artificial del número de instancias modificando ligeramente los datos disponibles.

4.8.2. Evaluación del modelo final

Después de realizar todos los experimentos y versiones mostradas a lo largo de este apartado, podemos decantarnos por la arquitectura y parámetros definitivos:

- **Número de *epochs*:** 70
- **Batch size:** 16
- **Optimizador:** Adam

- **Ratio de aprendizaje:** $0{,}001$
- **Steps per epoch:** $\frac{Numero\ de\ datos\ para\ entrenamiento}{batch\ size}$
- **Validation steps:** $\frac{Numero\ de\ datos\ para\ validacion}{batch\ size}$
- **Arquitectura**: Mostrada en la figura 4.12. El tamaño de la capa de entrada queda definido en $384x256$ píxeles

Tras definir la arquitectura y parámetros del modelo, comprobaremos su funcionamiento evaluando el modelo con las imágenes vacías pertenecientes al conjunto de *test*. El resultado obtenido es un valor MSE de 0.0043. Este valor es muy similar al que se obtenía durante el entrenamiento, mostrando que la red posee la capacidad de generalizar el conocimiento adquirido durante el entrenamiento hacia nuevas imágenes.

El último paso es utilizar esta misma arquitectura y parámetros para construir siete AE, uno por cada grupo o *cluster* de imágenes. Estos AE se almacenarán y utilizarán más adelante para la evaluación del modelo completo definido en 2.13.

4.9. Implementación y entrenamiento del clasificador

Hasta este punto, se han entrenado siete AE capaces de reconstruir en la salida la imagen que se toma como entrada bajo la premisa de que, si la imagen contiene un animal, su reconstrucción será peor.

Siguiendo el esquema global del software mostrado anteriormente en la figura 4.7, el último elemento a desarrollar será un clasificador capaz de predecir en base al error de reconstrucción si hay un animal o no en una fotografía. Como entrada, inicialmente tomará tres tipos de errores o diferencias entre la imagen original y su reconstrucción: MSE (error cuadrático medio), MAE (error absoluto medio) y SSIM (similaridad estructural). También se harán pruebas incluyendo el número de *cluster* asociado a la imagen, analizando si merece o no la pena incluir este dato en la entrada del clasificador.

Teniendo en cuenta que este TFG se centra en abordar el problema utilizando técnicas propias del DL y para continuar la misma línea de actuación, se construirá una red neuronal densamente conectada que actúe como clasificadora.

4.9.1. Elección de la arquitectura y entrenamiento

Una vez definido el tipo de modelo que se utilizará, es necesario establecer la arquitectura para su posterior entrenamiento.

A diferencia de lo que requeríamos del *dataset* para los AE, el entrenamiento de este modelo de red neuronal necesitará tanto imágenes vacías como imágenes con animales, de forma que se puedan captar las diferencias entre un tipo y otro. Sobre este conjunto de imágenes, para cada una se almacenará en un fichero los errores de reconstrucción y una etiqueta que indica si la imagen contiene realmente o no un animal. En la figura 4.27 se muestran 10 líneas de este fichero. Los datos almacenados servirán como conjunto de entrenamiento para la red.

	MSE	MAE	SSIM	Etiqueta
0	0.058056	0.083454	0.675811	1
1	0.157436	0.193699	0.300501	1
2	0.000959	0.012773	0.950198	0
3	0.226353	0.227295	0.686741	1
4	0.003955	0.026419	0.844570	0
5	0.005408	0.029255	0.797501	0
6	0.272516	0.262538	0.772409	1
7	0.001253	0.015093	0.940552	0
8	0.000858	0.012107	0.952025	0
9	0.103898	0.134440	0.458060	1

Figura 4.27: Diez primeras filas del fichero de errores. Cada fila representa los errores MSE, MAE y SSIM entre una imagen original y su reconstrucción junto a la etiqueta real, en la que 0 significa imagen vacía y 1 imagen con animales.

Como primera aproximación, se propone la arquitectura mostrada en la figura 4.28. Contiene una capa de entrada de tamaño 3 (pasaremos como entrada tres errores), dos capas densamente conectadas con veinte neuronas cada una y una capa de salida de dos neuronas con función de activación *softmax*, permitiendo clasificar la entrada en una de las dos clases.

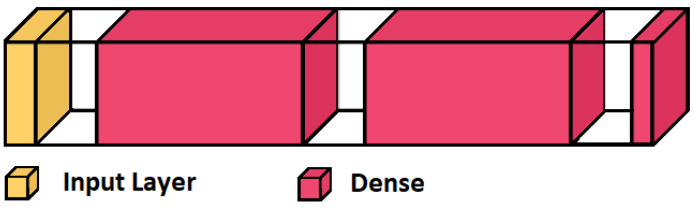


Figura 4.28: Arquitectura inicial de la red densamente conectada.

Los parámetros de la red neuronal se detallan a continuación:

- **Número de epoch:** 150
- **Batch size:** 16 (por defecto)
- **Optimizador:** Adam
- **Ratio de aprendizaje:** $0{,}001$ (por defecto)

Establecido el modelo, se procede al entrenamiento. La evolución de la métrica *accuracy* o exactitud proporcionada por Keras se puede apreciar en la gráfica 4.29.

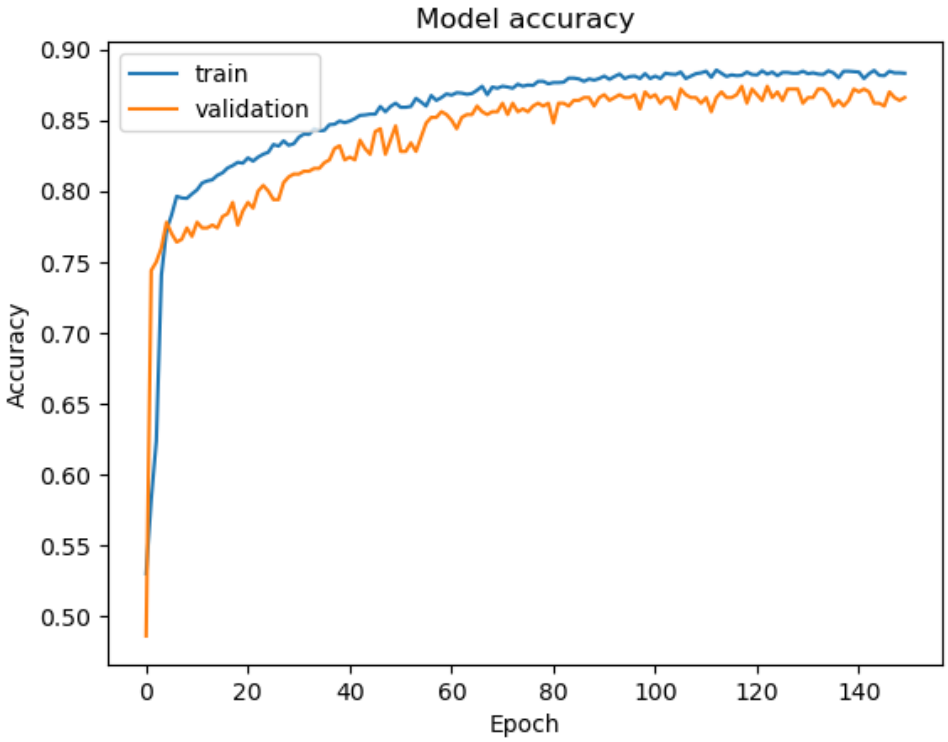
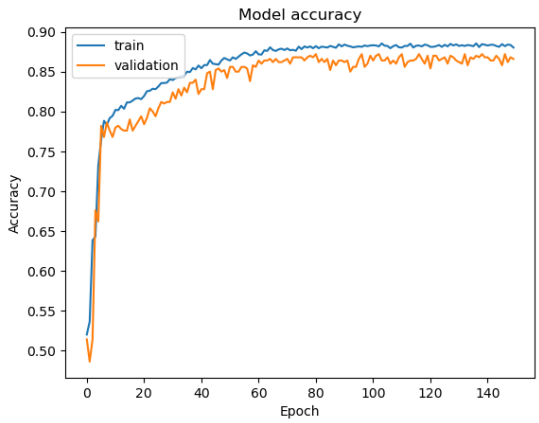


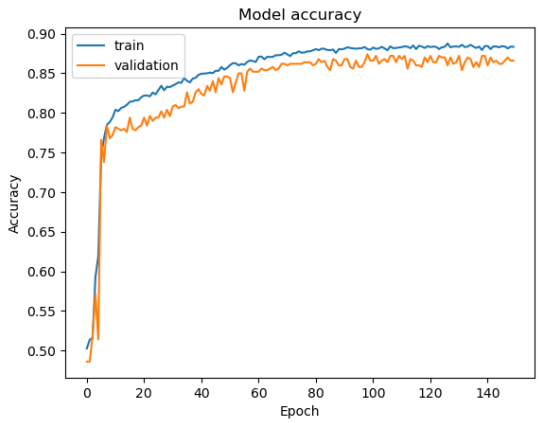
Figura 4.29: Evolución de la exactitud de la red densamente conectada en su primera versión.

Como se aprecia en la gráfica, la exactitud del modelo se estabiliza aproximadamente a partir de los 120 *epochs* en valores cercanos a 0.87 para el conjunto de entrenamiento y 0.86 para el conjunto de validación. Con el objetivo de agilizar el proceso de entrenamiento, se decide reducir el número de *epochs* a 120.

Se han analizado otras versiones del modelo para comprobar si aumentando el número de capas o el número de neuronas mejora el resultado de la red. En la figura 4.30 se muestra la evolución del rendimiento del modelo durante el entrenamiento de dos modelos con una ligera variación frente al original tras aumentar a tres el número de capas ocultas y aumentar el número de neuronas de 20 a 40 por cada capa oculta.



(a) Modelo con dos capas de 40 neuronas cada una.



(b) Modelo con tres capas de 20 neuronas cada una.

Figura 4.30: Evolución de la exactitud para dos modelos de red densamente conectada, el primero con dos capas ocultas de 40 neuronas cada una y el segundo con tres capas ocultas de 20 neuronas.

Según los resultados, el resultado de ambos modelos es prácticamente idéntico a la arquitectura original, haciendo ver que aumentar la capacidad de la red no implicará mejorar los resultados.

4.9.2. Inclusión del número de *cluster* como entrada a la red densa

Una vez hechos estos experimentos tomando como entrada los tres errores, MSE, MAE y SSIM, procedemos a incluir el número de *cluster* de la imagen como información adicional.

Cada AE ha sido entrenado para un *cluster* específico, tomando como entrenamiento únicamente las imágenes de ese grupo. Esto podría provocar que la calidad de reconstrucción de cada AE varíe pese a tener la misma arquitectura en función del número de imágenes del grupo, la complejidad de las mismas, etc. Si varía la calidad de reconstrucción, es previsible que también cambie el error entre las imágenes originales y las imágenes reconstruidas por cada grupo, variando el umbral de corte entre las imágenes vacías y con animales. Por esta razón, parece útil incluir como entrada a la red neuronal el número de *cluster* asociado a la imagen. De esta forma, podría diferenciar el error entre un grupo y otro y establecer para cada uno el "umbral" correspondiente.

Así, en los siguientes experimentos la red pasará a tener como entrada un vector de cuatro valores: MSI, MAE, SSIM y $ID_{cluster}$, siendo $ID_{cluster}$ el identificador del cluster, un número entero que toma valores [0,6].

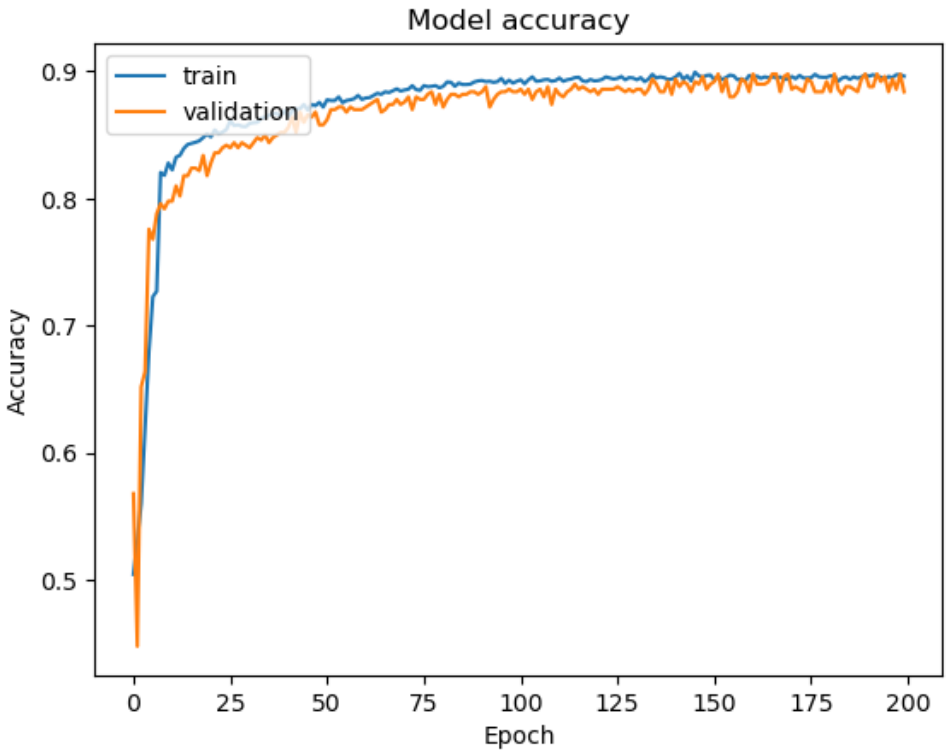


Figura 4.31: Evolución de la exactitud de la red densamente conectada en su segunda versión, tomando como entrada el índice de *cluster*.

La arquitectura que utilizaremos será idéntica a la mostrada en el apartado anterior: dos capas densamente conectadas de 20 neuronas cada una. Se analizará bajo 200 *epochs* y en base a los resultados se decidirá el número de *epochs* óptimo.

La evolución del rendimiento del modelo durante el entrenamiento se ve reflejado en la gráfica 4.31.

Como podemos ver, los resultados son ligeramente superiores a la primera versión en la que no se incluía el índice de *cluster*, algo que comprobaremos al evaluar el modelo sobre el conjunto de *test*. Sí se detecta con facilidad que no son necesarios 200 *epochs* ya que a los 150 *epochs* la exactitud del modelo se mantiene prácticamente constante. Por esta razón, se escogerán 150 *epochs*.

De esta forma, como se obtienen mejores resultados se ha decidido que el modelo MLP contará con una entrada de cuatro valores: tres errores y el índice de *cluster*.

4.9.3. Arquitectura y parámetros escogidos para la red clasificadora

Tras la realización de todos los experimentos mostrados, se detallan las características de la red neuronal densamente conectada utilizada para la clasificación:

- **Número de epoch:** 120
- **Batch size:** 16 (por defecto)
- **Optimizador:** Adam
- **Ratio de aprendizaje:** $0{,}001$ (por defecto)
- **Arquitectura:** Mostrada en la figura 4.28
- **Valores de entrada:** Errores MSE, MAE, SSIM e índice del *cluster* asociado a la imagen.

Capítulo 5

RESULTADOS

En este capítulo se presentarán y analizarán los resultados que se han obtenido tras la aplicación de los métodos indicados en el capítulo anterior, mostrando si la arquitectura propuesta es adecuada para la resolución del problema analizado en este TFG.

5.1. Reconstrucción de los autoencoders

La primera sección estará dedicada a mostrar los resultados de la reconstrucción de los siete AE, entrenados cada uno en un único *cluster*. Para ello, se escogerá de forma aleatoria una imagen vacía de cada cluster y se reconstruirá con el AE correspondiente, mostrando el MSE entre la imagen original y la reconstrucción. Además, se presentará la reconstrucción de tres de estas imágenes para comprobar visualmente los resultados.

La tabla 5.1 recoge el MSE entre las imágenes originales y reconstruidas de cada *cluster*. Por otro lado, en la figura 5.1 se observa la reconstrucción de tres imágenes, pertenecientes a tres *clusters* distintos.

Si comparamos esta reconstrucción con las que se lograban haciendo uso del AE genérico (no especializado en ningún *cluster* concreto) en la figura 4.26 y tabla 4.11 (tomando 70 *epochs*, la versión definitiva), podemos apreciar resultados claramente mejores. La reconstrucción a nivel visual produce unos resultados muy cercanos a la imagen original, llegando a captar incluso detalles del terreno que podrían pasar desapercibidos, como pequeñas piedras o sombras. Como es lógico, esto se ve reflejado en el MSE entre las imágenes originales y su posterior reconstrucción, llegando a te-

Imagen	MSE
Imagen 0	43.31
Imagen 1	22.78
Imagen 2	206.55
Imagen 3	836.26
Imagen 4	209.38
Imagen 5	375.47
Imagen 6	277.45
Media (vacío)	281.06

Tabla. 5.1: Error cuadrático medio para siete imágenes vacías, cada una perteneciente a un *cluster* distinto. El error se ha obtenido haciendo uso del AE correspondiente al *cluster*.

ner un error medio de solo 281.06, cuando la media conseguida para imágenes vacías con el AE general fue de 757.79.

Tal y como hemos mencionado, las reconstrucciones de los AE parecen ser lo suficientemente buenas como para abordar con éxito el problema.

5.2. Resultados de la clasificación

En esta sección se mostrarán los resultados obtenidos en la clasificación de las imágenes llevada a cabo por la red neuronal densamente conectada presentada en la sección 4.9.3. Sobre estos resultados se obtendrán diferentes estadísticas y gráficas que nos permitirán conocer más detalladamente los resultados [1].

Lo primero que podemos analizar es la métrica ***accuracy* o exactitud** del modelo sobre el conjunto de *test*. Esta medida representa la proporción de muestras que han sido clasificadas correctamente, independientemente de su clase. La exactitud del modelo se obtendrá utilizando el método *evaluate()* de la librería Keras.

Dicho método afirma que la exactitud de la red MLP y por tanto la exactitud del modelo completo desarrollado a lo largo del TFG es de $0{,}8919$, es decir, 89 de cada 100 imágenes son clasificadas correctamente en uno de los dos grupos, Animales o Vacío.

[1] Muchas de las gráficas han sido obtenidas haciendo uso de la librería Scikit-learn [49]

(a) Original 1

(b) Reconstrucción 1

(c) Original 2

(d) Reconstrucción 2

(e) Original 3

(f) Reconstrucción 3

Figura 5.1: Muestra de la reconstrucción de tres imágenes pertenecientes a distintos *clusters* haciendo uso del AE correspondiente a cada grupo.

Este valor se puede comparar con el obtenido haciendo uso de la red MLP al tomar como entrada únicamente los tres valores de error, no el índice de *cluster* asociado a la imagen. En ese caso, la exactitud obtenida se reducía ligeramente, consiguiendo un valor de $0{,}8707$. Con estos datos, podemos comprobar que ha sido buena idea incluir el índice de *cluster* como entrada a la red.

Además de la exactitud del modelo, es interesante conocer los detalles de la matriz de confusión de los resultados. La matriz de confusión es un método ampliamente utilizado en problemas de clasificación que permite recoger los resultados obtenidos y plasmarlos en una matriz como la mostrada en la figura 5.2 haciendo distinción entre dos clases: la clase positiva y la clase negativa. Esta representación hace mucho

más sencilla la tarea de analizar e interpretar los resultados de la clasificación. Las etiquetas TN y TP representan las clasificaciones correctas, ya sean negativas o positivas. Por su lado, FN y FP hacen referencia a clasificaciones erróneas negativas y clasificaciones erróneas positivas respectivamente.

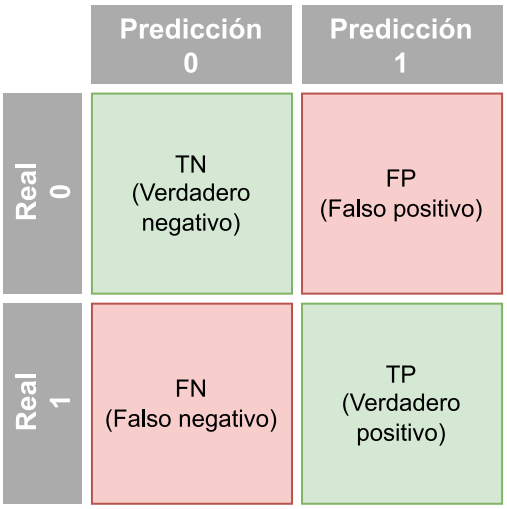


Figura 5.2: Representación de una matriz de confusión. El eje X representa las predicciones y el eje Y la realidad. Se presentan los verdaderos negativos (TN), falsos negativos (FN), falsos positivos (FP) y verdaderos positivos (TP).

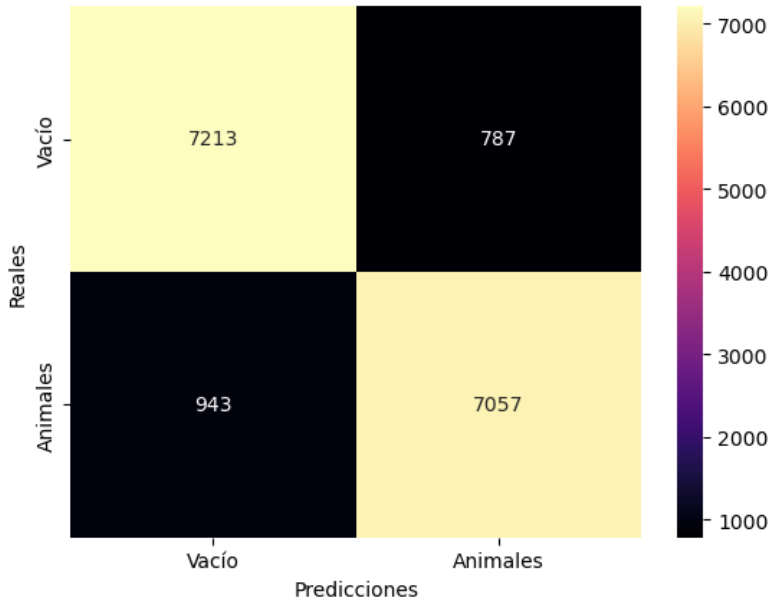


Figura 5.3: Matriz de confusión para mostrar los resultados obtenidos. El eje X representa las predicciones. El eje Y representa la realidad.

Esta definición se puede extrapolar al problema que nos ocupa. En este caso tendremos dos clases: vacío y animales. Basándonos en los resultados de la red, podemos clasificar cada predicción en una de las categorías que permite la matriz de confusión, que en nuestro caso pasarán a ser verdadero vacío, verdadero animales,

falso animales y falso vacío. La matriz de confusión resultante se presenta en la figura 5.3.

Podemos comprobar que, tal y como afirmaba la métrica de exactitud del modelo, las predicciones correctas de ambas clases sobresalen por encima de las predicciones erróneas, obteniendo 7213 predicciones correctas para la clase Vacío y 7057 para la clase Animales. Es destacable que, a pesar de que la clase Vacío posee más predicciones correctas que la clase Animales, también posee un número mayor de predicciones erróneas, 943 frente a 787.

Para seguir profundizando sobre la matriz de confusión, es necesario definir tres métricas más: precisión, *recall* o exhaustividad y puntuación F1.

La **precisión** define la proporción de predicciones correctas para cada clase. Su fórmula se describe en la ecuación 5.1, siguiendo los acrónimos incluidos en la figura 5.2.

$$Precision = \frac{TP}{TP + FP} \tag{5.1}$$

Tomando la clase positiva como Vacío, la precisión será $\frac{7213}{7213+943} = 0{,}884$. De las predicciones vacías, el $88{,}4\,\%$ son correctas.

Tomando la clase positiva como Animales, la precisión será $\frac{7057}{7057+787} = 0{,}900$. De las predicciones animales, el $90\,\%$ son correctas.

Recall es la medida que indica la proporción de predicciones correctas para cada clase sobre el número total de instancias de dicha clase. La ecuación 5.2 describe su obtención.

$$Recall = \frac{TP}{TP + FN} \tag{5.2}$$

Tomando la clase positiva como Vacío, *recall* será $\frac{7213}{7213+787} = 0{,}902$. De las 8000 muestras vacías, se han detectado y clasificado correctamente el $90{,}2\,\%$.

Tomando la clase positiva como Animales, la precisión será $\frac{7057}{7057+943} = 0{,}882$. De las 8000 muestras de animales, se han detectado y clasificado correctamente el $88{,}2\,\%$.

Por último, la **métrica F1** consiste en una combinación entre precisión y *recall*, generalmente obtenida como la media armónica entre ambas, descrita en la ecuación 5.3.

$$F1\ score = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{5.3}$$

Siguiendo esta ecuación, podemos calcular la puntuación F1 para cada una de las clases. Para la clase Vacío, la puntuación F1 es de $0{,}892$, mientras que para la clase Animales su valor es de $0{,}890$, ambos valores prácticamente idénticos indicando que el modelo clasifica de forma igualmente buena tanto imágenes con animales como imágenes vacías.

En la tabla 5.2 se muestra un resumen con los resultados para estas tres métricas.

Clase	**Precisión**	**Recall**	**F1-score**
Vacío	0.884	0.902	0.892
Animales	0.900	0.882	0.890

Tabla. 5.2: Cuadro resumen con los valores de precisión, *recall* y F1 para las clases Vacío y Animales.

Haciendo uso de los valores de la matriz de confusión presentada podemos obtener otras medidas interesantes: la curva ROC y el área bajo la curva.

La **curva ROC** es una de las medidas de rendimiento para los problemas de clasificación. Esta medida se representa en un gráfico, enfrentando el ratio de verdaderos positivos (en rango [0,1]) frente al ratio de falsos positivos (también en rango [0,1]) bajo diferentes umbrales. El ratio de verdaderos positivos, cuyo valor coincide con la métrica *recall*, se define en la ecuación 5.4 y el ratio de falsos positivos en 5.5.

$$Ratio\ de\ verdaderos\ positivos = \frac{TP}{TP + FN} \tag{5.4}$$

$$Ratio\ de\ falsos\ positivos = \frac{FP}{FP + TN} \tag{5.5}$$

El **área bajo la curva ROC** es la medida que indica la calidad de la curva formada. A mayor área, mejor calidad tendrá el modelo. Como la curva ROC se representa sobre un área cuadrada de lado 1, el área bajo la curva se encontrará en el rango de valores [0,1].

La red de clasificación implementada no devuelve directamente una etiqueta de la clase asociada a la entrada, sino que devuelve un vector con la probabilidad de que pertenezca a una clase u otra, es decir, para cada imagen devuelve la probabilidad

de que sea una imagen vacía y la probabilidad de que sea una imagen con animales. Para decidir la etiqueta final, se establece un umbral que, para este caso, ha sido $0{,}5$. De esta forma, si para una instancia la probabilidad de una clase determinada supera el valor $0{,}5$, la etiqueta final será la de dicha clase.

Tomando en cuenta que la clase positiva es Animales, si escogemos un umbral 0 implicará que el ratio de verdaderos positivos es 1 (todas las muestras han sido clasificadas como Animales) a la vez que el ratio de falsos positivos también será 1 (todas las muestras vacías se han clasificado como Animales). Al escoger un umbral 1 el comportamiento es justo el contrario, el ratio de verdaderos positivos será 0 (todas las muestras han sido clasificadas como Vacío) y el ratio de falsos positivos será también 0 (no se ha clasificado ninguna muestra como Animales cuando en realidad es Vacío).

La curva ROC se forma estableciendo numerosos umbrales distintos y analizando el ratio de verdaderos positivos y falsos positivos para cada uno. En la figura 5.4 se muestra la curva ROC del modelo desarrollado.

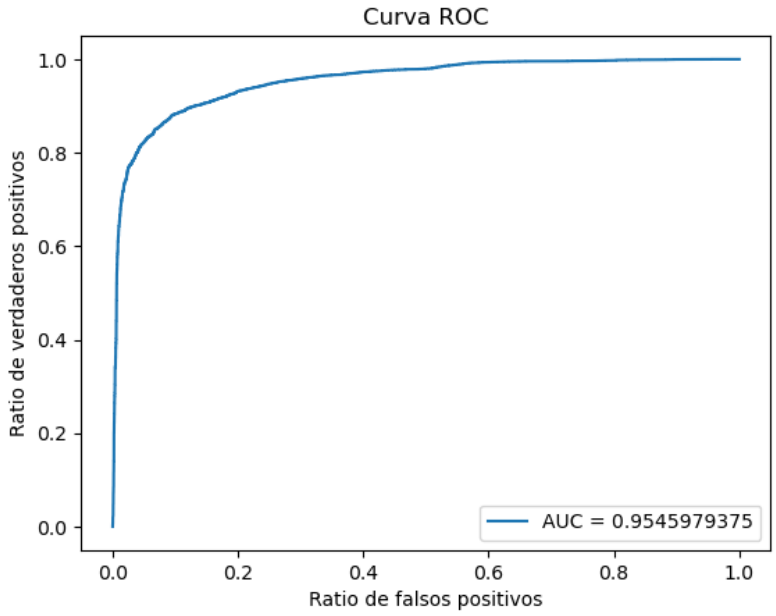


Figura 5.4: Curva ROC y área bajo la curva para el modelo software desarrollado.

Interpretar el resultado de la curva ROC es sencillo si definimos cuál sería la forma de la gráfica para un clasificador perfecto y para un clasificador aleatorio. El comportamiento ideal es la maximización de verdaderos positivos a la vez que se minimizan los falsos positivos. Siguiendo esta lógica, la "curva ROC perfecta" es aquella que forma una esquina de 90° en el extremo superior izquierdo, lo que indicaría que existe un umbral idóneo en el que el ratio de verdaderos positivos es 1 y el ratio de falsos positivos es 0: un clasificador perfecto. El área bajo la curva de este tipo de clasificadores será 1. Por su lado, un clasificador aleatorio formará una curva ROC representada co-

mo una recta desde el punto $(0, 0)$ a $(1, 1)$, indicando que para el mejor umbral tanto el ratio de verdaderos positivos como falsos positivos es 0.5. El área bajo la curva para un clasificador aleatorio será de 0.5.

Tomando como base los extremos mencionados en el párrafo anterior, podemos concluir que nuestro modelo es capaz de obtener unos muy buenos resultados, permitiendo obtener un ratio de verdaderos positivos grande manteniendo a la vez pequeño el ratio de falsos positivos. Es posible cuantificar la calidad de la curva gracias al área bajo la curva ROC, $0{,}955$ para nuestro caso.

La tabla 5.3 engloba un resumen con todas las estadísticas recogidas a partir de los resultados del modelo.

Clase	**Precisión**	**Recall**	**F1-score**	**AUC**	**Exactitud**
Vacío	0.884	0.902	0.892	-	-
Animales	0.900	0.882	0.890	-	-
General	0.892	0.892	0.891	0.955	0.892

Tabla. 5.3: Cuadro resumen con los valores de precisión, *recall*, F1, área bajo la curva ROC y exactitud del sistema desarrollado.

Capítulo 6

CONCLUSIONES

En este último capítulo se hablará sobre las conclusiones alcanzadas tras la finalización del proyecto. Se hará un breve resumen de lo que se ha hecho y se ha conseguido, los conocimientos adquiridos durante la realización del TFG y el trabajo futuro para seguir mejorando el proyecto.

6.1. Trabajo realizado

A lo largo de este TFG se han desarrollado diferentes técnicas y modelos para conseguir un objetivo final presentado en el capítulo 3: un modelo basado en técnicas de IA entrenado con imágenes de dispositivos de fototrampeo instalados en parques naturales para la detección y descarte de imágenes vacías (sin animales).

Para alcanzar esta meta, tal y como se analizó en el capítulo 4 se ha realizado un trabajo completo de análisis exploratorio y preprocesamiento de las imágenes de los dispositivos de fototrampeo, desarrollando los *scripts* necesarios para ello. Seguidamente, se propuso y diseñó un modelo global basado en la reconstrucción óptima de imágenes vacías con modelos DL que nos permita diferenciar entre imágenes vacías e imágenes con animales. El sistema completo está dividido en diferentes etapas:

1. En primer lugar se utiliza la información de los histogramas de las imágenes con el objetivo de detectar y crear grupos de imágenes con características similares. Para ello se desarrolló un algoritmo de *clustering* que dividió toda la BBDD en siete grupos o *clusters*.

2. Para cada uno de estos grupos, se diseñan modelos AE correspondientes. De esta forma, al especializar cada AE en imágenes relativamente parecidas, la reconstrucción será previsiblemente mejor que si utilizamos un único AE general para todas las imágenes.

3. A partir de los siete modelos, obtenemos para cada imagen los errores MSI, MAE y SSIM entre la imagen original y su posterior reconstrucción. Como los AE han sido entrenados únicamente con imágenes vacías, es de esperar que la reconstrucción de imágenes con animales sea peor.

4. Tanto los errores entre la imagen original y la reconstrucción como el índice de *cluster* asociado a cada imagen pasarán a formar la entrada de la red neuronal clasificadora densamente conectada. La red será la encargada de decidir, para cada imagen, el grupo al que pertenece: imágenes vacías o imágenes con animales.

Todo este sistema desarrollado llevó a los resultados mostrados en el capítulo 5, donde vimos que, en general, eran muy buenos, manteniendo una exactitud de casi un 90 %, es decir, nueve de cada diez imágenes serán clasificadas correctamente.

Los puntos débiles del modelo recaen en que, en ocasiones, el error de reconstrucción de imágenes con animales puede ser inferior al error de reconstrucción de imágenes vacías. Podemos ver un ejemplo en la tabla 5.1 mostrada en el capítulo anterior. La imagen con menor error de reconstrucción es la imagen 1, con un error de 22.78. En contraste, la imagen 3 posee un error considerablemente mayor, de 836.26 (ambos errores son MSE). Al analizar los detalles de la imagen, las cuales se pueden ver en la figura 5.1 (imágenes 1 y 3), comprobamos que la imagen 1 se corresponde a una fotografía nocturna, mientras que la imagen 3 es una fotografía de una amplia zona con pequeñas piedras y arbustos al fondo. Podemos deducir por qué hay una diferencia de error tan amplia. En la imagen nocturna aparece una zona muy grande únicamente negra, con una reconstrucción perfecta. Sin embargo, para la imagen 3 es difícil reconstruir todas las pequeñas piedras del suelo y los detalles de los arbustos, dando por tanto un error mayor.

Este hecho provoca que en ocasiones la reconstrucción de una imagen vacía sea peor que la reconstrucción de una imagen con animales, haciendo fallar al modelo de clasificación.

6.2. Conocimientos adquiridos

Uno de los objetivos que se planteaban además de crear un sistema para resolver el problema era la adquisición de conocimiento. En esta breve sección me gustaría hablar sobre todo lo que he aprendido durante la realización del proyecto.

Como es lógico, he aprendido muchas cosas sobre ciencia de datos, concretamente en el área del DL. Cómo se hace un preprocesamiento a un conjunto de imágenes o la importancia de separar los datos en conjuntos de entrenamiento, *test* y validación. Cómo se implementan redes neuronales en Python y qué herramientas tenemos a nuestra disposición para esa tarea o cómo funciona un algoritmo de *clustering* de histogramas. La mayoría de estos conocimientos eran nuevos para mi, ya que muchas de estas cosas apenas se ven en las asignaturas del grado de Informática.

Sin embargo no todos los conocimientos son referentes a la informática o la programación. También he aprendido cómo se puede gestionar un proyecto a largo plazo, más grande y realista que las prácticas de asignaturas a las que estoy acostumbrado, o técnicas de gestión del tiempo para compatibilizar el trabajo junto al resto de asignaturas pendientes.

Además, me gustaría destacar el hecho de haber aprendido cómo y dónde buscar referencias bibliográficas adecuadas y fiables.

6.3. Conclusiones

Tras la creación del sistema y la redacción de la memoria que lo acompaña, podemos afirmar que, aunque los resultados no son perfectos, se han cumplido todos los objetivos propuestos, tanto los referentes al ámbito de la informática (desarrollo del sistema, tratamiento de los datos, etc.), como los objetivos personales de adquisición de conocimiento.

La utilidad del TFG recae en reducir de forma notable el esfuerzo de descartar imágenes vacías en imágenes de fototrampeo que, tal y como se expuso en el capítulo de introducción, carecen de utilidad para los profesionales. Gracias a este proyecto el descarte de estas imágenes será automático, evitando dedicar tiempo y esfuerzo al descarte manual para emplearlo en lo más importante, el análisis y estudio de las imágenes en las que aparezcan animales.

Por otro lado, con este TFG se ha demostrado la viabilidad del uso de autoencoders junto a un clasificador para el descarte de imágenes vacías y, en general, la detección de anomalías.

En conclusión, desde un punto de vista personal creo que la elaboración de este TFG ha supuesto una ampliación excelente de los estudios asociados al grado, incrementando su valor al contribuir a la solución de un problema real.

6.4. Trabajo futuro

Como última sección, se incluyen algunas posibles propuestas para en un futuro mejorar el software desarrollado:

- Mejora del algoritmo K-Means. El algoritmo de agrupamiento implementado permite separar las imágenes en 7 grupos distintos según las características internas de las imágenes. Sin embargo, a pesar de que en la experimentación el resultado parecía ser adecuado, al aplicar el algoritmo sobre el conjunto de datos completo se observa que la separación no es tan buena, existiendo algunos *clusters* con miles de imágenes y otros con apenas algunos cientos. Se propone o bien trabajar con un subconjunto de datos mayor para que la experimentación sea más realista o realizar un preprocesamiento previo sobre las imágenes, como la ecualización del histograma.
- Para los AE también se pueden introducir algunas mejoras como, por ejemplo, obtener la arquitectura y parámetros idóneos para cada uno de los siete AE implementados (uno por cada *cluster* de imágenes) en lugar de utilizar una arquitectura estándar. Además, se podría probar con otros tipos de AE como *denoising AE* y comparar los resultados.
- Creación de una aplicación para el usuario. En este momento, aunque se han desarrollado todos los modelos necesarios para obtener los resultados, no se ha implementado un software que pueda manejar un usuario cualquiera. Una buena característica a incluir para este proyecto sería una aplicación de usuario en la que, seleccionando una carpeta con imágenes en su interior, la aplicación sea capaz de separar todas las imágenes en dos nuevas carpetas, una para las imágenes vacías y otra para las imágenes con animales.

Bibliografía

[1] Luigi Boitani. *Camera trapping for wildlife research*. Pelagic Publishing Ltd, 2016.

[2] Natasha De Bondi, John G White, Mike Stevens, and Raylene Cooke. A comparison of the effectiveness of camera trapping and live trapping for sampling terrestrial small-mammal communities. *Wildlife research*, 37(6):456–465, 2010.

[3] Samoili S, Lopez Cobo M, Gomez Gutierrez E, De Prato G, Martinez-Plumed F, and Delipetrev B. *AI WATCH. Defining Artificial Intelligence. Towards an operational definition and taxonomy of artificial intelligence*. Publications Office of the European Union, Luxembourg (Luxembourg), 2020. ISBN 978-92-76-17045-7. doi: $10.2760/382730(\text{online})$.

[4] Blagoj Delipetrev, Chrysi Tsinaraki, and Uros Kostic. Historical evolution of artificial intelligence. 2020.

[5] Nick Bostrom. *Superintelligence: Paths, Dangers, Strategies*. Oxford University Press, 2014. ISBN 978-0-19-967811-2.

[6] Tom M Mitchell and Tom M Mitchell. *Machine learning*, volume 1. McGraw-hill New York, 1997.

[7] Osvaldo Simeone. A very brief introduction to machine learning with applications to communication systems. *IEEE Transactions on Cognitive Communications and Networking*, 4:648–664, 2018.

[8] Batta Mahesh. Machine learning algorithms-a review. *International Journal of Science and Research (IJSR).*, 9:381–386, 2020.

[9] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521 (7553):436–444, 2015.

[10] Nada and Nadia Berchane. Artificial intelligence, machine learning, and deep learning: Same context, different concepts, 2018. URL https://cutt.ly/yKSVYnV. Comprobado en 29-06-2022.

[11] Favio Vázquez. A "weird" introduction to deep learning, 2018. URL https://cutt.ly/uKJM7S1. Comprobado en 29-06-2022.

[12] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[13] Donald Olding Hebb. *The organisation of behaviour: a neuropsychological theory*. Science Editions New York, 1949. ISBN 9781410612403 (eBook). doi: https://doi.org/10.4324/9781410612403.

[14] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[15] David E Rumelhart, Richard Durbin, Richard Golden, and Yves Chauvin. Backpropagation: The basic theory. *Backpropagation: Theory, architectures and applications*, pages 1–34, 1995.

[16] Pier Luigi Mazzeo and Paolo Spagnolo. *Deep Learning Applications*. IntechOpen, Rijeka, 2021. doi: 10.5772/intechopen.91576. URL https://doi.org/10.5772/intechopen.91576.

[17] Arshia Rehman, Saeeda Naz, Muhammad Imran Razzak, Faiza Akram, and Muhammad Imran. A deep learning-based framework for automatic brain tumors classification using transfer learning. *Circuits, Systems, and Signal Processing*, 39(2):757–775, 2020.

[18] Stephen Balaban. Deep learning and face recognition: the state of the art. *Biometric and surveillance technology for human and activity identification XII*, 9457: 68–75, 2015.

[19] Daniel W Otter, Julian R Medina, and Jugal K Kalita. A survey of the usages of deep learning for natural language processing. *IEEE transactions on neural networks and learning systems*, 32(2):604–624, 2020.

[20] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, L. Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 2016.

[21] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[22] Mariusconstantin Popescu, Valentina Emilia Balas, Liliana Perescu-Popescu, and Nikos E. Mastorakis. Multilayer perceptron and neural networks. *WSEAS Transactions on Circuits and Systems archive*, 8:579–588, 2009.

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60:84 – 90, 2012.

[24] Diego Calvo. Red neuronal convolucional cnn, 2017. URL https://cutt.ly/AKJO5fN. Comprobado en 29-06-2022.

[25] Jane Bromley, James W. Bentz, Léon Bottou, Isabelle Guyon, Yann LeCun, Cliff Moore, Eduard Säckinger, and Roopak Shah. Signature verification using a "siamese" time delay neural network. In *Int. J. Pattern Recognit. Artif. Intell.*, 1993.

[26] Sounak Dey, Anjan Dutta, Juan Ignacio Toledo, Suman K. Ghosh, Josep Lladós, and Umapada Pal. Signet: Convolutional siamese network for writer independent offline signature verification. *CoRR*, abs/1707.02131, 2017. URL http://arxiv.org/abs/1707.02131.

[27] Alex Sherstinsky. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *ArXiv*, abs/1808.03314, 2018.

[28] IBM Cloud Education. Recurrent neural networks, 2020. URL https://www.ibm.com/cloud/learn/recurrent-neural-networks. Comprobado en 29-06-2022.

[29] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

[30] Jie Feng, Xueliang Feng, Jiantong Chen, Xianghai Cao, Xiangrong Zhang, Licheng Jiao, and Tao Yu. Generative adversarial networks based on collaborative learning and attention mechanism for hyperspectral image classification. *Remote Sensing*, 12(7):1149, 2020.

[31] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-image translation with conditional adversarial networks. *CoRR*, abs/1611.07004, 2016. URL http://arxiv.org/abs/1611.07004.

[32] Sid Sharma. Ai can see clearly now: Gans take the jitters out of video calls. *The Official NVIDIA Blog*, 2020.

[33] David Charte, Francisco Charte, Salvador García, María J del Jesus, and Francisco Herrera. A practical tutorial on autoencoders for nonlinear feature fusion: Taxonomy, models, software and guidelines. *Information Fusion*, 44:78–96, 2018.

[34] Weifeng Liu, Puskal P Pokharel, and Jose C Principe. Correntropy: A localized similarity measure. In *The 2006 IEEE international joint conference on neural network proceedings*, pages 4919–4924. IEEE, 2006.

[35] Alex Bäuerle and Timo Ropinski. Net2vis: Transforming deep convolutional networks into publication-ready visualizations. *ArXiv*, abs/1902.04394, 2019.

[36] Chanakya Malireddy, Tirth Maniar, and Manish Shrivastava. Scar: Sentence compression using autoencoders for reconstruction. In *ACL*, 2020.

[37] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[38] Weideng Wei, Gai Luo, Jianghong Ran, and Jing Li. Zilong: A tool to identify empty images in camera-trap data. *Ecol. Informatics*, 55, 2020.

[39] O Rebecca Vincent, Olusegun Folorunso, et al. A descriptive algorithm for sobel image edge detection. In *Proceedings of informing science & IT education conference (InSITE)*, volume 40, pages 97–107, 2009.

[40] Michael A. Tabak, Mohammad Sadegh Norouzzadeh, David W. Wolfson, Steven J. Sweeney, Kurt C. Vercauteren, Nathan P. Snow, Joseph M. Halseth, Paul A. Di Salvo, Jesse S. Lewis, Michael White, Ben S. Teton, James C. Beasley, Peter E. Schlichting, Raoul Keith Boughton, Bethany Wight, Eric S. Newkirk, Jacob S. Ivan, Eric A. Odell, Ryan K. Brook, Paul M. Lukacs, Anna K. Moeller, Elizabeth G. Mandeville, Jeff Clune, and Ryan S. Miller. Machine learning to classify animal species in camera trap images: Applications in ecology. *Methods in Ecology and Evolution*, 2018.

[41] Jesús Enrique Cartas Rascón. Detección e identificación de presencia de animales en imágenes de fototrampeo, 2020.

[42] Adarsh Verma. Most popular programming languages for machine learning and data science, 2016. URL https://cutt.ly/SKJ24Nu. Comprobado en 29-06-2022.

[43] Orhan G. Yalcin. Top 5 deep learning frameworks to watch in 2021 and why tensorflow, 2021. URL https://cutt.ly/pKSVDWC. Comprobado en 29-06-2022.

[44] Google Trends. Comparison between tensorflow and pytorch, 2022. URL https://trends.google.com/trends/explore?date=today%205-y&q=%2Fg%2F11bwp1s2k3,%2Fg%2F11gd3905v1. Comprobado en 29-06-2022.

[45] Zhaomin Chen, Chai Kiat Yeo, Bu Sung Lee, and Chiew Tong Lau. Autoencoder-based network anomaly detection. In *2018 Wireless Telecommunications Symposium (WTS)*, pages 1–5, 2018. doi: 10.1109/WTS.2018.8363930.

[46] Francisco J Pulgar, Francisco Charte, Antonio J Rivera, and Maria J del Jesus. Choosing the proper autoencoder for feature fusion based on data complexity and classifiers: Analysis, tips and guidelines. *Information Fusion*, 54:44–60, 2020.

[47] Tapas Kanungo, David M Mount, Nathan S Netanyahu, Christine D Piatko, Ruth Silverman, and Angela Y Wu. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE transactions on pattern analysis and machine intelligence*, 24(7):881–892, 2002.

[48] Stuart Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137, 1982.

[49] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[50] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit*. O'Reilly Media, Inc., 2009.

[51] David Charte, Francisco Herrera, and Francisco Charte. Ruta: Implementations of neural autoencoders in R. *Knowledge-Based Systems*, 174:4–8, 2019.

[52] Paul Gavrikov. visualkeras. https://github.com/paulgavrikov/visualkeras, 2020.