Grado en Ingeniería Informática

Escuela Politécnica Superior de Jaén

Universidad de Jaén

Universidad de Jaén
Escuela Politécnica Superior de Jaén

Trabajo Fin de Grado

DETECCIÓN E IDENTIFICACIÓN DE PRESENCIA DE ANIMALES EN IMÁGENES DE FOTOTRAMPEO

Alumno: Jesús Enrique Cartas Rascón

Tutor: Prof. D. Francisco Charte Ojeda
Prof. D. Antonio Jesús Rivera Rivas

Dpto: Informática

Junio, 2020

Universidad de Jaén

Don Francisco Charte Ojeda y Don Antonio Jesús Rivera Rivas, tutores del Proyecto Fin de Carrera titulado: **Detección e identificación de presencia de animales en imágenes de fototrampeo**, que presenta Jesús Enrique Cartas Rascón, autorizan su presentación para defensa y evaluación en la Escuela Politécnica Superior de Jaén.

Jaén, junio de 2020

CARTAS RASCON JESUS ENRIQUE - 77379601B

Digitally signed by CARTAS RASCON JESUS ENRIQUE - 77379601B
Date: 2020.06.07 13:49:02 +02'00'

RIVERA RIVAS ANTONIO JESUS - 26015863B

Firmado digitalmente por RIVERA RIVAS ANTONIO JESUS - 26015863B
Nombre de reconocimiento (DN): c=ES, serialNumber=26015863B, sn=RIVERA RIVAS, givenName=ANTONIO JESUS, cn=RIVERA RIVAS ANTONIO JESUS - 26015863B
Fecha: 2020.06.08 09:28:36 +02'00'

Agradecimientos

Agradecer enormemente al profesor Francisco Charte Ojeda por su magnífica, motivadora e impecable forma de explicar los conceptos y formar una parte tan importante de este proyecto. Sus magistrales charlas nunca caerán en el olvido y sus tan entrañables trato y maneras permanecerán conmigo por siempre.

Agradecer a Alejandro de la Cruz López los tan acertados comentarios e ideas que han acabado siendo piezas clave en este proyecto, que de otra forma no hubieran sido consideradas, y por su gran apoyo moral y emocional durante todo este periodo.

Agradecer a Rosa Ana Salazar García su incansable apoyo, incondicional cariño y formales consejos que han contribuido a que este trabajo sea mejor, y, más importante, a que yo lo sea.

Por último, pero no por ello menos importante, agradecérselo todo a mi madre, que probablemente sea capaz de defender este trabajo mejor que nadie de tantas veces que me he apoyado en ella para reconstruir mis ideas y reformar mi mente, y es la que me ha ayudado a continuar para poder llegar hasta aquí.

Índice general

1. Introducción **11**

1.1. Cámaras de fototrampeo 11

1.1.1. Ventajas 12

1.1.2. Desventajas 13

1.2. Motivación 13

1.3. Objetivos 14

1.3.1. Objetivo general 15

1.3.2. Objetivos específicos 15

1.4. Estructura de la memoria 16

1.5. Glosario de términos 18

2. Fundamentos de las técnicas a utilizar **19**

2.1. Fundamentos del ML y DL 19

2.1.1. Situación en el espectro del conocimiento 19

2.1.2. Historia: origen, evolución y aprendizaje 20

2.1.3. Fundamentos: perceptrón y capas 23

2.1.4. Términos, parámetros y técnicas 24

2.2. Visión artificial 32

2.2.1. Imágenes binarias 32

2.2.2. Operaciones en imágenes binarias 33

2.2.3. Ecualización de histogramas 37

3. Estado del arte 41

3.1. Deep learning: arquitecturas más conocidas 41

3.1.1. Feed Forward and Deep Feed Forward 41

3.1.2. Convolutional Neural Network (CNN) 44

3.1.3. Auto Encoder (AE) 47

3.1.4. Recurrent Neural Networks (RNN) 48

3.1.5. Generative Adversarial Network (GAN) 49

3.2. Computer Vision 50

3.2.1. Background Subtraction 50

3.2.2. Image Segmentation & Blob Detection 51

3.3. Conclusiones y propuestas 52

4. Herramientas 54

4.1. Herramientas disponibles 54

4.1.1. Lenguajes de programación 54

4.1.2. Técnicas y librerías 56

4.2. Herramienta elegida 57

5. Preparación de los datos 58

5.1. Descarga y obtención de imágenes 58

5.2. Primeros pasos 58

5.2.1. Gestión y organización de las imágenes 59

5.3. Preprocesamiento de las imágenes 60

5.3.1. Procesamiento en paralelo 61

5.3.2. Serialización 61

6. Diseño de los modelos **62**

6.1. Primera configuración: Baseline 63

6.1.1. Análisis de resultados 63

6.2. Segunda configuración: Función de salida lineal 65

6.2.1. Análisis de los resultados 66

6.3. Tercera configuración: Escala de grises 66

6.3.1. Análisis de resultados 68

6.4. Cuarta configuración: Matriz de convolución traspuesta 69

6.4.1. Análisis de resultados 69

6.4.2. Conclusiones 70

6.5. Quinta iteración: Aumento de magnitud 71

6.5.1. Análisis de resultados 72

6.6. Última iteración: Función de pérdida 72

6.6.1. Análisis de los resultados 73

6.7. Modelo basado en CV 74

6.7.1. Flujo de trabajo 75

7. Interpretación y evaluación de resultados **76**

7.1. Resultados del modelo de DL 76

7.1.1. Preparación de los nuevos datos 76

7.1.2. Comparativa de imágenes 77

7.1.3. Selección de un umbral 79

7.1.4. Flujo final de procesamiento 84

7.1.5. Resultados 85

7.2. Resultados del algoritmo de CV 85

7.2.1. Resultados 85

7.2.2. Observaciones y conclusiones 87

8. Conclusiones, comparativa y propuestas 89

8.1. Propuestas de futuro 90

8.1.1. Desarrollo de software 90

8.1.2. Implementación en las cámaras 91

A. Planificación temporal 92

B. Ficheros de código 93

Bibliografía 104

Índice de figuras

1.1. Ejemplo de una cámara de fototrampeo 12

2.1. Diagrama ilustrativo de la jerarquía presente en la AI, ML y DL 20

2.2. Ejemplo de un perceptrón [Elaboración propia] 23

2.3. Ejemplo de diferentes tasas de aprendizaje y su comportamiento [Elaboración propia] 26

2.4. Función linear 27

2.5. Función sigmoide 28

2.6. Función ReLU 28

2.7. Función Leaky ReLU 28

2.8. Ejemplo de imagen binarizada utilizando el método Otsu. 34

2.9. Ejemplo de imagen binarizada utilizando el método de balanceo de histograma. 35

2.10. Ejemplo de dos elementos estructurantes provistos por la librería OpenCV, con tamaño de 5 píxeles. Esta es la punta de rotulador que vamos pasando a lo largo de la imagen para modificarla. 36

2.11. Ejemplo del efecto de erosión en una imagen con muchos detalles y ruido innecesarios. Si el objeto que queremos conservar está bien definido en la imagen original, pasará todos los filtros que le apliquemos a la imagen. 36

2.12. Ejemplo del efecto de ecualización del histograma de una imagen. Se puede apreciar gran diferencia en la distribución del histograma, así como en el aumento de contraste de la imagen. 38

2.13. Ejemplo de la interpolación efectuada en los diferentes casos. 38

2.14. Evolución del contraste de la imagen en función de diferentes tamaños de cuadrícula. 40

3.1. Ejemplo de la arquitectura *Feed Forward* [Elaboración propia] 42

3.2. Ejemplo del proceso de convolución en una imagen añadiendo padding [Elaboración propia] 46

3.3. Ejemplo de algunos de los tipos de Pooling [Elaboración propia] 46

3.4. Arquitectura clásica de una Convolutional Neural Network [Elaboración propia] 47

3.5. Ejemplo de la arquitectura de un autoencoder [Elaboración propia] . . . 48

3.6. Ejemplo de la arquitectura básica de una red neuronal recurrente 49

3.7. Ejemplo del funcionamiento y resultado del algoritmo Background Subtraction 51

5.1. Distribución de imágenes por localización 59

6.1. Visualización del primer modelo 63

6.2. Resumen de los resultados de nuestro primer modelo 64

6.3. Resumen de los resultados de nuestro modelo, con función de activación lineal 65

6.4. Visualización del modelo usando escala de grises 66

6.5. Resultados del modelo original trabajando en escala de grises 67

6.6. Resultados del modelo linear trabajando en escala de grises 67

6.7. Visualización del modelo con capa `Conv2DTranspose` 69

6.8. Resultados del modelo usando `Conv2DTranspose` 70

6.9. Visualización del último modelo 71

6.10. Resultados del modelo con capas de convolución de mayor magnitud . 71

6.11. Resultados del modelo final compilado con MSE 74

6.12. Diagrama del flujo de trabajo 75

7.1. Ejemplo de los resultados obtenidos con una imagen vacía y una con un ciervo. Se puede apreciar cómo en la reconstrucción con el ciervo presente no se distingue nada, ni el ciervo ni la escena original. 78

7.2. Resumen de los histogramas del MSE 80

7.3. Resumen de los histogramas del SSIM 80

7.4. Resumen de los diagramas de caja y bigotes 81

7.5. Ejemplo de una fotografía vacía por la noche 82

7.6. Ejemplo de los resultados obtenidos y sus correspondientes índices. . . 83

7.7. Gráfico del flujo final del proyecto [Elaboración propia] 84

7.8. Evolución de la precisión en función del umbral 86

7.9. Diferentes matrices de confusión en función del umbral 88

8.1. Placa PCIe Coral Edge TPU 91

A.1. Estimación temporal del proyecto. 92

Índice de cuadros

1.1. Glosario de todos los términos utilizados en el documento 18

5.1. Información de una de las fotografías del dataset 60

5.2. Información de una de las fotografías del dataset, procesada 61

6.1. Evolución del primer modelo en distintas iteraciones 63

6.2. Evolución del segundo modelo en distintas iteraciones 66

6.3. Evolución del modelo monocromo a la izquierda y evolución del modelo monocromo lineal a la derecha 68

6.4. Evolución del cuarto modelo en distintas iteraciones 70

6.5. Evolución del quinto modelo en distintas iteraciones 72

6.6. Evolución del último modelo en distintas iteraciones 74

7.1. Ejemplo de 5 muestras de los datos obtenidos para nuestro conjunto de imágenes mixto. Se ejecutaron las predicciones en el modelo y, con las imágenes de salida, se calcularon los factores de MSE y SSIM. 79

7.2. Muestra de evolución de la precisión con respecto al límite de niveles en el conjunto de evaluación. Se aprecia que el mejor resultado se encuentra en 5 niveles. 85

Índice de listados

1. Ejemplo de la estructura del archivo JSON, usando el estándar COCO [1] 94
2. Implementación del autoencoder sacado de la página de Keras. 95
3. Implementación del autoencoder, ligeramente modificado para nuestro problema. 96
4. Implementación del autoencoder con función de activación linear. . . . 97
5. Implementación del autoencoder en escala de grises. La especificación de la unidimensionalidad se hace en la *input shape*, al inicio. 98
6. Implementación del autoencoder con el uso de la capa `Conv2DTranspose`. 99
7. Implementación del autoencoder en su versión final, compilado con MSE. 100

Capítulo 1

Introducción

Para empezar, una breve introducción con el fin de establecer el marco contextual en el que vamos a trabajar.

El estudio de la fauna y la flora de un determinado lugar es una actividad de carácter naturalmente costoso. Un grupo de investigadores debe acceder al lugar, inspeccionar aquellos seres en los que estén interesados y repetir la tarea de forma periódica para observar cambios en las especies y poder hacer un seguimiento de las mismas.

Esta tarea se vuelve particularmente difícil para animales muy evasivos, que huirán rápidamente de la presencia de cualquier ser humano. Su estudio es igual, si no más importante que la de cualquier otro animal y de esta forma se convierte en una tarea inviable.

1.1. Cámaras de fototrampeo

La solución que se propone es la siguiente: *cámaras de fototrampeo* [2]. Estos dispositivos permiten obtener imágenes de la fauna y flora que las rodea de forma automática y no intrusiva para dichos seres. Estas fotografías se toman con la idea de poder estudiar su comportamiento y evolución de una forma más cómoda para ambas partes, investigadores y animales.

Las cámaras pueden funcionar de diferentes maneras según su arquitectura y tecnología, pero el factor común se resume a tomar una foto cuando se detecta movimiento en su campo de visión. Al estar ocultas entre la maleza del lugar, los animales a los que antes no se tenía opción de ver en directo son mucho más accesibles ahora.

Figura 1.1: Ejemplo de una cámara de fototrampeo

1.1.1. Ventajas

Las principales ventajas que resuelven las cámaras de fototrampeo es el estudio no invasivo de animales, como se mencionaba anteriormente. Una cámara puede situarse en cualquier lugar, véase la figura 1.1, ya sea en un árbol u oculta entre la maleza. Son, evidentemente, muy resistentes a las inclemencias del tiempo y pueden aguantar mucho tiempo monitorizando la actividad de dichos animales que sean más inaccesibles, ya sea por su horario de actividad o por su naturaleza evasiva.

Existen también modelos que, mediante tecnologías basadas en el *Internet of Things* (IoT), están conectados a una red y pueden enviar la información directamente, en tiempo real. Las hay incluso especializadas en *streaming* de vídeo, para monitorizar continuamente un área concreta. Esto permite que todo el trabajo humano que tendría que invertirse en semejante tarea, pueda delegarse a estos dispositivos.

1.1.2. Desventajas

Sin embargo, la toma de fotografías automática conlleva también grandes problemas: en todas las imágenes no aparecerán animales, y en las fotografías en las que sí aparezca un animal será difícil de buscar, si tratamos de estudiar uno en concreto.

Esto ocurre cuando, por ejemplo, un animal pasa demasiado rápido delante de la cámara, no siendo el tiempo de respuesta de la misma suficientemente bajo como para captarlo. A veces, la caída de una rama o el movimiento algo más exagerado de la flora debido al viento u otros factores pueden activar el mecanismo de igual forma. Las condiciones atmosféricas pueden también provocar que, aun la cámara tomando una fotografía de un animal tal y como es esperado, la calidad de la imagen se deteriore hasta el punto de ser inutilizable, como que la lente frontal esté húmeda y la imagen resulte borrosa, o la luz de ambiente sea tan alta o baja que la cámara no pueda compensar los parámetros de exposición por cualquier razón.

Además, también se han dado casos de imágenes corruptas a causa del sistema de guardado de la cámara. En resumen, las razones por las que una imagen puede no ser utilizable son muchas, y la línea entre una fotografía útil y una inútil se vuelve muy difusa.

Esto implica una tremenda cantidad de información que procesar, de la cual realmente el porcentaje útil es muy bajo. Este problema se satura considerablemente cuando tenemos varias unidades capturando y de forma relativamente frecuente. Hablamos de varios órdenes de magnitud de imágenes que hay que gestionar.

1.2. Motivación

> *Learning, like intelligence, covers such a broad range of processes that it is difficult to define precisely. A dictionary definition includes phrases such as "to gain knowledge, or understanding of, or skill in, by study, instruction, or experience," and "modification of a behavioral tendency by experience." Zoologists and psychologists study learning in animals and humans. In this book we focus on learning in machines.*

Nils Nilsson comenta esto en su libro *Introduction to machine learning* [3]. Se define inteligencia como la «obtención del conocimiento, la comprensión o la habilidad adquirida por estudio, enseñanza o experiencia» o «cambio en las tendencias del comportamiento por experiencia». Esto se puede aplicar a personas, a animales, o incluso a máquinas.

El aprendizaje automático o *machine learning* (ML) es una disciplina que, si bien está teniendo actualmente un gran auge debido a la cantidad de nuevos recursos y

tecnologías que nacen del imparable avance en investigación hardware, lleva mucho tiempo con nosotros. Ya en los años 50 se descubrieron los métodos estadísticos que darían lugar a la base sobre la que se construirían los primeros modelos con algoritmos simples.

El principal motor de este proyecto radica en el uso de las nuevas tecnologías a nuestro favor para convertir una tarea tan ardua y tediosa como es la clasificación e identificación de animales en una tarea cómodamente automática. Para estándares de hoy día, es completamente viable crear un sistema que clasifique, ordene y etiquete los grandes volúmenes de información que se tratan en disciplinas como esta, con una tasa de acierto suficientemente alta como para que todo ello merezca la pena.

Delegar esta tarea a personas es una opción poco eficiente, ya que los humanos, si bien somos precisos, pecamos de ser lentos. Las personas que se dedicaran a esto podrían estar investigando otros temas mucho más importantes. El ahorro de recursos humanos y aceleración y aumento de la eficiencia del proceso es otro gran motivo para acometer esta empresa. Es, aún así, destacable que muchos de los sistemas automáticos bien ajustados han llegado a superar la tasa de acierto del ojo humano, sobre todo en tareas de identificación de problemas médicos como tumores.

Además, la utilización de este tipo de sistemas permite una mejor organización y gestión de grandes conjuntos de imágenes, ya que las categorías se pueden generar automáticamente y esto implica una mejor indexación y, consecuentemente, acceso a diferentes imágenes cuya información requerimos en cierto momento.

Resumiendo y contextualizando: un gran porcentaje de las imágenes obtenidas, a consecuencia de los problemas enumerados anteriormente finalmente no corresponden a la imagen de un animal, sino que son imágenes vacías. Separar estas de aquellas en las que sí aparecen animales y que son objeto de interés real es una tarea muy tediosa y que, hasta el momento, se ha efectuado manualmente por parte de profesionales que podrían estar centrando sus esfuerzos en tareas más importantes. En las siguientes secciones se detallarán los objetivos planteados para intentar solucionar este problema.

1.3. Objetivos

En esta sección, describiremos a nivel general los objetivos que planteamos superar con este proyecto, y, más específicamente, en qué consiste cada uno, dividiéndolos en tareas fáciles de conceptualizar y completar.

1.3.1. Objetivo general

El objetivo de este proyecto, en última instancia, es desarrollar un sistema software que permita clasificar aquellas imágenes que posean animales de algún tipo. Automatizar este proceso informáticamente permite acometerlo de forma práctica y eficiente, para que el análisis y estudio de la evolución de estos animales pueda ser lo más cómodo y preciso posible.

1.3.2. Objetivos específicos

Más concretamente, los objetivos a tratar se resumen en:

Ciencia Datos

Obtención de una visión general del campo de Ciencia de datos y los métodos de Minería de datos esenciales, como la obtención del conocimiento y la gestión de grandes cantidades de información.

Creación de un dataset

- Recopilación de un conjunto de imágenes de fototrampeo de una o más localizaciones y etiquetado manual de aquellas en las que aparecen animales y su especie.
- Preprocesamiento pertinente de las imágenes para que se dispongan en un formato apropiado para las tecnologías que estemos usando.
- Serialización y gestión de las imágenes para su posible uso posteriormente.

Experimentación

- Se evaluarán diferentes metodologías y algoritmos para extraer el conocimiento de las imágenes y poder inferir si un animal está presente de forma automática.
- Se utilizarán técnicas de *Computer Vision* (CV) para el análisis y clasificación de las imágenes, probando con diferentes algoritmos.
- Se indagará en el uso de redes neuronales para el análisis y clasificación de las imágenes, probando varias configuraciones y modelos.

Comparativa y Evaluación

- Se compararán ambas técnicas en cuanto a eficacia y eficiencia en distintas dimensiones: tiempo de uso, dificultad de implementación, consumo de recursos, etc.

- Se evaluará cada sistema y se dará una descripción detallada de las ventajas e inconvenientes que cada uno posee, así como una conclusión de cuál es más apropiado considerando diferentes escenarios.

1.4. Estructura de la memoria

Finalmente, en esta sección se resumirá brevemente el contenido de los capítulos de la presenta memoria, a fin de que el lector pueda obtener una visión general sobre el contenido de cada uno de ellos.

Capítulo 1. Introducción

Capítulo en el que nos encontramos, donde proporcionamos el contexto y la información previa necesarias para comprender los objetivos y motivos de la realización de este proyecto.

Capítulo 2. Fundamentos del aprendizaje automático

En este capítulo se hará un breve repaso por los temas más teóricos y conceptuales del aprendizaje automático y profundo, así como la exposición de algunos de los puntos más básicos del mismo a nivel práctico.

Capítulo 3. Estado del arte

Aquí se revisarán varias de las técnicas más relevantes actualmente disponibles que nos ayuden a completar nuestro objetivo, así como una breve explicación y justificación de las mismas.

Capítulo 4. Herramientas

En el capítulo 4 expondremos varias de las herramientas que hay disponibles para acometer nuestro proyecto, sus ventajas y desventajas en el contexto del conocimiento previo adquirido y la justificación de la elección final.

Capítulo 5. Preparación de los datos

Capítulo dedicado a la primera fase de nuestro flujo de trabajo: la preparación de los datos. De dónde y cómo se obtuvieron, cómo se procesaron y cómo se almacenaron.

Capítulo 6. Diseño de los modelos

El capítulo 6 entra de lleno en el diseño de los diferentes métodos que vamos a utilizar. Se sigue un proceso iterativo de ajuste de parámetros para ver la evolución y mejora de los resultados obtenidos, y se exponen los motivos de las elecciones hechas.

Capítulo 7. Interpretación y evaluación de los resultados

Una vez obtenido un modelo, pasamos a evaluar cómo podemos utilizarlo y cómo de bien funciona. En este capítulo se analizan diferentes maneras de utilizar los datos para ver cuál es la mejor.

Capítulo 8. Conclusiones, comparativa y propuestas

En este capítulo haremos un breve repaso por todo el proceso de inicio a fin evaluando las diferentes posibilidades que podrían haberse tomado. Además, se propondrán algunos esquemas para implementar este proyecto en sistemas listos para producción.

Anexo A. Ficheros de código

En este anexo incluimos aquellos módulos de código y archivos relevantes para que el lector pueda hacerse una idea más tangible de cómo funcionan y se construyen estos sistemas realmente.

1.5. Glosario de términos

Se facilita aquí un resumen de los términos más utilizados en su versión en español, inglés y sus siglas en inglés, para facilitar la lectura y ahorrar posibles confusiones. Se omitirán aquellos que se utilicen solo una vez en el cuerpo de texto, ya que su definición estará presente en el momento en el que aparezcan. La tabla está ordenada en orden alfabético para el término en español.

Español	**Inglés**	**Siglas en inglés**
Aprendizaje automático	Machine Learning	ML
Aprendizaje profundo	Deep Learning	DL
Ecualización de histograma adaptativa	Adaptive Histogram Equalization	AHE
Ecualización de histograma adaptativa con límite de contraste	Contrast Limited Adaptive Histogram Equalization	CLAHE
Entropía Cruzada Binaria	Binary Cross Entropy	BCE
Error Medio Cuadrático	Mean Squared Error	MSE
Índice de similitud estructural	Structure Similarity Index	SSIM
Inteligencia artificial	Artificial Intelligence	AI
Perceptrón multicapa	Multilayer Perceptron	MLP
Red generativa antagónica	Generative Adversarial Network	GAN
Red neuronal artificial	Artificial Neural Network	ANN
Red neuronal convolucional	Convolutional Neural Network	CNN
Red neuronal recurrente	Recurrent Neural Network	RNN
Visión artificial	Computer Vision	CV
-	Autoencoder	AE

Cuadro 1.1: Glosario de todos los términos utilizados en el documento

Capítulo 2

Fundamentos de las técnicas a utilizar

Dedicaremos este capítulo a introducir brevemente el núcleo de los procedimientos más importantes que vamos a utilizar en nuestro proyecto: DL y CV.

Con respecto al DL, comenzaremos por su situación en el espectro del conocimiento de la informática y la ciencia, siguiendo con los orígenes, historia y avances a lo largo del tiempo, y finalmente expondremos varios de los factores más fundamentales y terminología para poder entender el proyecto en su totalidad lo mejor posible.

Posteriormente, haremos una breve pero completa revisión de los algoritmos más útiles que pondremos en uso para detectar la presencia de animales de forma automática.

2.1. Fundamentos del ML y DL

En esta sección se expondrán todos los datos necesarios para obtener unas nociones básicas de ML y DL, desde su historia y conceptos que inspiraron las técnicas actuales hasta su funcionamiento y piezas fundamentales.

2.1.1. Situación en el espectro del conocimiento

La inteligencia artificial o *artificial intelligence* (AI) es una rama del conocimiento de la informática que se especializa en la imitación de comportamientos inteligentes mediante cualquier tipo de recurso, ya sea más o menos rudimentario. Esto abarca desde cláusulas condicionales o árboles simples de decisión [4] hasta complejos sistemas capaces de reconocer imágenes, o incluso crearlas.

Dentro de esta categoría tenemos el ML, que estudia métodos generalmente estadísticos que tratan de aprender de la experiencia. Muchos de los algoritmos en este subconjunto ya incluyen una fase de entrenamiento para dar una respuesta, además de otros factores que estudiaremos en la subsección 2.1.3.

El DL se incluye dentro de esta última subcategoría, como se ilustra en la figura 2.1, definiéndose como todo el conjunto de algoritmos y técnicas que son expuestos a enormes cantidades de datos para su aprendizaje. La capacidad de resolución de problemas de altísima dificultad de esta última categoría abre un nuevo mundo de posibilidades, siendo posible el reconocimiento, clasificación o análisis de imágenes, texto, discurso o cualquier otro tipo de dato muy complejo, capacidad de la que las demás técnicas carecen.

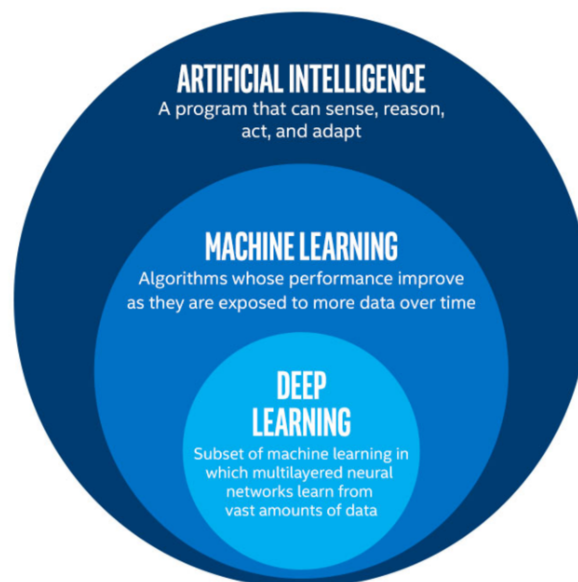


Figura 2.1: Diagrama ilustrativo de la jerarquía presente en la AI, ML y DL [5]

2.1.2. Historia: origen, evolución y aprendizaje

Gran parte de esta sección está inspirada en el magnífico trabajo de Haohan Wang y Bhiksha Raj, On the origin of Deep Learning [6] y las referencias que en dicho trabajo se hacen. En esta sección nos centraremos brevemente en conocer aquellas disciplinas y pensamientos que inspiraron estas tecnologías a ser como hoy las conocemos.

El nacimiento del DL como tal podría datarse, a priori y sin pensar mucho, del primer grupo de investigadores que lograran desarrollar un sistema primitivo pero, en esencia, similar a los que usemos hoy. Sin embargo, esto estaría dando por hecho miles de trabajos que se consiguieron antes de esto, que sirvieron como inspiración para llegar a dicho modelo.

El nombre de la disciplina se traduce con el nombre de *aprendizaje profundo*. En el mismo nombre aparece un área de estudio que podría no haberse considerado: la psicología. Además, muchos de los avances en este campo heredan gran parte de su contenido de la filosofía. Vamos a ver algunos de los más importantes para nuestro contexto, junto con el pequeño (o gran) granito de arena que aportasen.

Aristóteles, 300 a.C

Ya en el 300 a.C, Aristóteles nos dio algunas pistas muy esenciales de cómo debemos pensar para resolver el problema de conceptualizar el pensamiento. Introdujo el concepto de *asociacionismo*, que decía que la mente es un conjunto de elementos conceptuales que están organizados como asociaciones entre estos mismos elementos. Este fue el primer intento en la historia del ser humano para entender el conocimiento, el cerebro y las relaciones que hay entre ellos.

Aristóteles describía este proceso como *sentido común*. Por ejemplo, el olor o sabor de una manzana naturalmente nos llevan a concebir una manzana, es sentido común. Estas afirmaciones, aunque profundamente simples, son realmente los cimientos sobre los que se sustentan muchos de los algoritmos mencionados: muestras que están cercanas a otras (dada una distancia máxima) se agrupan conjuntamente, o eventos que ocurren consecutivamente de forma muy frecuente son asociados. Es lógico, es sentido común. Estamos tratando de *incluir* sentido común en nuestros algoritmos.

Alexander Bain, 1873

El filósofo y psicólogo Alexander Bain, en su trabajo Mind and body. The theories of their relation [7] ya hablaba de relaciones entre lo que denominaba *células*:

> *For every act of memory, every exercise of bodily aptitude, every habit, recollection, train of ideas, there is a specific grouping, or co-ordination, of sensations and movements, by virtue of specific growths in the cell junctions*

«Para cada acto de memoria, cada movimiento de cuerpo, hábito, pensamiento, existe una agrupación específica, o coordinación, de sensaciones y movimientos, causados por el crecimiento de uniones específicas entre neuronas.»

Esta visión ya se acerca mucho al sistema actual de neuronas interconectadas. Este trabajo sirvió de base para la teoría hebbiana del aprendizaje: *hebbian learning*.

Donald Hebb, 1949

Donald Hebb es considerado como el padre de las redes neuronales tal y como hoy en día las conocemos. En su libro *The Organization of Behavior* [8], utiliza y extiende el modelo presentado por Bain, añadiendo el factor de conexión entre las células.

> *When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased.*

«Cuando un axón de una neurona[1] A está lo suficientemente cerca como para excitar a una neurona B y repetida o persistentemente lo hace, un proceso metabólico se activa en alguna o ambas neuronas de forma que la eficiencia con la que A activa a la neurona B aumenta.»

Este es el modelo de aprendizaje antes mencionado: *hebbian learning*. En esencia, cuando dos células se activan a la vez, se conectan. La fuerza de esta conexión será proporcional al número de veces que esto ocurra.

Esta definición ya es realmente similar al esquema de red neuronal que conocemos. Se habla de las neuronas, se habla de sus conexiones y de los pesos de las conexiones.

Conclusiones y resumen

A partir de aquí, las investigaciones que se hicieron tuvieron un carácter más práctico y directamente relacionado con las redes neuronales modernas. La más destacable ha de ser primer modelo de una neurona construido por Warren McCulloch y Walter Pitts, en su trabajo *A logical calculus nervous activity* [9], creando lo que se conoce como *MCP neural model*.

Sin intención de restarle importancia a las mismas, en esta sección quería centrarme en la teoría y conceptos que subyacen detrás de todos los avances que vinieron después. Si bien, el ámbito está muy relacionado con la matemática y la informática, podemos ver que las referencias aquí mencionadas proceden del campo de la filosofía y la psicología, principalmente.

Como resumen final, es destacable que todos estos autores y trabajos aquí mencionados coinciden y se complementan en varios puntos de una manera muy elegante:

[1] En términos biológicos, una neurona es la célula más pequeña existente en el cerebro capaz de procesar información. Las neuronas se componen de todo lo que se compondría una célula, como núcleo, mitocondria, etc., y el axón es el canal por el que se comunican con las demás neuronas.

- Las diferentes ideas y la mente se conceptualizan como elementos interconectados.
- La fuerza de unión entre dos elementos viene dada por el número de veces que se activen a la vez.
- La frecuencia como elemento clave para el aprendizaje. Cuantas más veces ocurra algo, mejor se aprenderá.

Se pueden entrever de forma bastante evidente varios de los temas que se tratan a la hora de entrenar una red neuronal: construir un modelo con capas de lo que denominamos *neuronas*, interconectar dichas neuronas, entrenar un número de veces el modelo (*epochs*) para que aprenda, por solo nombrar algunas. Introduciremos todos estos parámetros y sus definiciones junto con otros igual de importantes más adelante.

2.1.3. Fundamentos: perceptrón y capas

El perceptrón es la mínima unidad de conocimiento que contemplamos en esta disciplina. Un perceptrón admite unos valores de entrada x_i ponderados por unos pesos w_i y, mediante una función de activación $f(x)$ produce una salida si se supera un umbral θ.

El *MCP neural model* mencionado anteriormente define matemáticamente esta estructura, mostrada gráficamente en la figura 2.2, de la siguiente manera:

$$y = \begin{cases} 1, & \sum_i w_i x_i \ \geq \ \theta \\ 0, & \text{en otro caso} \end{cases} \tag{2.1}$$

donde y es la salida, x_i es la entrada de las señales y w_i es el peso de cada canal de entrada. θ es el umbral de actvación del perceptrón.

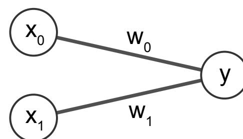


Figura 2.2: Ejemplo de un perceptrón [Elaboración propia]

Un perceptrón es esencialmente una definición de una función lineal de los valores de entrada. Para hacer obtener una capacidad de representación más poderosa, podemos disponer varios perceptrones juntos, formando una capa. Extendiendo el proceso,

se pueden conectar varias capas, formando lo que se denomina *Multilayer Perceptron* (MLP), formándose así la versión más básica de una red neuronal. Aún así, el añadir muchos más perceptrones o capas no necesariamente implicará el aumento de la capacidad de representación de nuestra red.

Llegados a este punto, tenemos una arquitectura que se compone de miles de parámetros que podemos manipular para hacer una red neuronal: número de perceptrones, los pesos de las conexiones entre ellos, las funciones de activación de cada uno, el número de capas, el tipo de capas, etc., entre muchísimos otros. Veremos los más importantes en la siguiente subsección, con objeto de entenderlos mejor.

La gran pregunta siempre está ahí: ¿cuál es la combinación óptima de dichos parámetros? Aunque una respuesta no se podrá dar para todos los casos, sí se discutirán en las siguientes secciones, y, de hecho, dedicaremos toda la sección de experimentación al análisis de los mismos en práctica.

2.1.4. Términos, parámetros y técnicas

Dedicaremos esta sección a la definición y explicación de algunos de los parámetros, términos, funciones y demás elementos pertenecientes a las redes neuronales de forma que aquellas secciones en las que se mencionen de algún modo queden completamente claras.

Tipos de aprendizaje

El ML consta de muchos métodos para simular un proceso de aprendizaje, pero estos métodos se pueden clasificar en tres categorías:

- **Aprendizaje supervisado**: el algoritmo entrena con un conjunto de datos etiquetado. Cada vez que el algoritmo hace una predicción de una muestra en la fase de entrenamiento, se le muestra la respuesta correcta a dicha muestra (la etiqueta correspondiente). El algoritmo entonces actuará en consecuencia de si su respuesta coincide con la etiqueta o no.
- **Aprendizaje no supervisado**: el conjunto de datos no está etiquetado. El objetivo aquí es que el algoritmo encuentre patrones en los datos por sí mismo, trate de agruparlos o resumirlos, ofreciendo perspectivas reveladoras que de otra forma no hubieran podido considerarse.
- **Aprendizaje por refuerzo**: es el tipo de entrenamiento que toma en cuenta como los agentes actuarían en un determinado entorno tratando de maximizar el beneficio acumulado. El aprendizaje por refuerzo difiere del supervisado en no

tener etiquetas para el conjunto de datos, y en no necesitar que las acciones sean explícitamente corregidas durante el entrenamiento. El principal objetivo es encontrar un equilibrio entre la exploración del espacio de búsqueda no conocido y la explotación del conocimiento ya adquirido.

Funcionamiento básico de una red neuronal moderna

Las redes neuronales modernas cuentan con un elevado número de perceptrones distribuidos en capas que ocupan roles muy específicos dentro de la red, con conexiones variantes durante el proceso que garantizan que el aprendizaje sea lo más efectivo y eficiente posible.

Una red neuronal simple con arquitectura *Feed Forward* recibe un conjunto de entradas. La capa de entrada procesa estas señales y las conexiones con la capa siguiente se actualizan. Este proceso se repite hasta llegar a la capa de salida, en la que la red hace una estimación, es decir, ofrece una respuesta para este estímulo de entrada. Esta respuesta se evalúa comparándola con la *verdad absoluta*, es decir, el valor de la etiqueta que manualmente hemos hecho con intención de que la red aprenda. En función del error que se haya cometido, la red reajusta los pesos de forma pertinente. Este ajuste puede hacerse de varias maneras, aunque los algoritmos más comunes son el gradiente descendente y el *backpropagation*. Al final del entrenamiento y si el proceso ha ido bien, la red ha ajustado los pesos de sus conexiones de forma muy específica para poder ofrecer una respuesta correcta en el mayor número de muestras de entrada posible.

Algoritmos de optimización para el cálculo de gradiente

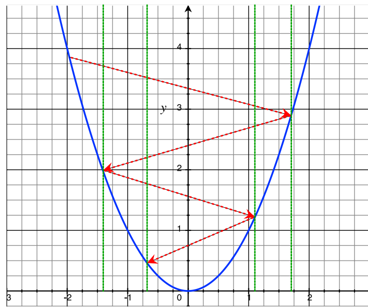
1. **Gradiente descendente**

   Uno de los métodos de optimización más simples de entender y utilizados es el descenso de gradiente [10]. Matemáticamente, es un algoritmo de optimización iterativo para encontrar un mínimo local de una función diferenciable. Para encontrar un mínimo local de una función usando la pendiente de gradiente, tomamos pasos proporcionales al negativo del gradiente de la función en el punto actual.

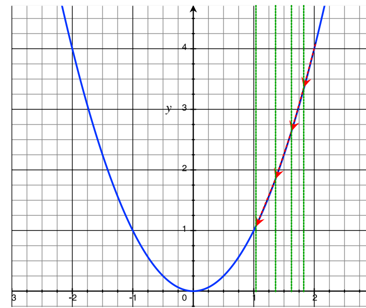
   Esto, en otras palabras, quiere decir que nuestro objetivo es encontrar el mínimo o máximo de una función, dependiendo del problema, y que, para llegar ahí, procederemos iterativamente a analizar en qué punto del espacio de búsqueda estamos y, actuaremos consecuentemente dando el paso más favorable.

   Un factor importante en este algoritmo es la tasa de aprendizaje, o *learning rate*. Es el factor que determina cómo de grande será el siguiente paso que demos en nuestra búsqueda del óptimo de la función.

Si damos pasos muy grandes, podremos llegar muy pronto, pero corremos el riesgo de pasarnos de la raya. Si damos pasos muy pequeños, no nos pasaremos, pero tardaremos mucho más. Ser capaces de establecer un valor variable es una de las mejores estrategias, en las que primero se den pasos grandes para explorar y progresivamente se vayan haciendo más pequeños conforme nos acerquemos al punto de convergencia.



(a) Descenso del gradiente con una tasa de aprendizaje demasiado alta



(b) Descenso del gradiente con una tasa de aprendizaje demasiado baja

Figura 2.3: Ejemplo de diferentes tasas de aprendizaje y su comportamiento [Elaboración propia]

2. ***Backpropagation***

   Backpropagation [11] es un algoritmo propuesto para calcular el gradiente que anteriormente se menciona. La idea es averiguar cuál es el coste medio de la red en cada iteración[2] y ver cómo afecta cada neurona de la salida a ese coste. En función de ello, podemos propagar hacia atrás el error medido en la capa de salida con respecto a la etiqueta correspondiente, ajustando los pesos de forma que se minimice la función de coste para esta iteración.

   En esencia, al generarse la salida estimada y compararse con la etiqueta, calculamos cómo de mal se ha hecho. Este cálculo se hace para cada perceptrón de la capa de salida y medimos cuantitativamente cómo de lejos estamos de la respuesta correcta, midiendo el cuadrado de la diferencia de la respuesta con la etiqueta. Una vez calculado el error, podemos propagar hacia atrás este error, de forma que las capas intermedias son capaces de ver qué valor tendrían que haber tenido sus pesos para poder proveer la respuesta correcta.

   En este proceso es muy importante la derivación de las funciones de activación. Si una función de activación no es derivable, no se puede hacer uso de esta técnica.

Estas dos técnicas trabajan en conjunción para que el proceso de aprendizaje converja rápidamente. *Backpropagation* calcula efectivamente el error local en cada itera-

[2]Una iteración se completa al evaluar todo el conjunto de entrenamiento una vez. El entrenamiento acaba cuando se cumplen todas las iteraciones establecidas por el usuario, o alguna función de *callback* lo finaliza antes.

ción y el gradiente descendente nos ofrece la respuesta a cómo debemos actuar para movernos en la dirección correcta.

Funciones de activación

Anteriormente mencionamos el valor θ como umbral de activación del perceptrón. Si la suma de los pesos del perceptrón superaba el umbral, dicho perceptrón se activaba. Este umbral no es un valor fijo, sino que suele implementarse como una función de activación.

Una función de activación recibe la suma de las entradas ponderadas por los pesos que llegan a dicho perceptrón y ofrece una salida, que será la salida del perceptrón. Alterando las funciones de activación podemos alterar el comportamiento de la red y, con un buen dominio de las mismas, optimizarla a su mejor versión.

Una función de activación ha de tener un cálculo computacional eficiente, ya que para cada uno de los perceptrones presentes en la red existirá una función de activación asociada. Además, el uso extendido de la técnica *backpropagation* satura este efecto considerablemente, así como el de la derivada de la función. Si la función de activación presenta una carga considerable, el entrenamiento de la red será muy costoso y difícil.

Un exhaustivo análisis de varias de las funciones de activación más relevantes puede consultarse en *Activation Functions: Comparison of Trends in Practice and Research for Deep Learning* [12].

Veamos algunos ejemplos de las funciones de activación más comunes:

1. **Función lineal o identidad**

   Esta función devuelve una salida proporcional a la entrada. Al ser una función linear, su derivada no depende de las entradas (es constante 1), por lo que no puede usarse en métodos más avanzados como *backpropagation*, que dependen exclusivamente de la derivada de la función de activación.

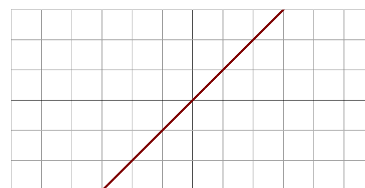


Figura 2.4: Función linear

$$f(x) = x$$

2. **Función logística o sigmoide**

Las principales ventajas es que la función normaliza los valores de salida en el rango $[0, 1]$, pero sufre del *vanishing gradient problem* [13], además de que es computacionalmente costosa.

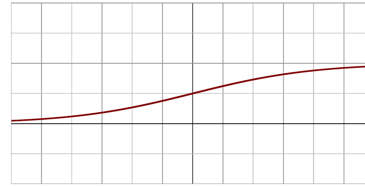


Figura 2.5: Función sigmoide

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

3. ***Rectified Linear Unit* (ReLU)**

Se propone como una solución a muchos de los problemas mencionados. Esta función sí posee derivada dependiente de las entradas, por lo que puede usarse en *backpropagation* y es muy fácil de calcular, pero posee un problema propio, llamado *dying ReLU problem* [14]. Aún así, es una de las funciones de activación más comunes.

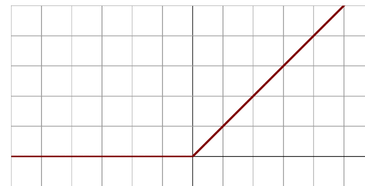


Figura 2.6: Función ReLU

$$f(x) = max(0, x)$$

4. ***Leaky ReLU***

Es un *arreglo* de la ReLU original que trata de solucionar el *dying ReLU problem*. En todos los demás aspectos es igual que su predecesora. Un problema que surge en esta variante es que las predicciones no son consistentes para valores de entrada negativos.

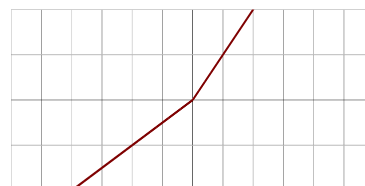


Figura 2.7: Función Leaky ReLU

$$f(x) = max(0{,}01x, x)$$

5. ***Softmax***

No es una función común como las anteriormente expuestas, ya que recibe un vector de valores, en lugar de un valor único. Esta función normaliza todos los valores en el rango $[0, 1]$ y los divide por el sumatorio de todos, lo que ofrece una función de probabilidad de que dicha muestra pertenezca a una clase específica de las provistas. Solo se utiliza en la capa de salida de la red, a fin de generar una distribución de probabilidades para problemas multiclase.

$$f_i(\vec{x}) = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}} \tag{2.2}$$

Tipos de capas

Hoy en día, los diferentes *frameworks* con los que podemos trabajar nos abstraen de muchas partes de todo el proceso de creación, entrenamiento y gestión de información que acaece en el interior de una red neuronal. Si bien es crucial conocer todas las partes de las que se componen para poder decidir con el mejor criterio posible el diseño de la misma, no es necesario tener que implementarlo todo desde cero, ya que ello supone una tarea tediosa, que puede muy fácilmente evitarse. Uno de los lemas más representativos de esto es «No reinventes la rueda», enalteciendo a la utilización de recursos que ya están disponibles y evitando tener que perder tiempo en crearlos nosotros mismos.

Prueba de ello es la construcción modular que proponen la mayoría de los *frameworks* basada en capas. Cada capa tiene un rol específico en función de sus propiedades. Gestionar el diseño en función de capas es mucho más intuitivo. Veamos algunas de las capas más importantes que nos ofrece la librería que hemos utilizado en nuestro proyecto, Keras, aunque estas estarán presentes en prácticamente cualquier otro:

- `Input`: es la capa de entrada. Es muy importante ya que sus dimensiones definirán la estructura de la red. Las dimensiones de esta capa se adecuarán a las dimensiones de nuestro conjunto de datos.
- `Dense`: construye una capa densa con cuantos perceptrones deseemos.
- `Conv2D` y `Conv2DTranspose`: capas que crean y automatizan el proceso de aplicar filtros convolucionales a la imagen para extraer diferentes características. Se extiende en la explicación de este factor en la sección 3.1.2.

- `MaxPooling2D` y `UpSampling2D`: estas capas actúan como módulos para reducir o aumentar las dimensiones de los datos. Se extiende en la explicación de este factor en la sección 3.1.2.
- `Flatten` y `Reshape`: capas para modificar la dimensionalidad de nuestros datos. *Flatten* en inglés es aplanar, lo que convierte un array multidimensional en uno unidimensional. *Reshape* modifica las dimensiones de un array como se desee.

Entrenamiento

Todos los algoritmos mencionados incluyen lo que se conoce como una fase de entrenamiento: proceso en el cual el algoritmo es provisto con una gran cantidad de datos y, dependiendo de su naturaleza, da respuestas a esos datos. El algoritmo, al finalizar el entrenamiento, es capaz de inducir los patrones que subyacen tras los datos, pudiendo inferir respuestas a nuevas muestras.

Esta fase de entrenamiento es muy importante, y por ello debe ser controlada. En muchas ocasiones, si no se gestiona bien este proceso, el algoritmo puede *aprenderse de memoria* las respuestas, con lo cual no estaría generalizando lo suficiente. Esto se manifiesta con unos valores de precisión muy altos, que se ven claramente afectados cuando salimos del conjunto del que hemos entrenado y proveemos otras muestras, que son de las que realmente queremos obtener información. Este fenómeno se denomina sobreajuste, u *overfitting* [15]. Para solucionarlo, dado nuestro conjunto de datos al completo, siempre haremos 3 divisiones: una de entrenamiento, otra de validación y otra de evaluación.

- **Partición de entrenamiento**: subconjunto de datos con los que proveemos al algoritmo para entrenar.
- **Partición de validación**: subconjunto de datos con los que el algoritmo reajusta sus valores al final de cada iteración en la fase de entrenamiento, para garantizar un aprendizaje óptimo.
- **Partición de evaluación**: subconjunto de datos provisto al final del entrenamiento, con objeto de evaluar la precisión real del algoritmo. En esta fase ya no se ajustan parámetros, es una simulación de las predicciones que haría el algoritmo.

Por definición matemática, una partición de un conjunto es una operación en el mismo que lo subdivide en subconjuntos disjuntos, es decir, los elementos de un subconjunto están única y exclusivamente en él, y en ningún otro. Esto es importante, ya que la idea es enfrentar al algoritmo en las fases de validación y evaluación con datos que no ha visto nunca, para poder evitar así el problema de sobreajuste mencionado.

La fase de entrenamiento suele disponer en la mayoría de *frameworks* de un gran número de parámetros[3] que podemos manejar para garantizarse el mejor control posible. Algunos de estos parámetros, generalizados a cualquier librería, son:

- **Iteraciones**: define el número de veces que el entrenamiento se repetirá. Si ofrecemos datos de validación, al final de cada iteración se hará un entrenamiento especial con el subconjunto de validación que permite a la red generalizar mejor y ajustar sus pesos pertinentemente.

- **Tamaño de lote**: este es uno de los parámetros más importantes. El tamaño del lote indica cuántas muestras se procesarán a la vez en cada paso de una iteración. Por ejemplo, en nuestro caso, tenemos 3000 imágenes, de las cuales el subconjunto de entrenamiento son un 60 %, es decir, 1800 imágenes. En cada iteración del entrenamiento se procesarán las 1800 imágenes. El tamaño del lote indica cuántas se procesarán a la vez. Este valor suele depender de nuestra capacidad de cómputo, más específicamente, de la RAM (o VRAM) disponible en nuestro dispositivo. Se suele usar un valor de entre 8 y 64, en potencias de dos. Si tomamos un valor de 16, por ejemplo, se procesarán las imágenes en lotes de 16 muestras hasta completar el subconjunto.

- **Subconjunto de datos de validación**: si tenemos una partición de validación, la especificamos para que el algoritmo la utilice.

- **Partición de validación**: algunos *frameworks* ofrecen este parámetro para que, en caso de no disponer de una partición de validación, el algoritmo automáticamente nos crea una partición dado nuestro conjunto de datos de entrenamiento. Se pasa un número entre 0 y 1 que indica qué porcentaje del conjunto de entrenamiento se usará para validación. Si escogemos 0.3, por ejemplo, un 30 % de las muestras se usarán como validación, dejando el conjunto de entrenamiento en un 60 % del original.

- **Barajar el subconjunto de datos**: muchas librerías ofrecen la opción de barajar el conjunto de entrenamiento al inicio de cada iteración, para evitar crear asociaciones entre datos que no tienen una componente temporal o consecutiva.

- ***Callbacks***: Las *callbacks* son funciones avanzadas que permiten controlar y monitorizar aspectos del entrenamiento durante su ejecución. La lista es bastante extensa y probablemente varíe de librería a librería. Una de las funciones más comunes, sin embargo, es la de la finalización prematura del entrenamiento. Podemos monitorizar los valores de pérdida de nuestra red neuronal y, si la diferencia entre iteraciones de este valor empieza a disminuir, podemos considerar

[3]En realidad los parámetros del modelo (de la red neuronal), aquellos que se ajustan internamente durante el entrenamiento, esencialmente son los pesos y *bias*. Luego están los hiperparámetros, que son los que se mencionan a continuación, que nosotros manejamos, pero nos referiremos a ellos como ”parámetros”.

que la red ha convergido, por lo que no tendría sentido continuar. Una función de *callback* puede ayudarnos en este proceso y finalizar cuando lo estipulemos.

2.2. Visión artificial

La visión artificial o *Computer Vision* (CV) es la rama de la informática que se dedica a diseñar algoritmos que permitan extraer información de imágenes. Su principal objetivo es entender y automatizar procesos que el sistema de visión humano puede ejecutar. Linda Shapiro y George Stockman comentan en su libro *Computer Vision* [16]:

> *The goal of computer vision is to make useful decisions about real physical objects and scenes based on sensed images.*

«El objetivo de la visión artificial es ofrecer decisiones útiles acerca de objetos físicos reales y escenas basadas en las imágenes percibidas.»

En esta sección trataremos brevemente algunas de las técnicas propuestas basadas en CV. Si bien no entraremos en detalle acerca de las demostraciones matemáticas de por qué funcionan ni ahondaremos demasiado en su funcionamiento, ya que ello no entra en el ámbito de este proyecto, sí que haremos un breve repaso a modo introductorio de todos aquellos que específicamente se vayan a utilizar, con objeto de que todos los resultados de las experimentaciones se entiendan de la forma más clara posible.

2.2.1. Imágenes binarias

Una imagen binaria es una imagen cuyos píxeles son 0 o 1. Es realmente así de simple. Aún así, será uno de los elementos más importantes en los procedimientos que acometeremos, ya que todos los flujos de trabajo centran su atención en que en algún punto se obtenga una imagen binaria de la que podemos extraer información mucho más fácilmente que de una imagen en escala de grises o a color.

El tratamiento de una imagen binaria es mucho más simple y se puede extraer mucho conocimiento de las mismas ya que muchas imperfecciones salen a la luz cuando se aplican diferentes efectos para potenciarlos. Esta filosofía es la que seguiremos durante el proceso para cumplir nuestro objetivo.

2.2.2. Operaciones en imágenes binarias

Dedicaremos esta sección a repasar algunas de las técnicas y algoritmos más utilizados para obtener imágenes binarias y los fundamentos que subyacen bajo los mismos.

Umbralización

La umbralización es uno de los algoritmos más simples pero más importantes. Consiste en la conversión de una imagen no binaria en una imagen binaria, así que lo utilizaremos inevitablemente en alguna parte de nuestro algoritmo.

El proceso de umbralización consiste en determinar un valor umbral, T y biseccionar la imagen a partir de dicho umbral. Todos los píxeles $I_{i,j}$ de la imagen I que tengan un valor menor que el umbral serán cero, todos los que superen dicho umbral, uno.

$$\forall\ I_{i,j} \in I \begin{cases} 0, & \text{si } I_{i,j} < T \\ 1, & \text{en otro caso} \end{cases} \tag{2.3}$$

Ecuación 2.3: Ecuación de umbralización

La pregunta que nace tras esta exposición es bastante obvia: ¿cuál es el umbral T más apropiado?

La elección del umbral es una decisión tan importante que se han hecho muchos estudios y desarrollado técnicas para poder determinar el más apropiado de forma automática.

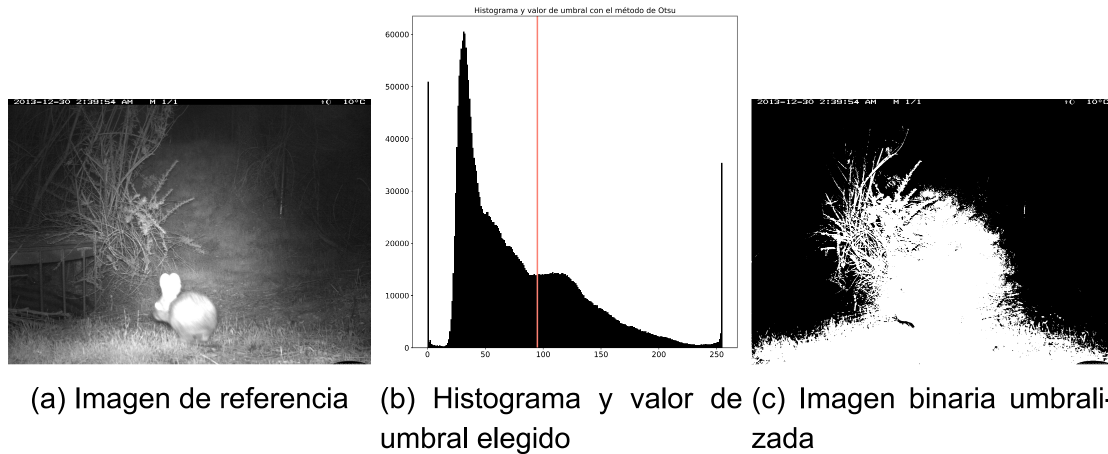
Umbralización Otsu

El proceso de umbralización comúnmente denominado como *El método de Otsu* [17] es uno de los primeros procesos de selección automática de umbral.

La principal idea es que el umbral devuelto consiga separar los píxeles de la imagen en dos conjuntos: el fondo y el sujeto principal.

El algoritmo prueba todos los posibles valores de umbral y busca aquel que minimice la varianza intraclase o, lo que es equivalente, maximice la varianza entre clases, siendo las clases los conjuntos a separar.

Este es uno de los algoritmos más simples para elección de umbral automática y precursor de muchos otros métodos que mejoraron la técnica de selección automática



(a) Imagen de referencia

(b) Histograma y valor de umbral elegido

(c) Imagen binaria umbralizada

Figura 2.8: Ejemplo de imagen binarizada utilizando el método Otsu.

de umbral, pero que de algún modo se basan en él. Un ejemplo de su funcionamiento se presenta en la figura 2.8.

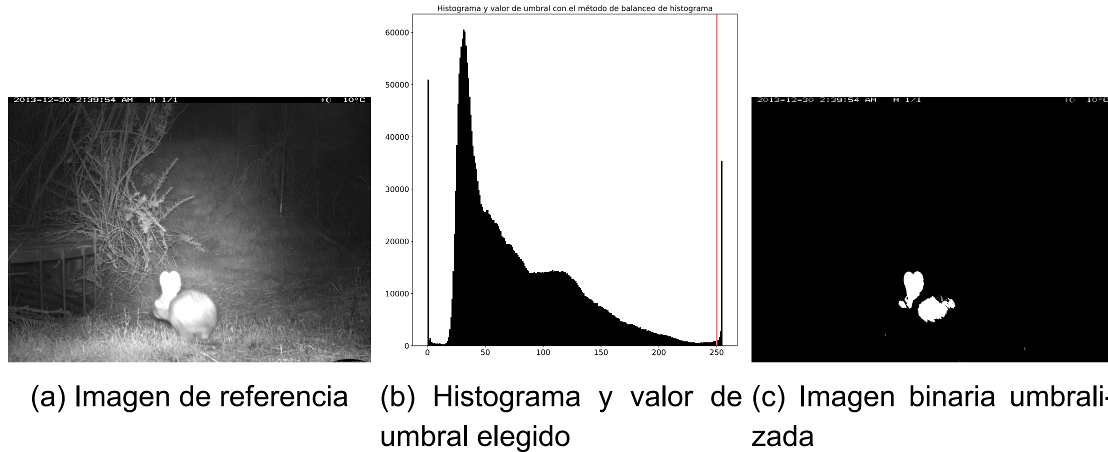
Balanceo de histograma

Otra técnica muy similar es la de balanceo de histograma [18]. Esta ténica, al igual que *El método de Otsu* trata de encontrar aquel umbral que divida la imagen en dos clases.

Para ello, esta técnica interpreta el histograma de la imagen como si fueran pesos en una balanza, y estos se desplazan fuera de la balanza al deslizarse por su propio peso. Conforme se van deslizando y eliminando los lados más pesados, el otro lado irá ganando peso, por lo que la balanza se tornará hacia la otra dirección. El proceso acaba cuando la balanza se estabiliza porque a ambos lados tiene el mismo peso, que es el punto en el histograma que con más probabilidad divida la imagen en las dos clases anteriormente mencionadas. Un ejemplo de su funcionamiento se presenta en la figura 2.9.

Otros algoritmos de umbralización

Como se mencionó anteriormente, el proceso de selección automática de umbral ha sido objeto de estudio de forma exahustiva estos últimos años, y se han desarrollado muchas ténicas que buscan diferentes formas y objetivos para segmentar la imagen. En *Survey over image thresholding techniques and quantitative performance evaluation* [19] se evalúan muchos de estos métodos, además de ofrecer una clasificación para todos, tomando como criterio el principal recurso que utilizan. Dicha clasificación puede resumirse en:



(a) Imagen de referencia (b) Histograma y valor de umbral elegido (c) Imagen binaria umbralizada

Figura 2.9: Ejemplo de imagen binarizada utilizando el método de balanceo de histograma.

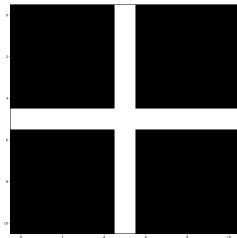
- **Métodos basados en histograma**, en donde la forma del histograma se analiza para buscar el punto óptimo
- **Métodos basados en agrupamiento**, donde los diferentes píxeles de la imagen se agrupan en función de su valor para formar el fondo y el sujeto.
- **Métodos basados en entropía**, que buscan la entropía del fondo y el sujeto para diferenciarlos, como *Background Subtraction*.
- **Métodos basados en detección de atributos de objetos**, donde se buscan grupos de píxeles similares, bordes, etc.
- **Métodos espaciales** que utilizan distribuciones de probabilidad o correlación entre píxeles.
- **Métodos locales** que adaptan el valor del umbral para cada píxel en función del contexto en el que dicho pixel está inscrito. Es decir, se elige un valor T diferente para cada píxel en la imagen.

Dilatación y erosión

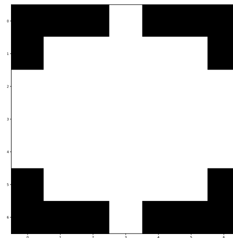
La dilatación y erosión son dos técnicas de carácter morfológico que tratan de dar forma a los conjuntos de píxeles de una imagen binaria.

Los dos métodos son complementarios, es decir, el proceso es el mismo pero la dilatación se centra en los píxeles blancos y la erosión, en los negros.

La idea es pasar lo que se denomina un *elemento estructurante* por la imagen. Para entenderlo mejor, podemos acudir a la analogía de que es como el puntero del pincel en cualquier software de carácter artístico, como Paint o Photoshop. Un ejemplo de cómo son estos elementos puede verse en la figura 2.10.



(a) Elemento estructurante en forma de cruz



(b) Elemento estructurante en forma de elipse

Figura 2.10: Ejemplo de dos elementos estructurantes provistos por la librería OpenCV, con tamaño de 5 píxeles. Esta es la punta de rotulador que vamos pasando a lo largo de la imagen para modificarla.

El centro de dicho elemento, cuya forma se define a la que más convenga, se pasa por todos los píxeles de la imagen, y se actualizan los píxeles que caigan debajo del elemento estructurante en función del modo que hayamos elegido.

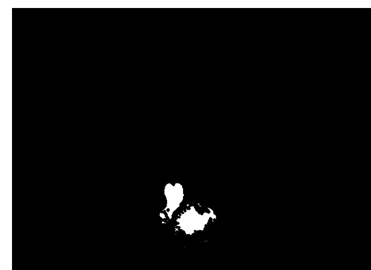
Por ejemplo, a la hora de dilatar una imagen binaria, tomamos dicho elemento y pasamos su centro por todos los píxeles. Si el píxel es negro, se ignora y continúa. Si, en cambio, es blanco, se *estampa* el elemento centrado en dicho píxel, coloreando de blanco su vecindario.

Alternativamente, al erosionar, si encontramos un píxel blanco lo ignoramos, pero si encontramos uno negro, estampamos el elemento, coloreando de negro su vecindario.

Este proceso cambia la forma de los diferentes grupos de píxeles de la imagen y además rellena los posibles huecos que hubiera dentro de ellos. Es una operación muy útil para eliminar ruido que se escapa a la umbralización y limpia de forma efectiva la imagen binaria.

(a) Imagen umbralizada con muchos artefactos y detalles

(b) Después de una pasada de erosión, con tamaño de kernel de 3 píxeles



(c) Después de 5 pasadas de erosión, con tamaño de kernel de 1 píxel

Figura 2.11: Ejemplo del efecto de erosión en una imagen con muchos detalles y ruido innecesarios. Si el objeto que queremos conservar está bien definido en la imagen original, pasará todos los filtros que le apliquemos a la imagen.

Como vemos en la figura 2.11 el resultado es una imagen binaria mucho más limpia y de la que podemos obtener bastante información. Este es, en general, el objetivo que hay que perseguir a la hora de construir un sistema con estas técnicas.

La erosión y dilatación se suelen utilizar muy a menudo consecutivamente. El proceso de erosionar y después dilatar se llama apertura (*opening*), y el proceso de dilatar y después erosionar, cierre (*closing*). Estos conceptos y nombres realmente se heredan de las operaciones existentes en morfología matemática.

2.2.3. Ecualización de histogramas

La ecualización de histograma es una técnica de procesamiento de imágenes que consiste en redistribuir las intensidades de los píxeles en la imagen con objeto de que el histograma quede plano. Esto implica una distribución uniforme de todos los valores de intensidad por todo el rango al que pertenecen, lo que, generalmente, suele resultar en un aumento de contraste y calidad de la misma.

Este proceso es también muy útil en nuestro flujo de trabajo, ya que permite extraer los detalles que pudieran perderse en la imagen original, y tiene buena sinergia con los procesos de umbralización anteriormente mencionados, ya que, al variar el histograma, los algoritmos funcionarán de forma diferente.

En la figura 2.12 se puede apreciar el efecto conseguido en una imagen tomada del conjunto de datos. La distribución de las intensidades se ensancha, lo que aumenta el rango de la imagen, traduciéndose en una mejora de contraste.

Ecualización Adaptativa

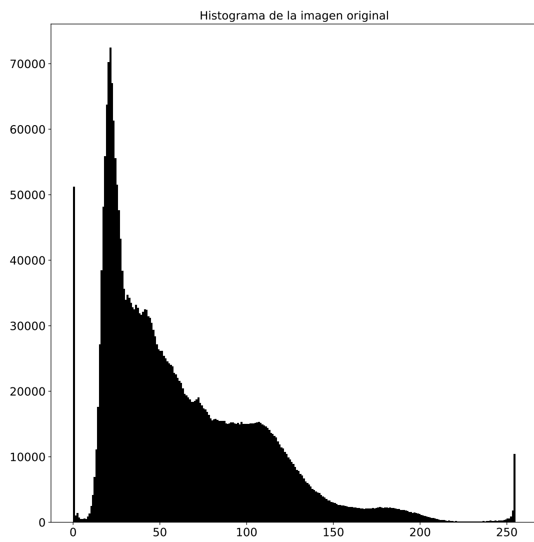
La ecualización de histograma adaptativa (*Adaptive Histogram Equalization*, AHE [20]) es una mejora a la implementación clásica de ecualización. El factor adaptativo se implementa como una ecualización de histogramas pero de segmentos de la imagen, definidos por una cuadrícula. Ecualizando el histograma en pequeños segmentos de la imagen ofrece una mejor ecualización de cada fragmento, ya que el contexto es local.

La idea consiste en subdividir la imagen en una retículo con un tamaño de cuadrícula predefinido. Se aplicará ecualización de histograma en cada fragmento a nivel local. Para evitar inconsistencias, los diferentes píxeles de los cuadrantes se interpolarán pertinentemente dependiendo del contexto en el que se hallen.

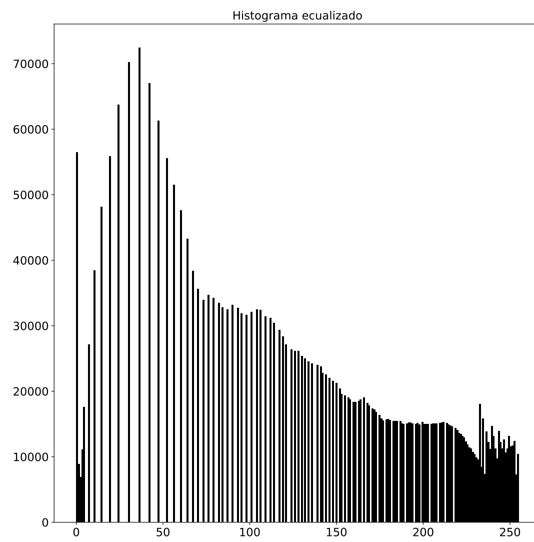
En la figura 2.13 se presenta un ejemplo para comprender mejor el proceso.

(a) Imagen original

(b) Imagen al ecualizar el histograma



(c) Histograma original



(d) Histograma ecualizado

Figura 2.12: Ejemplo del efecto de ecualización del histograma de una imagen. Se puede apreciar gran diferencia en la distribución del histograma, así como en el aumento de contraste de la imagen.

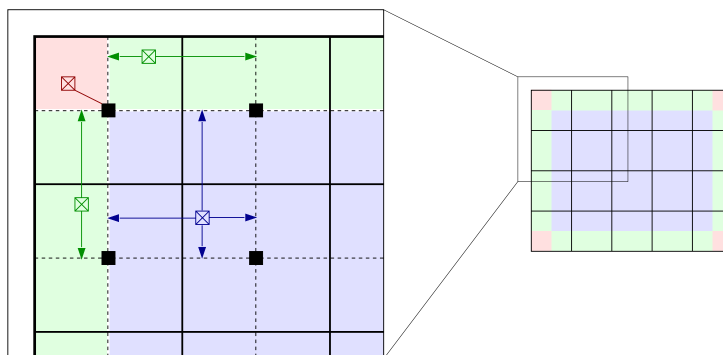


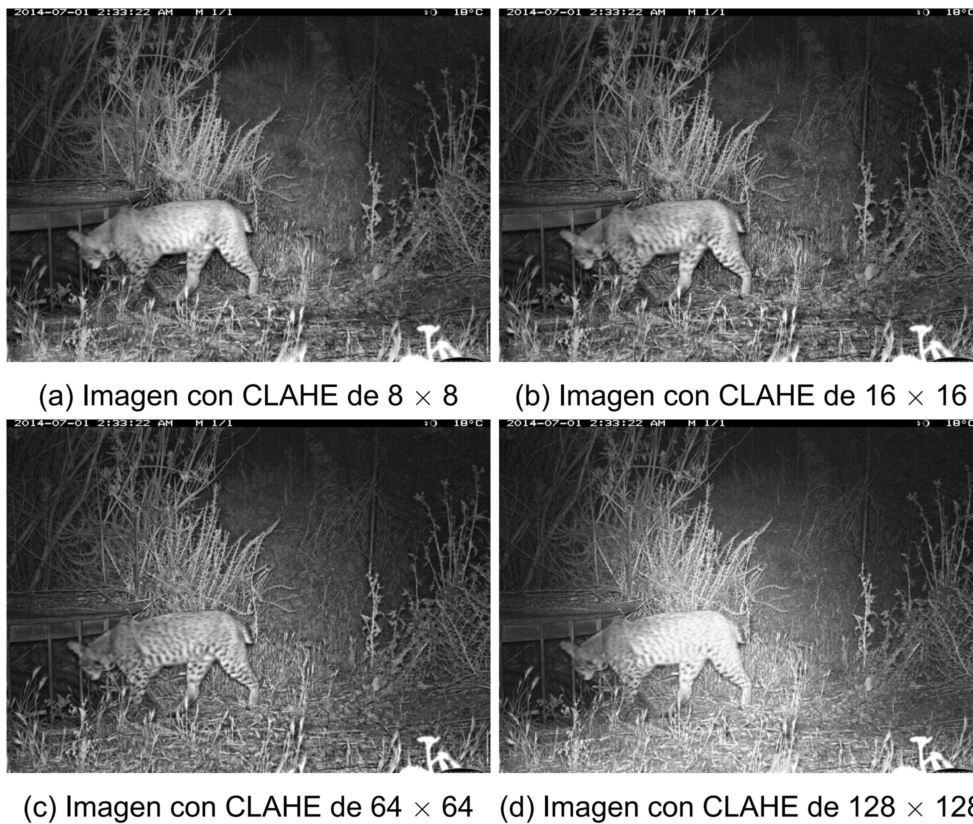
Figura 2.13: Ejemplo de la interpolación efectuada en los diferentes casos.

Al dividir la imagen usando el retículo predefinido, se calcula un histograma, una función de distribución acumulativa y las funciones de transformación pertinentes. Las funciones de transformación son funciones de interpolación que dependerán del contexto del píxel. La ecualización del histograma basado en el histograma generado y la función de distribución acumulativa son apropiadas para los píxeles marcados como negros en la imagen. Para aproximar los demás, se usan las funciones de transformación de este modo:

- En los píxeles que pertenecen al interior de la imagen, marcados con **azul**, se utiliza una función de interpolación bilineal.
- En los píxeles de los bordes, marcados con **verde**, se interpola linealmente a sus vecinos, tal y como se expresa en el gráfico: siguiendo el eje del borde.
- En los píxeles de las esquinas, marcados con **rojo**, se utiliza el vecino más cercano.

Un problema que tiene este método es que tiende a sobresaturar el contraste de la imagen, lo que puede provocar pérdidas de información. La versión que más se utiliza es la refinada *Contrast Limited Adaptive Histogram Equalization* (CLAHE) [21].

Esta técnica establece un umbral máximo de contraste y, si tras la operación se sobrepasa dicho límite, esos píxeles se distribuyen en las demás intensidades, hasta que el histograma no supere el umbral establecido, o no lo supere por un margen que el usuario indique. Es una adición simple pero efectiva para controlar de una manera más intuitiva el contraste de la imagen. Una demostración de cómo varía la imagen en función de la cuadrícula puede consultarse en la figura 2.14



(a) Imagen con CLAHE de 8 × 8 (b) Imagen con CLAHE de 16 × 16

(c) Imagen con CLAHE de 64 × 64 (d) Imagen con CLAHE de 128 × 128

Figura 2.14: Evolución del contraste de la imagen en función de diferentes tamaños de cuadrícula.

Capítulo 3

Estado del arte

En este capítulo haremos un breve repaso de las técnicas y algoritmos que más relevantes nos puedan resultar para poder resolver nuestro problema. Para ello, tendremos en cuenta que el elemento principal que vamos a estudiar es la imagen y todo lo relativo a la extracción de información de la misma.

3.1. Deep learning: arquitecturas más conocidas

En esta sección discutiremos algunas de las arquitecturas más utilizadas en la actualidad, definiremos sus principales propiedades y usos y concluiremos con un resumen de aquellas que puedan funcionar mejor para el problema que nos ocupa.

3.1.1. Feed Forward and Deep Feed Forward

Partiendo de la arquitectura de un perceptrón en la figura 2.2, incluyendo varias capas e interconectándolas entre sí obtenemos la arquitectura de una *Feed Forward Artificial Neural Network*. Una de las principales características de estas redes es la ausencia de ciclos en la misma. La información, como se puede ver en la figura 3.1 solo circula hacia delante, de ahí el nombre *Feed Forward*. Este tipo de arquitectura también suele tomar el nombre de *Multilayer Perceptron* (MLP), debido a su simplicidad y similitud con la misma.

Esta arquitectura y su funcionamiento son la base de muchos otros tipos de redes. Veamos brevemente su funcionamiento a nivel conceptual, ya que este es el funcionamiento que muchas de las demás arquitecturas tendrán.

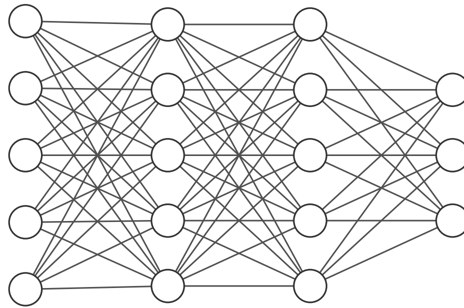


Figura 3.1: Ejemplo de la arquitectura *Feed Forward* [Elaboración propia]

Una vez procesados los datos y dispuestos de una manera que la red pueda entender, podemos proceder a introducirlos. La capa de entrada admite estos datos y, como ya sabemos, esta capa está amoldada a la dimensionalidad de los mismos. Los perceptrones de esta capa se activan con una intensidad que representa el valor que han recibido en la entrada. Tomando como ejemplo la figura 3.1, tenemos a la salida 3 neuronas y una clase asociada a cada una de ellas. Tras pasar por todas las capas, la capa de salida debe dar la misma respuesta que la etiqueta asociada a esta entrada, por ejemplo, la clase 0 (la primera empezando por arriba).

Para calcular esto, debemos ir neurona por neurona y, para cada neurona, conexión por conexión, calculando un nuevo valor para cada conexión enumerada de forma que para la entrada ofrecida, la salida correspondiente se active con mayor intensidad y que las demás salidas se activen con muy baja o ninguna intensidad. Este es el problema que en realidad está resolviendo la red, encontrar los pesos adecuados entre perceptrones, o, abstrayéndonos un poco, crear asociaciones correctas, crear conocimiento.

Calcular el valor de un determinado perceptrón en una determinada capa es una función que depende de todos los perceptrones de la capa anterior, el *bias* de dicho perceptrón y su función de activación. Podemos resumirlo tal que así:

$$a_k^{(c)} = f(w_{k,0}\ a_0^{(c-1)} + w_{k,1}\ a_1^{(c-1)} + \cdots + w_{k,n}\ a_n^{(c-1)} + b_k) = f(\sum_{i=0}^{n} w_{k,i}\ a_i^{(c-1)} + b_k) \quad (3.1)$$

Ecuación 3.1: Cálculo del valor de un perceptrón

donde $a_k^{(c)}$ es el valor del perceptrón que se va a calcular, c es la capa en la que se encuentra dicho perceptrón y $w_{k,n}$ es el peso asociado entre el perceptrón a_k en la capa c y el perceptrón a_n en la capa $c-1$, es decir, el peso de la línea que los une en los gráficos. f es una función de activación asociada al perceptrón y b es lo que se denomina *bias*, un valor que sirve a modo de umbral asociado a cada perceptrón para que no se active a menos que tenga un valor mayor o igual a b. Este valor b también

va evolucionando en el entrenamiento, sirve como un parámetro más para ofrecer un grado de libertad mayor.

En resumen, para calcular el valor que tendrá una determinada neurona en una capa, tomamos el valor de todas las neuronas de la capa anterior y hacemos una suma ponderada con los pesos de las conexiones que hay entre dicha neurona y todas los otras. Añadimos el *bias*, aplicamos la función de activación y obtenemos el valor de dicha neurona.

El cálculo de todos los pesos podría hacerse de forma iterativa, yendo neurona por neurona y, para cada neurona, conexión por conexión, como mencionábamos. Pero esto es computacionalmente ineficiente y muy lento. La manera en la que se hace realmente es mucho más elegante: utilizando matrices.

Para representar los perceptrones de una determinada capa, los agruparemos en un vector vertical:

$$a^{(c)} = \begin{bmatrix} a_0^{(c)} \\ a_1^{(c)} \\ a_2^{(c)} \\ \vdots \\ a_n^{(c)} \end{bmatrix} \tag{3.2}$$

Ecuación 3.2: Representación vectorial de una capa

donde $a_n^{(c)}$ es el valor del perceptrón, n es un índice que le damos a los perceptrones de cada capa para diferenciarlos y c es la capa en la que se encuentran. Es decir, el perceptrón $a_3^{(0)}$ es el cuarto perceptrón de la primera capa. Los agrupamos todos en el vector $a^{(c)}$.

Para representar las conexiones de forma matricial, se procede tal que:

$$W = \begin{bmatrix} w_{0,0} & w_{0,1} & \dots & w_{0,n} \\ w_{1,0} & w_{1,1} & \dots & w_{1,n} \\ w_{2,0} & w_{2,1} & \dots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,0} & w_{k,1} & \dots & w_{k,n} \end{bmatrix} \tag{3.3}$$

Ecuación 3.3: Representación matricial de los pesos

donde cada elemento $w_{k,n}$ representa el peso de un perceptrón con el perceptrón de la capa anterior. Más específicamente, cada fila representa todos los pesos que hay entre todos los perceptrones de una capa y un específico perceptrón de la capa

siguiente, a_k. Así, el vector $\begin{bmatrix} w_{k,0} & w_{k,1} & \dots & w_{k,n} \end{bmatrix}$ es el resumen del peso de todas las conexiones que hay entre el perceptrón a_k en la capa c y todos los perceptrones de la capa $c-1$.

Aplicando lo expuesto en la ecuación 3.1 pero generalizándolo a una expresión matricial, nos queda la ecuación 3.4, que puede resumirse como la ecuación final 3.5.

$$\begin{bmatrix} a_0^{(c)} \\ a_1^{(c)} \\ a_2^{(c)} \\ \vdots \\ a_n^{(c)} \end{bmatrix} = f\left(\begin{bmatrix} w_{0,0} & w_{0,1} & \dots & w_{0,n} \\ w_{1,0} & w_{1,1} & \dots & w_{1,n} \\ w_{2,0} & w_{2,1} & \dots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,0} & w_{k,1} & \dots & w_{k,n} \end{bmatrix} \begin{bmatrix} a_0^{(c-1)} \\ a_1^{(c-1)} \\ a_2^{(c-1)} \\ \vdots \\ a_n^{(c-1)} \end{bmatrix} + \begin{bmatrix} b_0^{(c)} \\ b_1^{(c)} \\ b_2^{(c)} \\ \vdots \\ b_n^{(c)} \end{bmatrix} \right) \tag{3.4}$$

Ecuación 3.4: Representación matricial del cálculo de todos los pesos de una capa a otra

$$a^{(c)} = f(W\ a^{(c-1)} + b) \tag{3.5}$$

Ecuación 3.5: Representación resumida del cálculo de los pesos

Esto es el cálculo que se hace para determinar el valor de los diferentes pesos y perceptrones de una capa a la siguiente. Este proceso se repite para la interacción entre todas las capas y se ofrece una salida final.

A partir de aquí, dependiendo de la complejidad de la red, se obtendrá una salida y se comparará con la etiqueta, se ejecutará el algoritmo de *backpropagation* si es que estuviera implementado y este proceso se repetiría para cada una de las muestras. Así, los valores de los pesos irán evolucionando, siendo cada vez más apropiados para ofrecer una respuesta correcta.

Calcular los parámetros de la red neuronal de esta forma es muy eficiente por, entre otras, dos razones principales: el cálculo y operaciones con matrices está muy optimizado en el ámbito de la computación y las tarjetas gráficas son incluso más rápidas haciendo operaciones con vectores y matrices de lo que lo son los procesadores. Combinando un buen cálculo de matrices con dispositivos familiarizados con ellas hace que el proceso sea mucho más rápido de lo que de otra forma sería. Es por ello que se recomienda entrenar redes neuronales en las tarjetas gráficas, en vez de en la CPU.

3.1.2. Convolutional Neural Network (CNN)

Las operaciones utilizadas en las redes similares a una multilayer perceptron pueden representarse como una operación matricial de las funciones de activación y los

pesos asociados. Las redes convolucionales, en cambio, utilizan otro tipo de función lineal: la *convolución*, en conjunción con métodos de *downsampling*, como el *pooling*.

Convolución

La convolución es una operación matemática aplicada en dos funciones (f y g), que da como salida otra función ($f * g$), la cual describe cómo la primera se ve afectada por la segunda.

Para nuestro contexto, esta definición se implementa de forma que un núcleo (*kernel*) se aplica por la imagen, ofreciendo una nueva imagen cuyos píxeles resultan ser el valor de multiplicar la correspondiente porción de imagen que cae debajo del *kernel* con los valores de dicho *kernel*, y después sumar todo este resultado. Este proceso se hace para toda la imagen, en forma de barrido. Este proceso se ilustra en la figura 3.2.

Para los casos de los extremos, lo más común es añadir una fila y una columna de 0 a cada lado, de forma que el *kernel* pueda encajar correctamente y no desbordarse. Otras alternativas incluyen el método de *espejo* o *cilindro*, que toman la imagen como un objeto cilíndrico de forma que el extremo de la derecha está conectado con el de la izquierda, y el de arriba con el de abajo. A la hora de acceder a un elemento fuera de la imagen, accederíamos al del lado contrario. Sin embargo, este es menos utilizado.

El resultado de estas operaciones es un mapa de características que sería muy difícil de conseguir utilizando solo perceptrones en una MLP. El tamaño equivalente sería desproporcionadamente grande y muy costoso de entrenar, mientras que estas redes pueden alcanzar un mejor resultado en una arquitectura mucho más compacta.

Estas redes presentan un proceso de regulación, que indica que no todos los perceptrones de una capa están conectados con las de la siguiente, sino que esto depende de la intensidad de los píxeles en los mapas de características generados por las convoluciones. Esto causa que la red sufra de un índice de *overfitting* mucho menor del que suelen presentar las multilayer perceptron.

Como característica final, es destacable añadir que estas redes poseen un aspecto similar al del córtex visual de los animales, algo que es muy interesante de cara al procesamiento de imágenes.

Debido a la ingente cantidad de información que porta una imagen, por lo general no se suele utilizar la imagen al completo. Prácticamente todas las implementaciones de redes neuronales de cualquier tipo que tengan una componente convolucional con el objetivo de procesar imágenes deberán hacer uso de algoritmos de reducción de información, o *down sampling*. El más usual es el *pooling*.

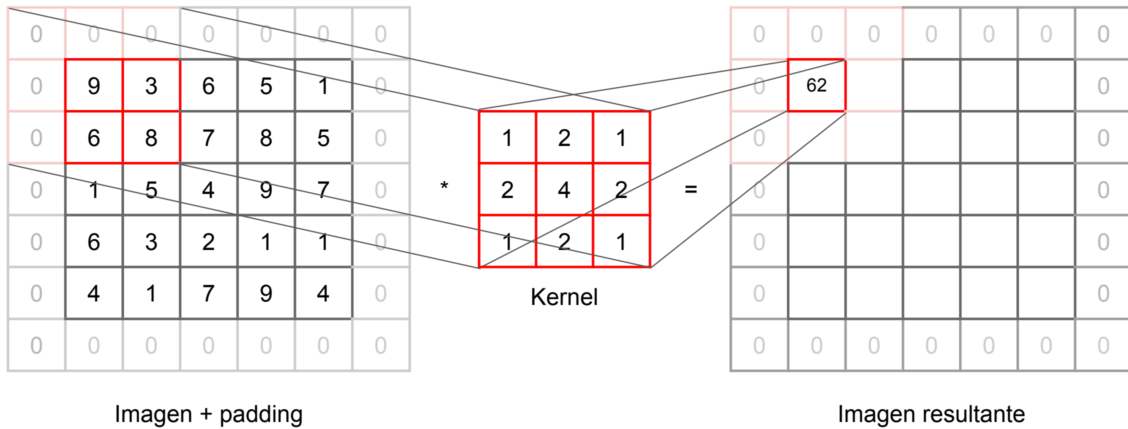


Figura 3.2: Ejemplo del proceso de convolución en una imagen añadiendo padding [Elaboración propia]

Pooling

Pooling, del inglés agrupar, trata de convertir un conjunto de entrada en uno de salida más pequeño, pero conservando la información esencial que inicialmente se tiene.

El algoritmo toma un argumento: el factor de agrupamiento que deseamos. La salida será una imagen con unas dimensiones n veces más pequeñas, siendo n el argumento que pasamos. Si $n = 2$, la imagen será 2 veces más pequeña.

El criterio para elegir los nuevos valores dependerá del tipo de pooling que hagamos. `MaxPooling` toma el máximo de los grupos seleccionados, `AveragePooling` toma la media, o incluso `GlobalPooling`, que trata de comprimir toda la imagen en un solo valor. Se ilustran los ejemplos en la figura 3.3.

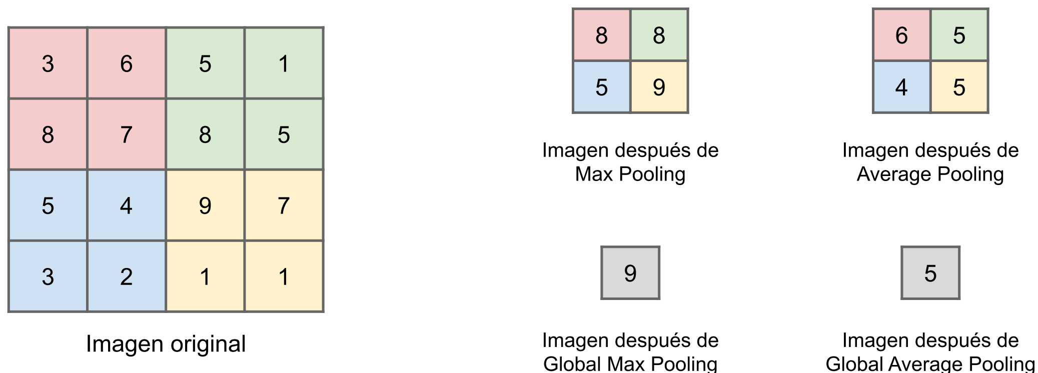


Figura 3.3: Ejemplo de algunos de los tipos de Pooling [Elaboración propia]

Flat layer y capa final

Finalmente, todos los datos se desenrollan en una capa unidimensional. La idea de esta última capa plana con una sola dimensión es la creación de un sistema completamente interconectado con la capa final, que permita obtener como entrada este vector unidimensional y ofrecer como salida las características que se buscan. La capa de salida suele tener un número de valores concreto, como por ejemplo las características que queramos encontrar, o incluso ser binaria, indicando la presencia o ausencia de aquello para lo que hemos entrenado el modelo. El resumen de toda esta explicación puede verse al completo en la figura 3.4.

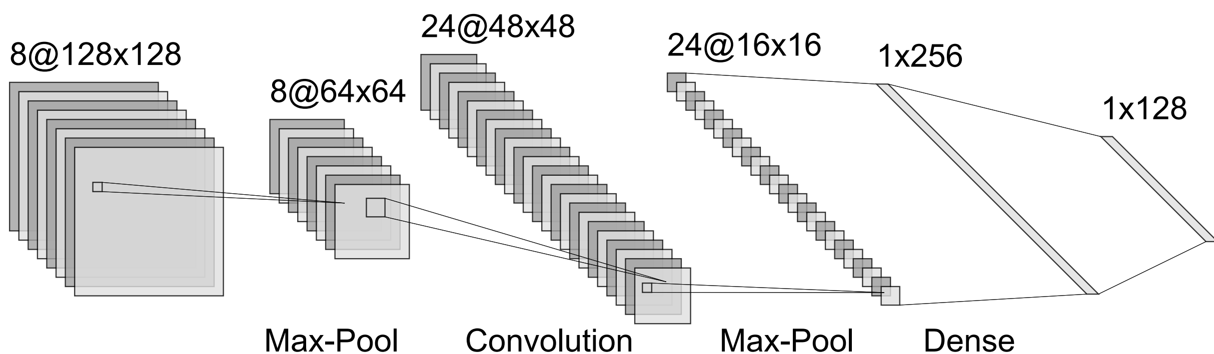


Figura 3.4: Arquitectura clásica de una Convolutional Neural Network [Elaboración propia]

3.1.3. Auto Encoder (AE)

Un AE [22] es, en esencia, una red neuronal que produce codificaciones para los datos de entrada y son entrenadas para que las descodifiaciones de salida se parezcan lo máximo posible a dichos datos de entrada. El principal objetivo de los AE es la reducción de la dimensionalidad de los datos, algo equivalente a un proceso de *Principal Component Analysis* (PCA) [23] pero de forma no lineal.

El AE presenta una estructura con un conjunto de capas que codifican los datos de entrada, el *encoder*, una capa intermedia, que guarda los datos comprimidos *codificados* y un conjunto de capas, simétricas a las del encoder, llamado *decoder*. La característica esencial de un AE es que la capa de entrada y de salida son idénticas, ya que muchas veces se busca que el autoencoder sea capaz de reconstruir la entrada lo más fielmente posible.

La implementación más simple de esta arquitectura es capaz de efectuar lo que se conoce como *feature fusion*: aglomerar características similares y eliminar información redundante.

El AE entra dentro de la categoría de algoritmos no supervisados, ya que no necesita ningún tipo de etiqueta para los datos que procesa. Se utiliza para fusionar carac-

terísticas fundamentales y, posteriormente, poder formar subconjuntos en los datos en base a esas características.

Más allá de la simetría en el número de perceptrones en las capas, no hay mucha más diferencias realmente destacables con respecto a una ANN común.

A partir de esta arquitectura, el AE puede aderezarse con cualquier valor que le queramos añadir: si necesitamos codificar imágenes, haremos un AE convolucional, si necesitamos procesar texto, lo basaremos en una *Long-Short Term Memory* (LSTM) [24], si necesitamos crear un modelo generativo podemos optar por un AE variable. Las posibilidades son infinitas, ya que esta estructura es meramente concecptual.

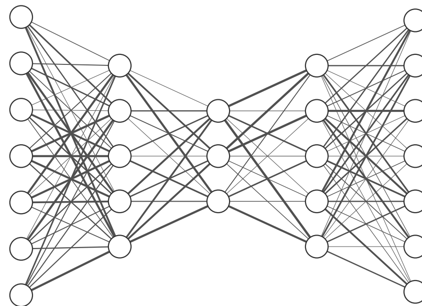


Figura 3.5: Ejemplo de la arquitectura de un autoencoder [Elaboración propia]

3.1.4. Recurrent Neural Networks (RNN)

Las RNN son un tipo de red neuronal que presenta un paradigma de conexiones que se asemeja a una línea temporal. De esta forma, este tipo de arquitecturas pueden procesar datos de longitud variable y son especialmente útiles para el tratamiento de cualquier tipo de dato que tenga inherente una componente temporal, como por ejemplo vídeos o texto.

La principal característica de esta arquitectura es que es capaz de recordar el pasado y las nuevas decisiones que tome estarán influenciadas por los datos que aprendió previamente.

Esto no quiere decir que las redes neuronales convencionales no *recuerden* cosas, sino que simplemente las recuerdan durante el entrenamiento. Un clasificador aprende lo que es un «1» durante el entrenamiento, y luego utiliza ese conocimiento para distinguir los demás números. La diferencia con las RNN es que este conocimiento, fuera del entrenamiento, es estático.

Las RNN son capaces de recordar aspectos de entradas anteriores a la hora de generar nuevas salidas. Además, pueden tomar uno o más vectores de entrada y proveer uno o más vectores de salida, pero estos no están solo influenciados por los pesos entre las conexiones sino también por los vectores de estado internos que representan

el contexto de las anteriores entradas/salidas. De esta forma, la misma entrada podría producir una salida diferente en dos ocasiones en función del contexto considerado.

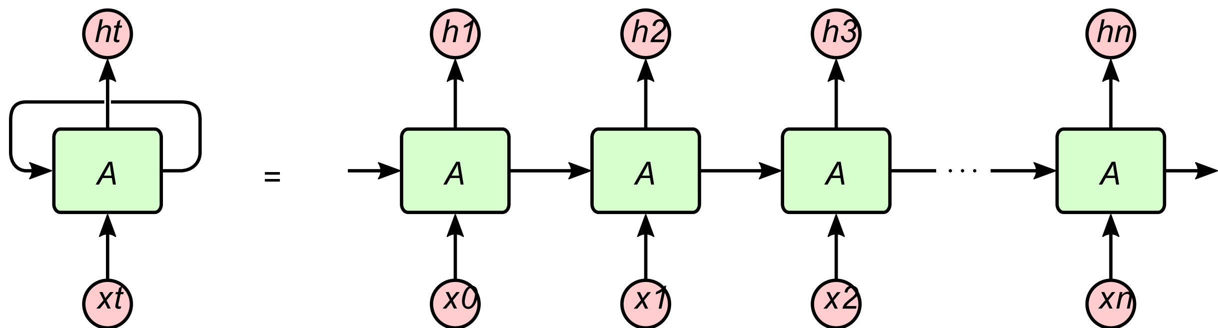


Figura 3.6: Ejemplo de la arquitectura básica de una red neuronal recurrente [25]

Las redes convolucionales clasificadoras de imágenes basan su éxito principalmente en la efectividad de convoluciones bidimensionales para compartir los parámetros extraídos, donde cada filtro convolucional extrae la ausencia o presencia de cierta característica en la imagen, que es en realidad una función de no solo un píxel sino también aquellos de alrededor.

En otras palabras, el éxito de las CNN y las RNN principalmente viene del concepto de *compartición de parámetros*, que supone una muy efectiva manera de distribuir las relaciones entre una de las entradas y los elementos de su vecindario de una manera más intrínseca, en comparación con una red neuronal simple.

3.1.5. Generative Adversarial Network (GAN)

Un muy buen resumen de lo que son las GAN puede encontrarse en el abstract del paper que Ian J. Goodfellow y sus compañeros propusieron cuando idearon este sistema [26]:

> *We propose a new framework for estimating generative models via an adversarial process, in which we simultaneously train two models: a generative model G that captures the data distribution, and a discriminative model D that estimates the probability that a sample came from the training data rather than G. The training procedure for G is to maximize the probability of D making a mistake. This framework corresponds to a minimax two-player game.*

«Proponemos un nuevo marco de trabajo para producir modelos generativos mediante un proceso de enfrentamiento en el que, simultáneamente, entrenamos dos modelos: un modelo generativo G que estudia la distribución de los datos, y un modelo discriminatorio D, que estima la probabilidad de que una determinada muestra tuviera

su origen en el conjunto de entrenamiento en vez de ser generada por G. El proceso de entrenamiento para G es maximizar la probabilidad de D equivocándose. El marco de trabajo se corresponde con un juego para dos jugadores con estrategia minimax.»

En esencia, se enfrentan mano a mano un generador contra un discriminador. El generador tiene como tarea ser capaz de generar nuevas muestras dado un conjunto de entrada a modelar y el discriminador tiene como tarea ser capaz de distinguir si la muestra que se le está ofreciendo para ser analizada es o no falsa, es decir, si la ha generado la máquina o es un dato de verdad. El sistema converge cuando el discriminador no es capaz de distinguir las muestras reales de las ficticias. Esto resulta en un generador increíblemente realista.

Este concepto puede aplicarse a la disciplina que queramos, ya sea en un contexto de imágenes, texto o cualquier otro tipo de datos debido a que, al igual que en el caso del autoencoder, esta estructura es meramente conceptual.

Es destacable el magnífico (y preocupante) resultado que ofrece esta arquitectura, donde es incluso más apreciable en el ámbito de las imágenes. Una red entrenada con suficientes datos de caras de personas puede generar nuevas imágenes que son extremadamente fieles a la realidad. Algunos ejemplos son StyleGAN [27] o deepfake [28].

Algunas utilidades que esta tecnología puede ofrecernos aparte de generar caras de personas que no existen es la mejora de imágenes, concretamente las astronómicas [29] o el super-escalado de texturas de videojuegos [30], entre otros.

3.2. Computer Vision

En esta sección mencionaremos dos de los algoritmos de CV más relevantes en lo que a nuestro problema refiere y haremos una breve exploración de su funcionamiento, fundamentos y posibles ventajas que nos ofrezcan. Prácticamente todos los algoritmos de CV, considerados aquí o no, funcionan con los principios y técnicas expuestas en el capítulo anterior.

3.2.1. Background Subtraction

También conocido como *Foreground Detection* [32], esta categoría de algoritmos tratan de diferenciar varias capas en una imagen, la del fondo y la principal.

Para ello, se toman muchas muestras del mismo lugar y se crea una diferencia entre el plano principal y el del fondo, pudiéndose así extraer aquellos elementos que

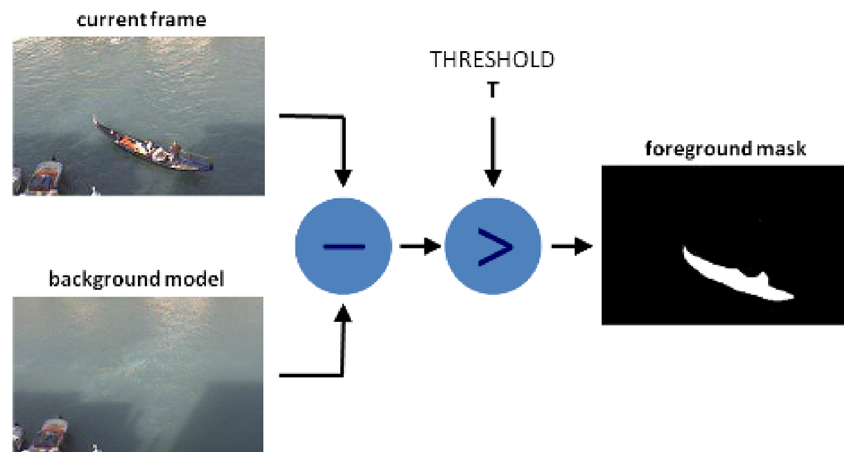


Figura 3.7: Ejemplo del funcionamiento y resultado del algoritmo Background Subtraction [31]

hayan sido añadidos a lo que podemos considerar como el fondo de la imagen, que siempre será estático. Se ilustra un ejemplo en la figura 3.7 para ayudar a comprender mejor el proceso.

Esta técnica encaja muy bien con la naturaleza de nuestro problema, ya que, usualmente, disponemos de grandes cantidades de imágenes vacías, que solo corresponden al fondo. De esta forma, se puede crear un modelo bastante preciso de qué es el fondo con un subconjunto de imágenes catalogadas manualmente como vacías, y tratar de encontrar formas que no pertenecen a dicho modelo en las imágenes que queramos analizar.

Dicho esto, hay muchas maneras de abarcar esta técnica. En *Elimination of useless images from raw camera-trap data* [33] se analizan algunos de los métodos más comunes para acometer este problema.

3.2.2. Image Segmentation & Blob Detection

También conocido como reconocimiento de regiones, es el conjunto de técnicas de procesamiento de imágenes que tratan de encontrar zonas en la imagen cuyas propiedades difieren, bajo un cierto umbral, con respecto a sus alrededores.

El objetivo del reconocimiento de regiones es simplificar o cambiar la representación de una imagen en una que sea más representativa y fácil de analizar. Normalmente se utiliza para localizar objetos y sus contornos en imágenes. Más precisamente, la segmentación de imágenes es el proceso de asignar una etiqueta a cada píxel en la imagen de manera que los píxeles con la misma etiqueta tengan características similares o pertenezcan al mismo grupo.

Este reconocimiento de regiones y segmentación de la imagen se puede hacer de muchas maneras: con métodos de umbralización (*thresholding*), con algoritmos de *clustering* [34], métodos de crecimiento de regiones o inundación [35], o métodos basados en histograma [36].

3.3. Conclusiones y propuestas

Llegados a este punto tenemos una visión muy general de algunas de las arquitecturas más famosas que tenemos disponibles y algunos de los algoritmos más útiles para el reconocimiento de siluetas.

Redes Neuronales

Recordando brevemente nuestro objetivo, deberemos procesar un gran conjunto de imágenes para determinar si en ellas aparece un animal o no. Ya solo con ese factor, está claro que debemos partir de una propuesta basada en redes convolucionales de algún tipo. Como las imágenes no tienen carácter temporal ni relación entre ellas, podemos descartar las redes recurrentes. Nuestro objetivo no es generar una imagen así que las GAN tampoco son de ayuda.

Esto nos deja con AE y CNN. Un AE es bueno codificando las características esenciales de los datos de entrada, mientras que una red convolucional es buena procesando imágenes y extrayendo características.

Por suerte, no tenemos que elegir sólo una, podemos quedarnos con lo mejor de las dos: la capacidad de extracción de características de las CNN y la capacidad de síntesis de las mismas del AE. La propuesta es un AE convolucional.

La principal idea de un AE es comprimir las características más esenciales de la entrada en la capa intermedia. De esta forma, el AE entrenado será capaz de replicar a la salida lo que obtuvo en la entrada. Esto es útil, porque si entrenamos el AE para que su compresor pueda extraer las características relevantes de muchísimas imágenes vacías, podrá replicar casi en su totalidad cualquier imagen vacía que le proporcionemos (idealmente).

De esta forma, al recibir una imagen que no esté vacía, el AE no podrá recomponerla con los datos que tiene en su compresor, provocando una salida de ruido o basura. De hecho, si medimos el error que se produce al reconstruir la imagen, sabremos si la imagen está vacía (con un error muy bajo, la imagen se reconstruye sin problemas) en contraposición con una imagen con animales (presenta un error muy alto).

Computer Vision

En otro orden de cosas, cualquier método de los propuestos en la sección 3.2 podrían ser de utilidad a la hora de reconocer la figura de un animal en una determinada imagen. El algoritmo de Background Subtraction puede ser el más prometedor, ya que podemos muy fácilmente formar un modelo de la imagen vacía, ya que disponemos de muchas de ellas.

Comparativa

Haciendo una breve comparativa para terminar, una de las principales ventajas de las redes neuronales es la precisión. Un buen modelo bien entrenado es capaz de superar incluso a un humano. Sería muy ambicioso decir que llegaremos a dicho nivel en este proyecto, pero es algo a tener en cuenta.

En cambio, las principales desventajas son el tiempo y la arquitectura. Hallar una arquitectura apropiada que supla las necesidades de nuestro propósito no es tarea fácil y la complejidad del entrenamiento que requiere un modelo de cualquier tipo suele ser muy alta. Si no se dispone de hardware especializado, puede hacerse tediosamente largo, y es un aspecto que debe ser considerado.

Los algoritmos clásicos son, en contraposición, sorprendentemente rápidos en ejecución, y, siguiendo las referencias mencionadas, probablemente más sencillos de implementar.

Su principal desventaja puede ser la sensibilidad a cambios bruscos o no contemplados en las imágenes de entrada, que supongan que el algoritmo no funcione correctamente afectando a su tasa de acierto.

Se puede observar el claro balance entre sencillez/eficacia, y todo lo que dichos factores implican. Se hará un estudio de los mismos para comprobar qué método es mejor en los siguientes capítulos.

Capítulo 4

Herramientas

En este capítulo se discutirán algunas de las herramientas disponibles para el procesamiento y clasificación de imágenes en el contexto en el que nos encontramos. Se detallarán algunas de las ventajas y desventajas que presentan y se justificará la elección de la herramienta que se acabe utilizando finalmente.

4.1. Herramientas disponibles

En esta sección se contemplarán los diferentes lenguajes de programación, *frameworks* y demás recursos que utilizaremos en nuestro proyecto. Se aportará, también, una valoración subjetiva de cada uno para justificar la elección del mismo.

4.1.1. Lenguajes de programación

Los lenguajes considerados son los siguientes:

- **Python**
    - Python es uno de los lenguajes de programación más utilizados en esta disciplina, si no el que más. Tiene soporte de varios tipos de librerías tanto de ML como de CV, lo cual lo hace especialmente interesante.
    - La principal ventaja de este lenguaje con respecto a los demás es la experiencia. Python es un lenguaje multipropósito que personalmente he usado en casi todos mis proyectos en los últimos años de estudio. Además, presenta un gran soporte por parte de la comunidad, una muy fácil gestión de los paquetes que se tengan que usar y una interfaz bastante accesible. Muchas de las librerías, como `Numpy`, están magníficamente optimizadas para

este tipo de tareas. La principal desventaja es que es un lenguaje interpretado, por lo que se puede perder algo de rendimiento, pero las rutinas internas a las que llamamos usando las bibliotecas están reescritas en C, así que no sufre tanto en ese aspecto.

- **C++**
  - C++ es un muy conocido y antiguo lenguaje de programación, cuya principal diferencia con su antecesor, C, es la orientación a objetos. También se disponen de varias librerías para ambas disciplinas, aunque no tantas como en Python.
  - La principal ventaja de este lenguaje es la profundidad de uso y optimización. C++ permite controlar en gran profundidad la memoria que usamos y cómo lo hacemos. Un buen uso del mismo resulta en programas muy bien optimizados, que aprovechan bien los recursos y aportan buenos resultados. Sin embargo, esto trae consigo una curva de aprendizaje mucho más inclinada. No existe gestión de paquetes como en Python, por lo que debemos resolver dicho problema a la vieja usanza.
- **Java**
  - Java es probablemente el lenguaje de programación más usado en el mundo. Millones de aplicaciones y dispositivos usan Java, y la adición de librerías de Machine Learning era solo cuestión de tiempo.
  - Java es un lenguaje algo más fácil de utilizar que C++ ya que sí se implementa la gestión de memoria automática y el recolector de basura, pero manteniendo aspectos importantes de optimización y gestión manual. Otro punto importante es que también poseo conocimientos suficientes del mismo como para acometer un proyecto de este calibre.
- **R**
  - R está especialmente orientado a operaciones estadísticas, lo cual lo convierte casi nativamente en un lenguaje muy apto para tareas de aprendizaje máquina. Se disponen de muchos paquetes que facilitan la implementación de modelos y de su entrenamiento, así como un exhaustivo análisis de dichos datos.
  - Los principales puntos a favor son la existencia de un amplio catálogo de paquetes y la similitud con Python, sin embargo es un lenguaje que desconozco y por mucha similaridad que guarde, uno debe adaptarse a los factores particulares de cada lenguaje. Aún así, no se descarta esta opción.

Se han mencionado los cuatro lenguajes que personalmente encuentro más relevantes y familiares a día de hoy, añadiendo el factor del propósito que le vamos a dar:

análisis de imágenes con redes neuronales. Por supuesto, habrá muchos otros lenguajes que superen a los arriba mencionados, pero lo ideal es seleccionar herramientas en las que ya se tiene experiencia.

Expuesta esa información, la elección será **Python**. Es un lenguaje con el que me encuentro muy familiarizado, dispone de muchísimas librerías muy útiles para todo lo que queremos hacer y de un gran soporte y de calidad por parte de la comunidad.

4.1.2. Técnicas y librerías

Se han contemplado las siguientes técnicas de obtención de información de imágenes, y varias librerías relacionadas, todo en referencia al lenguaje elegido:

- **Computer Vision**
    - La visión artificial reúne un conjunto de operaciones y algoritmos que permiten obtener información de una imagen, como las caras de diferentes personas, o la enumeración de objetos.

      Existen varias librerías para el procesamiento de imágenes como Pillow [37] o imutils [38], pero sin duda la más importante y relevante en nuestro caso es OpenCV [39].

      OpenCV viene de Open source Computer Vision, y añade mucha funcionalidad de cara a la obtención de información de las imágenes, como algoritmos para detección de caras y análisis de *blobs*[1], cosa que las dos mencionadas anteriormente no tienen.
- **Deep Learning**
    - **Tensorflow (Keras)** [40]
        - Tensorflow es probablemente la librería más famosa de DL en el panorama. Concretamente, existe una capa de más alto nivel llamada Keras, pero realmente todo este código se traduce a código de Tensorflow. Tensorflow se basa en modelos estáticos de redes neuronales, tiene un gran soporte por la comunidad y una curva de aprendizaje ligeramente más alta, que se compensa por la cantidad de recursos que hay disponibles en internet.
    - **PyTorch** [41]
        - PyTorch por su parte es capaz de definir modelos dinámicos, lo que permite implementar capas de longitud y forma variable. Si bien esto es

[1]Un *blob* es una forma de referirse a un conjunto de píxeles de interés. Muchas de las veces, en las imágenes, tras ser procesadas, aparecen masas informes de píxeles, y por eso se conocen comúnmente como tal.

un punto a favor, no será necesario para una implementación como la que buscamos. Es una librería más fácil de utilizar según la comunidad, pero no se dispone ni del soporte ni de los recursos que se tienen para Tensorflow.

- **SciKit** [42]
  - SciKit es una librería que implementa muchísimos de los algoritmos de aprendizaje automático más conocidos, como K-means, árboles de decisión o regresiones. Además, esto se complementa con un gran conjunto de herramientas que facilitan la tarea del preprocesamiento, gestión y postprocesamiento de los datos, útiles en conjunción con otras librerías.

4.2. Herramienta elegida

Concluyendo, expuestos los diferentes puntos para cada una de las librerías, elegiremos **OpenCV** y **Keras** para nuestro proyecto. Las razones son muy similares, ambas librerías son las librerías referentes en su ámbito y se añade el factor de que he trabajado con ellas antes, lo cual facilita la tarea en cierta medida.

Capítulo 5

Preparación de los datos

5.1. Descarga y obtención de imágenes

El primer paso será, evidentemente, obtener el material. Para ello, haremos uso de la extensa base de fotografías que nos ofrece LILA BC, concretamente su base de datos de *Caltec Camera Traps* [43], cuya particularidad con respecto de las demás es que alrededor de un 70 % del total están catalogadas como vacías.

Descargarlo no fue un paso tan trivial como acceder al enlace de descarga, ya que un archivo tan masivo (105 GB) no está pensado para ser procesado por el navegador y tomaría un tiempo excesivo en completarse (alrededor de 2 días según la estimación de mi navegador, Google Chrome).

Para ello, hemos optado por la alternativa que la misma página nos ofrece: hacer uso de la herramienta **AzCopy** [44], de Microsoft Azure, el servicio de almacenamiento masivo en el que las bases de datos en cuestión están alojadas.

5.2. Primeros pasos

La idea de haber descargado esta base de datos en concreto es por lo que menos nos interesa: las imágenes vacías. Vamos a abordar el problema de forma que consigamos diferenciar aquellas imágenes que poseen algún animal en ellas de las que estén completamente vacías.

Como esta base de datos tiene muchísimas imágenes vacías, podemos crear un modelo que comprenda perfectamente qué características definen a una imagen como tal, y nos permitirá, posteriormente, distinguirlas.

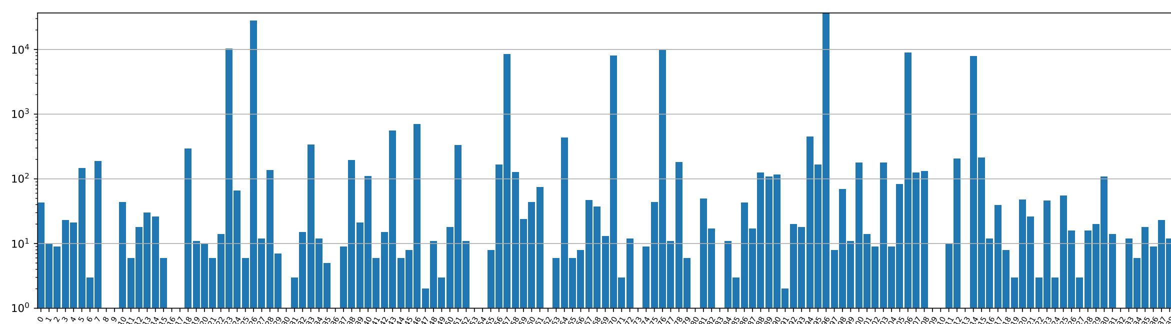


Figura 5.1: Distribución de imágenes por localización

5.2.1. Gestión y organización de las imágenes

Antes de procesar las imágenes que nos interesan, primero debemos hacer un esfuerzo en encontrarlas manualmente para poder proporcionárselas a nuestra red neuronal. Si todo va bien y el modelo responde de forma correcta, habremos encontrado una solución que, muy probablemente, funcione con cualquier otro dataset.

Como se mencionó anteriormente, el primero de nuestros objetivos es ser capaces de crear un modelo que comprenda qué características definen a una escena vacía. Para ello, debemos encontrar todas las imágenes vacías y entrenarlo con ellas.

Un archivo de metadatos en formato `.json`, representado en la Lista 1 (pág. 94), acompañaba al dataset muy pertinentemente. En él, se especifican, entre otros, dos datos importantes que utilizaremos en esta primera fase de la gestión de los archivos: la localización y la etiqueta de categoría.

La localización nos servirá para agrupar las fotografías en función de la cámara que las disparó, suponiendo que en una localización solo existe una cámara.

La etiqueta de categoría indica qué hay en la foto: coyote, ciervo o, más importante: nada. Concretamente, el valor que corresponde a la categoría «empty» es el 2.

Este proceso se ha acometido escribiendo un módulo de software que lea el archivo de metadatos, obtenga información de los nombres de qué imágenes están catalogadas como vacías, y finalmente organice dicho subconjunto en función de la localización.

En la figura 5.1 se provee una gráfica de la distribución de las fotos por población. Podemos observar un total de 126 localizaciones, aunque el máximo sea 136. El contraste también es apreciable: hay localizaciones completamente vacías, otras con alrededor de 5 fotografías y alguna localización con más de 35000.

La idea es que si el entrenamiento en una determinada localización funciona con el modelo que configuremos, habrá una muy alta probabilidad de dicho modelo funcione igual de bien para cualquier otro contexto.

5.3. Preprocesamiento de las imágenes

Ya hemos completado el primer objetivo: clasificar manualmente las imágenes vacías y además agruparlas por localización. La figura 5.1 nos ayuda a identificar qué localizaciones utilizaremos: la 26 o la 96, ambas con alrededor de 30000 fotografías. El pequeño script las organizó en carpetas con el mismo nombre, así que el acceso también es fácil.

A la hora de procesar muchas imágenes, ya sea en esta disciplina o cualquiera, se busca una consistencia en cuanto a las dimensiones de la fotografía: ancho, alto, número de canales, etc., cualquier propiedad debe ser homogénea en todo el conjunto de datos. Haber clasificado las imágenes por localización no solo nos permite organizar las imágenes, sino que nos garantiza la consistencia dimensional que buscamos al existir una sola cámara por localización (y la misma cámara siempre produce los mismos resultados a menos que se cambien algunos parámetros de configuración).

Además, es necesario a la hora de entrenar el modelo, disponer de muchas imágenes del mismo sitio, para que se aprecien los cambios de luz o flora a lo largo del tiempo y sea robusto a dichos cambios.

Las fotografías están en color originalmente. Se ejecutarán pruebas en las que las imágenes se carguen con solo un canal en vez de tres, reduciendo considerablemente el espacio que ocupan en memoria. Esto nos ayudará a comprobar cómo de importante es el color en las imágenes.

File name	5968c0f9-23d2-11e8-a6a3-ec086b02610b.jpg
Document type	JPEG
File size	2048 $\times$ 1494 px
Image size	619 KB (619.054 bytes)
Colour Model	RGB

Cuadro 5.1: Información de una de las fotografías del dataset

Como podemos comprobar en el cuadro 5.1, esa imagen concreta tiene una resolución de 2048 $\times$ 1494 píxeles y ocupa aproximadamente 619 KB. Puede parecer poco, pero cuando se extrapola a un conjunto de 30000 imágenes, el tamaño escala preocupantemente rápido.

Es por eso que antes de proveérselas al modelo, debemos reescalarlas para hacerlas más pequeñas. Un tamaño de aproximadamente 10 veces menos que el original puede ser más que suficiente, e incluso eliminando los canales de color se puede ahorrar considerablemente en espacio.

5.3.1. Procesamiento en paralelo

Procesar una imagen haciéndola más pequeña y normalizando sus valores no es una tarea muy compleja. Pero la historia se repite: extrapolando la duración del procesamiento de 1 imagen a un conjunto de entre 5000 a 10000 imágenes se vuelve una tarea algo más complicada.

Afortunadamente, al ser una tarea completamente dedicada a la CPU, podemos hacer un muy conveniente uso de los recursos programando varios hilos para procesar todas las imágenes.

Para ello, se ha hecho uso de 8 hilos de ejecución. La tarea, para un conjunto de prueba de unas 700 imágenes, pasa de tomar 46 segundos de media a solo 12 segundos aproximadamente. Es fácilmente inferible que para un conjunto más grande de imágenes la diferencia será incluso más apreciable.

El resultado es una fotografía cuyos metadatos se muestran en el cuadro 5.2.

File name	5968c0f9-23d2-11e8-a6a3-ec086b02610b.jpg
Document type	JPEG
File size	256 $\times$ 144 px
Image size	16,0 KB (16.384 bytes)
Colour Model	RGB

Cuadro 5.2: Información de una de las fotografías del dataset, procesada

Vemos que las diferencias más importantes son el tamaño y el peso. Hemos reducido la imagen en un factor de 8. Para obtener esas exactas proporciones, primero recortamos el lateral derecho ligeramente, para que al dividir quedasen números que la red neuronal pudiera procesar correctamente. El tamaño también ha variado enormemente, ocupando la imagen procesada solo un 2,6 % de lo que ocupa la original.

5.3.2. Serialización

También es destacable que no necesitamos procesar las imágenes siempre que vayamos a entrenar o testear con las fotos de cierta localización. Si ya hemos procesado las imágenes de una localización, podemos guardarlas para usos futuros.

Para ello, hacemos uso de la función `save` en `numpy`, que permite guardar la estructura de datos que pasemos como argumento en un archivo `.npy`. La estructura de datos es un *array* de imágenes ya procesadas. Guardaremos el conjunto correspondiente para el entrenamiento, la validación y el testeo en diferentes archivos.

Capítulo 6

Diseño de los modelos

Tras haber procesado todos los datos y tenerlos listos para pasarlos a los modelos, efectuaremos un proceso iterativo de experimentación para determinar cuál es la configuración de hiper parámetros que mejor se adapta a nuestro problema. Finalmente se propone una propuesta de diseño de un algoritmo basado en CV.

El entorno de entrenamiento se configura de la siguiente manera:

- Este proceso de experimentación se hará sobre un subconjunto de 3000 imágenes de la localización 96.
- Se destinará un 60 % para el conjunto de entrenamiento, un 20 % para el de validación y el último 20 % para el de evaluación.
- El optimizador utilizado es el de Adam. Otros optimizadores pueden verse en [45].
- La tasa de aprendizaje es la utilizada por defecto, 0.01.
- El tamaño de lote será de 64, a menos que se especifique lo contrario.
- Se activará la opción de barajar el conjunto de datos en cada iteración.
- El número de iteraciones será de 300.

Utilizamos un pequeño subconjunto de 3000 imágenes con el objeto de que el proceso de experimentación no tome un tiempo excesivo.

La visualización de los modelos se ha hecho utilizando la herramienta Net2Vis [46].

6.1. Primera configuración: Baseline

Esta primera configuración parte de la implementación que se hace en la documentación de Keras [47].

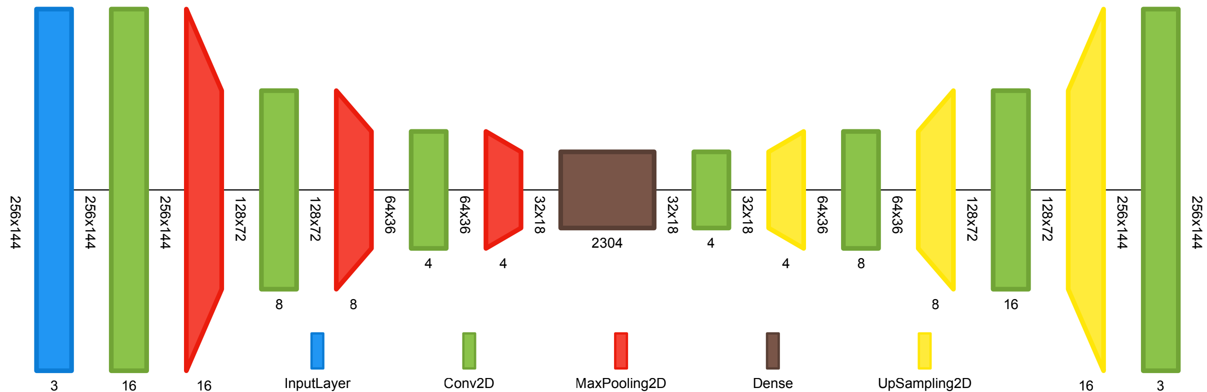


Figura 6.1: Visualización del primer modelo

Se parte de una configuración sencilla, pero hubo que modificarla ligeramente, ya que dicha configuración funcionaba bien para el *dataset MNIST* [48], pero no era suficiente para nuestro problema.

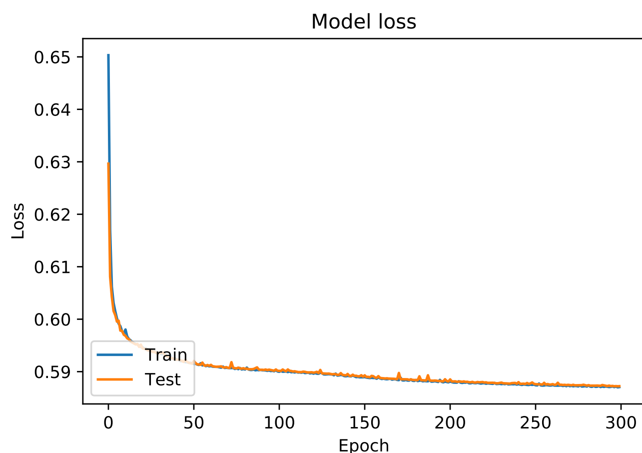
Como se puede comprobar, la diferencia entre la implementación original (listado 2, pág 95) y nuestra primera iteración (listado 3, pág 96) es la adición de una capa densa en medio que funciona como encoder, representada en marrón en la figura 6.1, además de eliminar una de las capas convolucionales internas, ya que ello presentaba una compresión excesiva de nuestras fotografías.

6.1.1. Análisis de resultados

Iteración	Valor de pérdida
1	0.6503
50	0.5916
100	0.5900
150	0.5889
200	0.5880
250	0.5876
300	0.5871

Cuadro 6.1: Evolución del primer modelo en distintas iteraciones

Podemos ver en el cuadro 6.1 que la red se estabiliza en el valor 0.5871.



(a) Función de pérdida

(b) Ejemplo de una de las predicciones

Figura 6.2: Resumen de los resultados de nuestro primer modelo

Se aprecia que ambas líneas, la de entrenamiento y la de evaluación son prácticamente iguales, solo que la de entrenamiento empieza con un valor de pérdida mayor. Muy pronto acaban fusionándose en casi la misma función.

Por lo general, los resultados de entrenamiento suelen ser algo mejores que los de la evaluación, ya que en la evaluación se entrena con datos que no se encuentran en el subconjunto de entrenamiento. Esto evita que la red *memorice* dicho subconjunto, produciendo resultados muy buenos, pero implicando una muy mala generalización, lo que implicaría un síntoma de *overfitting*.

Cuando la gráfica de entrenamiento está muy por debajo de la de evaluación, es indicativo de *overfitting*. En nuestro caso, ambas coinciden casi a la perfección, por lo que se indica que el entrenamiento va por el buen camino y la red se está entrenando correctamente.

Como se puede ver en la figura 6.2b, la imagen de arriba es la original procesada, tomada del conjunto de evaluación y, la de abajo, la reconstruida por la red. La predicción parece una imagen borrosa. Esto es algo normal en este tipo de redes, ya que intentamos comprimir las 3000 fotografías en una única codificación, e intentamos imitar una fotografía nueva en base a esa codificación. El resultado es una *media* de dichas fotos pero ponderada mayoritariamente hacia la imagen de entrada. Los detalles de las altas frecuencias[1] no se aprecian porque son los que más varían de una foto a otra, pero se puede ver claramente cómo la distribución de colores es prácticamente igual.

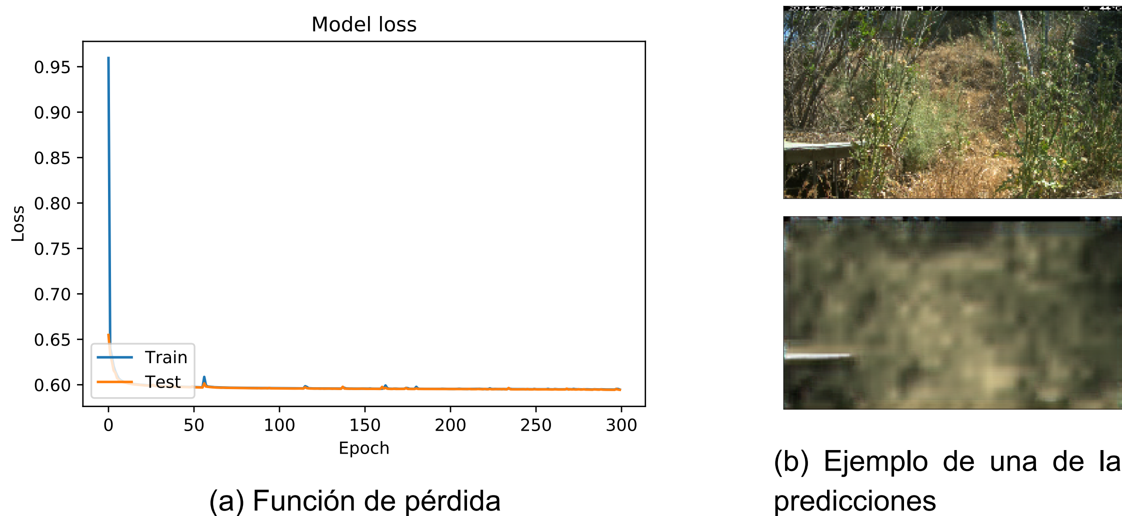
Un detalle interesante es que la plataforma presente en la zona inferior izquierda está presente en todas las fotos y destaca bastante con respecto al ecosistema. Esto

[1] Altas frecuencias en teoría de imagen hace referencia a aquellas partes de la imagen en la que los píxeles cambian bruscamente. Esto se corresponde con los detalles de la imagen.

ha sido captado pertinentemente por la red y los detalles estáticos los reproduce casi a la perfección.

Esta es la primera iteración de nuestra arquitectura de red. Procederemos a cambiar varios de los hiperparámetros para ver cómo afectan a la eficacia de la misma, con la esperanza de encontrar la configuración que mejor se adapte a nuestro problema.

6.2. Segunda configuración: Función de salida lineal



(a) Función de pérdida

(b) Ejemplo de una de las predicciones

Figura 6.3: Resumen de los resultados de nuestro modelo, con función de activación lineal

Nuestra segunda configuración es idéntica a la primera 6.1, pero hemos cambiado un parámetro clave: la función de activación de la última capa. Originalmente, utilizabamos una función sigmoide.

Es muy común utilizar funciones sigmoide, como la ilustrada en la figura 2.5 (pág. 28), en redes como las que estamos entrenando. Estas redes no tienen que dar una salida discreta, como un valor booleano (verdadero, falso) o interpretar qué numero corresponde a la imagen, o de qué raza de perro se trata. Esta red trata de recrear una imagen a partir de una de entrada, dada una codificación.

Para mejorar este resultado, una hipótesis es que si utilizábamos una función lineal al final, en la capa de decodificación, dicha función respetaría el valor que la red le ha asignado a cada píxel en su representación, mientras que usando la sigmoide se corría el riesgo de que dichos valores pudieran verse restringidos al estrecho rango que la derivada establece.

6.2.1. Análisis de los resultados

Iteración	Valor de pérdida
1	0.9596
50	0.5975
100	0.5963
150	0.5958
200	0.5954
250	0.5954
300	0.5947

Cuadro 6.2: Evolución del segundo modelo en distintas iteraciones

En la figura 6.3 y cuadro 6.2 se aprecian los resultados de esta nueva configuración.

La gráfica es similar, pero vemos que se estabiliza mucho antes que la gráfica en la anterior configuración. En la iteración 50 ya se obtenía un valor de 0.5975. El error de entrenamiento al inicio es radicalmente mayor, 0.9596, pero el valor en el que se estabiliza es prácticamente idéntico.

En la figura 6.3b vemos la predicción para la misma imagen que para la configuración anterior. En este caso, la red ha ofrecido una representación de peor calidad en general. Los colores son más tenues, por lo que la información no se reproduce adecuadamente. Es también una predicción más borrosa. Aparentemente, este paso no ha favorecido a la configuración de la red.

6.3. Tercera configuración: Escala de grises

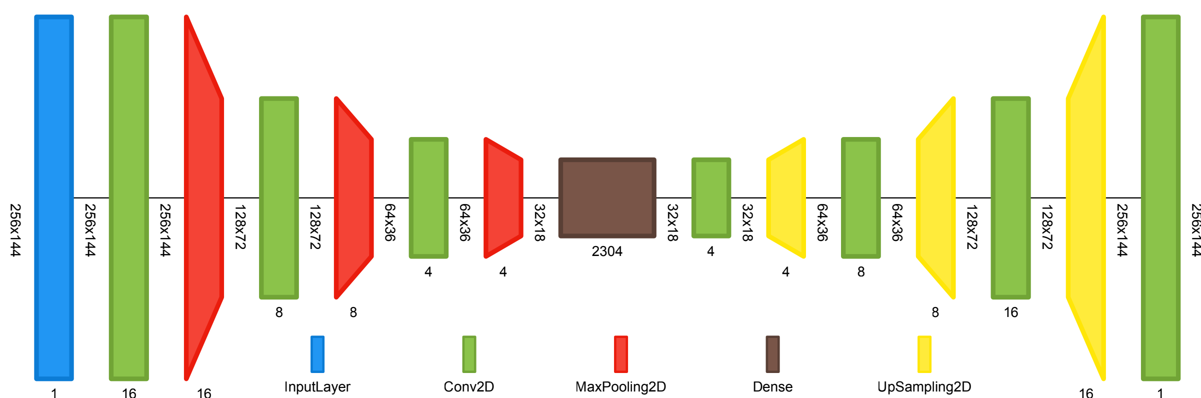
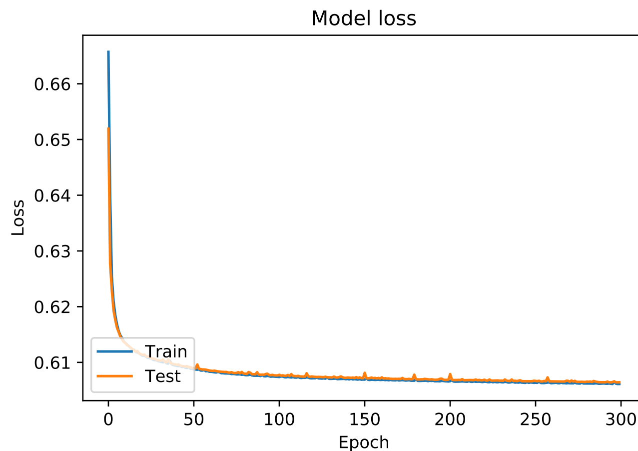


Figura 6.4: Visualización del modelo usando escala de grises

Como posible alternativa, se realizaron los mismos experimentos pero utilizando un solo canal en vez de tres. Ahora la capa de entrada tiene solo una dimensión, como indica la figura 6.4.

El modelo original ha ofrecido un resultado similar al anterior, quizá se ha perdido algo de detalle en general, pero nada especialmente destacable. Es una versión en escala de grises de la anterior.

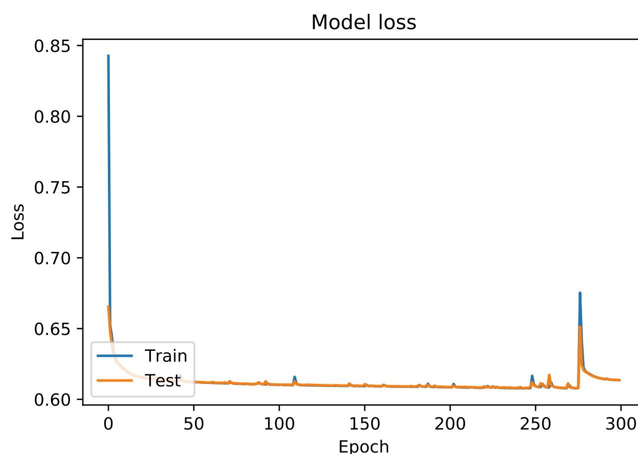


(a) Función de pérdida

(b) Ejemplo de una de las predicciones

Figura 6.5: Resultados del modelo original trabajando en escala de grises

Sin embargo, el modelo lineal ha ofrecido unos resultados muy pobres en comparación, algo sorprendente. La imagen 6.6b, como se puede apreciar, no es en absoluto utilizable.



(a) Función de pérdida

(b) Ejemplo de una de las predicciones

Figura 6.6: Resultados del modelo linear trabajando en escala de grises

Iteración	Valor de pérdida	Iteración	Valor de pérdida
1	0.8427	1	0.6657
50	0.6122	50	0.6089
100	0.6102	100	0.6074
150	0.6094	150	0.6069
200	0.6086	200	0.6066
250	0.6105	250	0.6063
300	0.6136	300	0.6061

Cuadro 6.3: Evolución del modelo monocromo a la izquierda y evolución del modelo monocromo lineal a la derecha

6.3.1. Análisis de resultados

Las gráficas 6.5a y 6.6a no difieren mucho de las versiones a color. Los valores de inicio pueden variar ligeramente, pero todas las veces el modelo acaba estabilizándose alrededor del valor 0.6 de pérdida y lo hace en aproximadamente el mismo número de iteraciones, no superándose nunca las 50.

Baseline

La imagen 6.5b es bastante similar a la 6.2b. Vemos que no se ha perdido mucho detalle, por no decir que prácticamente lucen iguales. En ambos casos, el modelo ha hecho una predicción bastante razonable y parece un buen punto de partida.

Función de activación lineal

En cambio, el modelo lineal es claramente peor que el original. Ambas versiones, a color 6.3b y en escala de grises 6.6b, son predicciones de peor calidad que las equivalentes en el modelo con la función sigmoide.

La versión linear en escala de grises, sin embargo, es prácticamente inutilizable, en comparación con la versión lineal pero a color.

Conclusiones

Dados estos resultados, configuraremos un nuevo modelo basado en el original, descartando la opción con la función de activación lineal. Trataremos algunas técnicas

que se nos proveen como matrices traspuestas que veremos detalladamente en las siguientes secciones.

6.4. Cuarta configuración: Matriz de convolución traspuesta

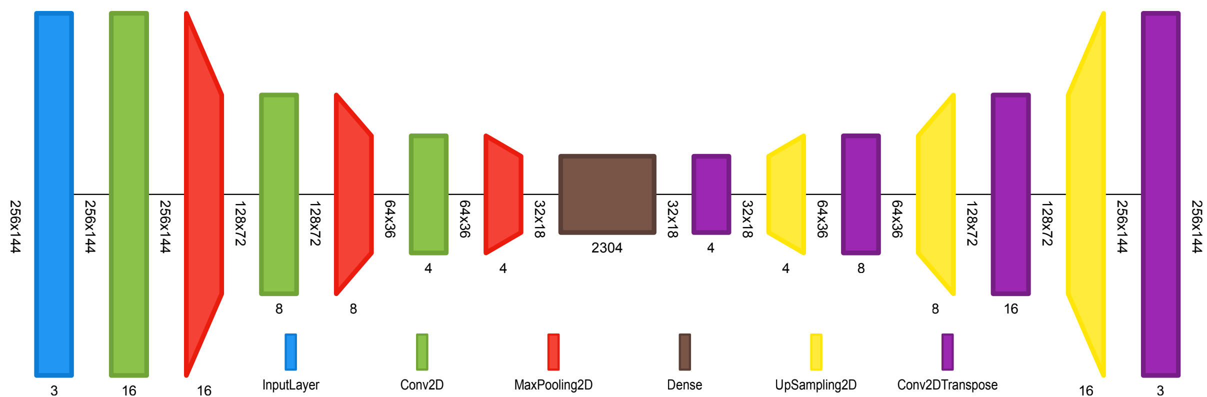


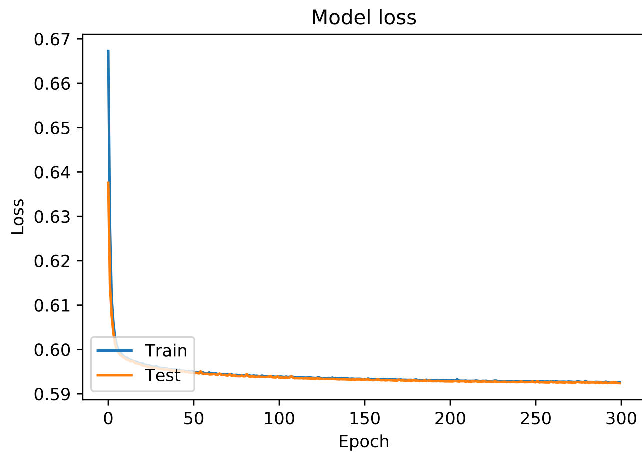
Figura 6.7: Visualización del modelo con capa `Conv2DTranspose`

Nuestra siguiente propuesta incluye una nueva capa que puede ser clave en el desarrollo de, concretamente, este tipo de arquitectura: la capa `Conv2DTranspose`, o una capa de convolución traspuesta, indicada, ahora en morado, en la figura 6.7.

A la hora de comprimir las imágenes, utilizamos la capa `Conv2D`, que genera mapas de características basados en la matriz de convolución que se dé en la entrada, tal y como mencionamos anteriormente. En nuestra arquitectura no solo comprimimos, sino que también realizamos el proceso inverso. Tiene sentido que para dicho proceso de *descompresión* se efectúen todos los pasos inversos a la compresión. Ahí es donde entran en juego las capas de convolución traspuestas.

6.4.1. Análisis de resultados

La figura 6.8b nos muestra que el resultado no ha sido tan bueno como se esperaba. El error alcanzado en esta iteración es muy similar, 0.5926 en esta en comparación con 0.5947 de la segunda iteración. Esto se refleja en la imagen reconstruida en la salida, no se aleja mucho de la versión de la arquitectura con la función de activación lineal 6.3b, que ya vimos no era ideal.



(a) Función de pérdida

(b) Ejemplo de una de las predicciones

Figura 6.8: Resultados del modelo usando `Conv2DTranspose`

Iteración	Valor de pérdida
1	0.6673
50	0.5951
100	0.5939
150	0.5933
200	0.5930
250	0.5928
300	0.5926

Cuadro 6.4: Evolución del cuarto modelo en distintas iteraciones

6.4.2. Conclusiones

Tomadas estas decisiones, se puede inferir que la solución no está en los parámetros que hemos estado modificando, sino probablemente la magnitud de nuestra red. Si generamos un resumen de la misma, podemos ver que disponemos de alrededor de 90000 parámetros entrenables, con solo 6 capas, sin contar la capa central. Un total de siete. Dichas capas además son relativamente pequeñas, ya que se usan mapas de convolución de como mucho 16 dimensiones, disminuyendo a ocho y cuatro.

En las siguientes iteraciones abordaremos este problema.

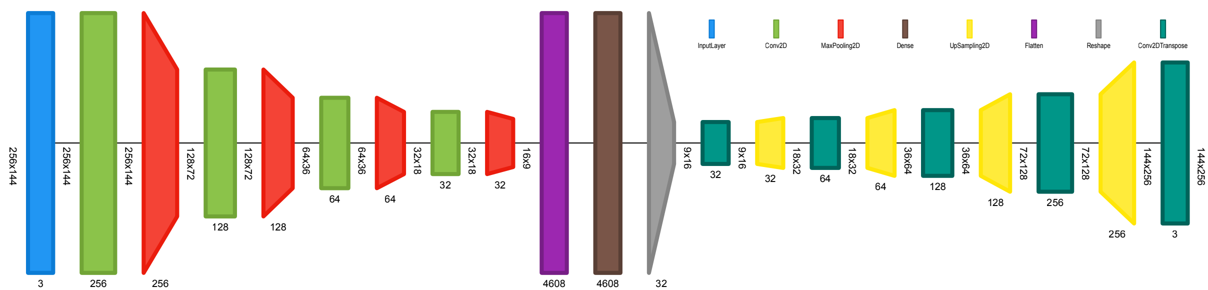


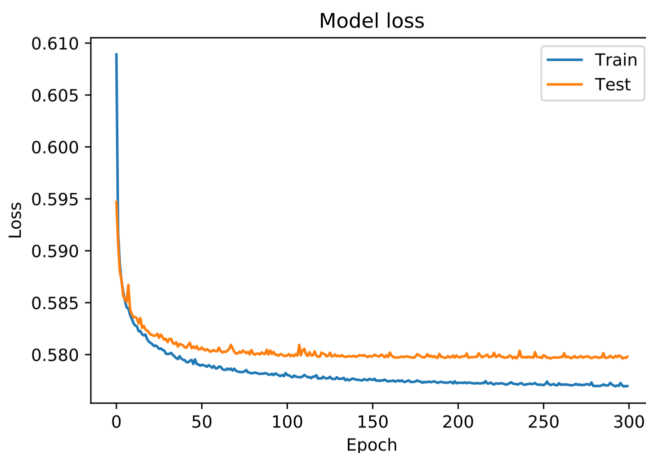
Figura 6.9: Visualización del último modelo

6.5. Quinta iteración: Aumento de magnitud

En esta iteración simplemente añadimos una magnitud de las capas de convolución mucho más grande, haciéndolas mucho más densas, ya que, hasta ahora nos hemos encontrado con que la red parece estar haciendo el trabajo correcto, pero que falla en los medios de los que dispone para ello, como se aprecia en la figura 6.9.

Las capas ahora crean 256, 128, 64 y 32 mapas de convolución respectivamente, conservando la función sigmoide del *baseline* y las capas convolucionales traspuestas. Esto añade una capa más en cada parte del autoencoder y aumenta la magnitud de reproducción de cada capa significativamente.

Esta red tiene alrededor de 22 millones de parámetros entrenables, lo cual es bastante representativo del sustancial cambio que estamos haciendo. Sin embargo, un gran poder conlleva una gran responsabilidad, ya que este modelo tardó en entrenarse al completo alrededor de unas 20 horas, cuando todos los vistos hasta ahora tomaron, como mucho, tres.



(a) Función de pérdida

(b) Ejemplo de una de las predicciones

Figura 6.10: Resultados del modelo con capas de convolución de mayor magnitud

6.5.1. Análisis de resultados

Iteración	Valor de pérdida
1	0.6089
50	0.5789
100	0.5782
150	0.5776
200	0.5772
250	0.5771
300	0.5770

Cuadro 6.5: Evolución del quinto modelo en distintas iteraciones

Es claramente apreciable la radical diferencia. La inversión de tiempo extra ha realmente supuesto un cambio sustancial en la precisión del modelo, ya que la imagen reproducida es casi idéntica a la original. Se pierden algunos detalles en las plantas que, probablemente, más se muevan o sean más sensibles a cambios. Pero podemos decir que el resultado de esta arquitectura y modelo ya son suficientes para el siguiente paso.

Es notable, sin embargo, que el valor de función de pérdida en la última iteración ha variado en 0.02 con respecto a la iteración anterior, pasamos de 0.5926 a 0.5770. Para un cambio tan radical se esperaba también una mejoría en la función de pérdida. Analizaremos la causa de esto en la siguiente sección.

6.6. Última iteración: Función de pérdida

En esta última iteración se trata de refinar algunos de los parámetros que no se han contemplado en los anteriores experimentos, para pulir el modelo lo máximo posible de cara a la reconstrucción de las fotografías y al más profundo análisis de los resultados que se expondrá en las siguientes secciones.

Investigando un poco más acerca de funciones de pérdida y optimizadores, intenté averiguar la razón por la que se utilizaba *binary crossentropy* en la implementación propuesta por la página de Keras, mencionada en la sección 6.1. *Binary crossentropy*, o BCE, es una función de pérdida enfocada al análisis de variables categóricas, diseñada en dos partes que se excluyen si cualquiera de los términos es 0, dejando así como resultado únicamente el término que no lo sea.

$$CE = -\sum_{i=1}^{C=2} t_i log(s_i) = -t_1 log(s_1) - (1 - t_1) log(1 - s_1) \quad (6.1)$$

Ecuación 6.1: Función de pérdida Binary Crossentropy

Como se puede ver en 6.1, donde t_i es la respuesta correcta, o el valor de la etiqueta para la muestra i, y s_i es la predicción del modelo para dicha muestra, se divide la ecuación en los términos que a cada una le corresponde. Al ser binaria, se asume que el conjunto de posibles respuestas C tiene una cardinalidad de 2, por lo que se puede extender el sumatorio de esa forma.

Un requisito para que esta ecuación se use en el entorno correcto es que los valores de entrada deben estar en el rango $[0, 1]$. Esto, en nuestro caso, se cumple, ya que normalizamos los valores de los píxeles en la fase de preprocesamiento, dividiendo entre 255, que es el máximo valor para los canales de 8 bits.

Pero, considerando la naturaleza de nuestro problema, la idea es averiguar la diferencia entre la imagen de entrada y la de salida. El concepto de *diferencia* en nuestro caso puede verse como una función de error basada en distancia, como el error cuadrático medio, o MSE (Mean Squared Error), cuya fórmula se expone en la ecuación 7.1 (pág. 78). Si bien BCE puede comparar los píxeles para ofrecer un valor acerca de cómo de lejos ha estado la red en cuanto a su predicción, el valor de pérdida no es idóneo ni muy representativo. Una función de pérdida como MSE puede ser mucho más apropiada para nuestro caso.

Para finalizar, exponiendo un ejemplo en el que dada una imagen de entrada en la que uno de los píxeles tenga un valor de 0.5, en nuestro caso t_i = 0.5. Suponiendo que la red, en la imagen que ofrece a la salida, ha predicho bien y ha devuelto un valor de 0.5 en dicho píxel, tenemos que s_i = 0.5. Si sustituimos los valores en la ecuación, obtenemos un valor de 0.30102. Con MSE, el error sería claramente 0, lo cual tiene mucho más sentido, ya que queremos averiguar cuál es la diferencia de la imagen de entrada con la de salida.

En otro orden de cosas, aunque también importantes, el recorte de las fotografías se ha cambiado para que se tome la parte inferior, donde mucho más probablemente estarán los animales, además eliminando la barra negra superior que todas las fotografías tienen reproduciendo información de la fecha, hora y temperatura. La forma de la imagen es la misma, solo que se extrae un fragmento diferente, más apropiado.

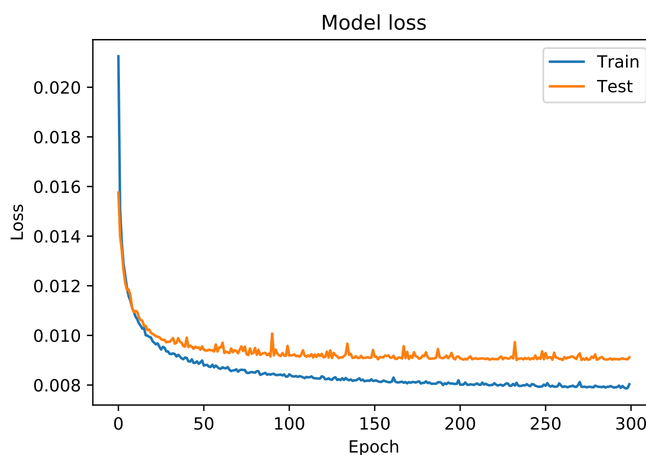
6.6.1. Análisis de los resultados

Como cabía esperar, aunque los resultados no son radicalmente mejores que en la iteración anterior, son mejores, que nunca está de más.

Iteración	Valor de pérdida
1	0.0212
50	0.0090
100	0.0083
150	0.0082
200	0.0082
250	0.0080
300	0.0080

Cuadro 6.6: Evolución del último modelo en distintas iteraciones

La escala de la gráfica ahora ha cambiado, aunque su naturaleza es la misma. Tiende a estabilizarse en las iteraciones 100 - 150.



(a) Función de pérdida

(b) Ejemplo de una de las predicciones

Figura 6.11: Resultados del modelo final compilado con MSE

Con este modelo entrenado ya disponible, podemos empezar a abarcar la tarea de clasificar. Gracias a los resultados que se nos ofrezcan, la idea es encontrar una manera en la que se pueda distinguir, con un margen de error, si una imagen presenta un animal o no. Estableceremos los métodos y pasos a seguir en el siguiente capítulo.

6.7. Modelo basado en CV

Finalmente, exponemos en esta sección los métodos de trabajo utilizados con visión artificial para resolver nuestro problema, haciendo uso de las técnicas vistas en capítulos anteriores.

6.7.1. Flujo de trabajo

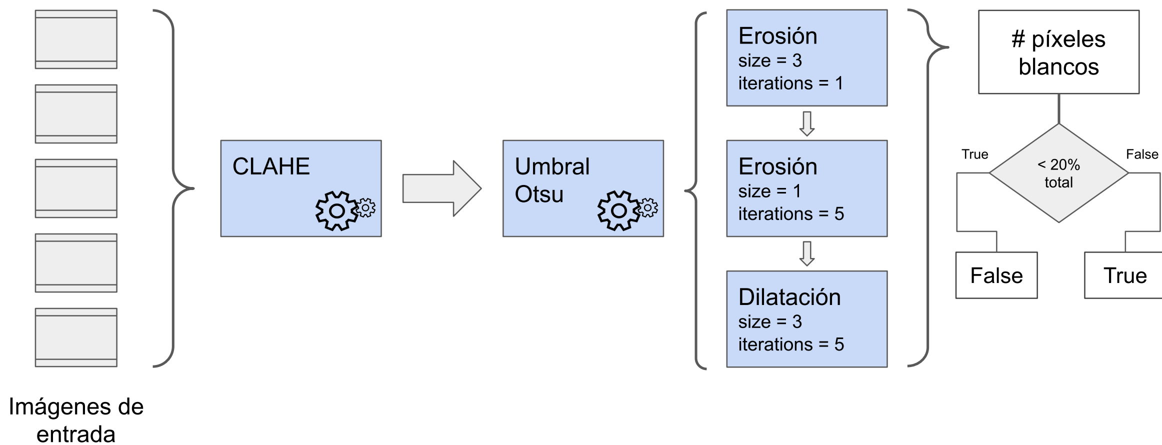


Figura 6.12: Diagrama del flujo de trabajo

En la figura 6.12 se ha ilustrado un ejemplo gráfico para ayudar a entenderlo.

Las imágenes de entrada se pasan primero al módulo CLAHE, que, por defecto, tiene un límite de contraste de 3.0 y un tamaño de cuadrícula de 128 $\times$ 128. Esto aumenta el contraste de la imagen y la prepara para la umbralización.

Seguidamente, usamos el método de Otsu para umbralizar la imagen, que al tener un alto contraste, suele funcionar bien.

En este punto del proceso ya tenemos una imagen binaria, así que primero erosionamos con un *kernel* mediano para eliminar grandes partes de blanco que puedan ser molestas, y luego con un *kernel* más pequeño, de sólo 1 píxel, eliminando el ruido y cualquier otro pequeño resto que al primer filtro de erosión pudiera haber escapado.

Seguidamente, dilatamos los píxeles blancos para consolidar mejor la forma del animal, si es que la hubiera. Si no la hay, este módulo efectivamente no hace nada.

A partir de aquí, contamos el número de píxeles blancos. Si dicho número cae por debajo de un umbral, decimos que la imagen está vacía, en otro caso, hay un animal. La idea es que tras el proceso, los animales hayan quedado retratados con formas blancas presentes en la imagen, y las imágenes vacías serían prácticamente negras. Se establece un umbral aún así, ya que siempre puede quedar algún que otro píxel blanco que no fuera nada representativo, pero que igualmente escaparía al condicional.

En la siguientes secciones analizaremos y evaluaremos cómo de efectivos son cada uno de los métodos y estableceremos una comparativa.

Capítulo 7

Interpretación y evaluación de resultados

En este capítulo haremos uso de los modelos entrenados que tenemos disponibles y diferentes herramientas de comparación de imágenes para poder discernir si una fotografía tiene un animal o no, con un porcentaje de acierto lo más alto posible. Se presentarán todos los pasos previos de preparación y se expondrán los resultados para concluir si este método funciona o no.

7.1. Resultados del modelo de DL

Dedicaremos esta sección a analizar los resultados del AE entrenado y ver cómo podemos extraer la información de los mismos.

7.1.1. Preparación de los nuevos datos

En esta sección discutiremos los pasos previos necesarios para realizar la comparativa y evaluación. Generaremos un conjunto de imágenes que ya sí incluyen animales, etiquetas para las imágenes y pasaremos a evaluar el rendimiento siguiendo diferentes métodos.

Nuevo dataset

Para entrenar la red, nos centramos exclusivamente en el procesamiento, análisis y comprensión de las imágenes vacías. Ahora que tenemos un buen conocimiento de

sus patrones, incluiremos fotografías de animales, para poder evaluar el rendimiento del sistema.

Primero debemos encontrar las fotografías con animales para esta localización. Siguiendo el mismo exacto proceso que se siguió en la subsección 5.2.1, encontramos aquellas imágenes cuya etiqueta **no** sea vacía, y las agrupamos por localización.

Un aspecto que salta a la vista es la diferencia de número que cada carpeta posee. La localización 96, que es la que estamos tratando, tiene alrededor de 37000 fotografías catalogadas como vacías, y sólo algo más de 1000 catalogadas con algún animal presente. Se puede ver aquí, muy claramente, la magnitud del problema que queremos resolver. Filtrar manualmente 37000 imágenes de un conjunto de un total de 38000 es una tarea básicamente inviable.

Con las imágenes de animales y vacías disponibles, se creó un nuevo dataset mixto. Exactamente tenemos 1078 fotografías con animales, así que se tomaron exactamente 1078 imágenes vacías para que fuera más equilibrado el análisis. Estas 1078 imágenes vacías se tomaron aleatoriamente del conjunto de las mismas.

7.1.2. Comparativa de imágenes

La principal idea que subyace detrás de todo este sistema es la capacidad de reconstrucción de nuestra red neuronal. Como ya hemos visto en la sección anterior, hemos ido iterando el modelo hasta conseguir que la imagen de salida se parezca lo máximo posible a la de entrada. Esto es lo que distingue al AE: dicha reconstrucción solo será buena si la entrada coincide con los patrones que ha condensado en sus capas internas. Si algo no se corresponde, la imagen de salida ofrecerá un muy mal resultado, y ahí es donde tenemos que fijarnos.

En el ejemplo expuesto en la figura 7.1 se puede ver claramente el objetivo. Ahora podemos comparar cómo de diferentes son las imágenes y estimar, en base a ello, la presencia de un animal.

Error cuadrático medio, MSE

Uno de los métodos más básicos para comparar la similitud entre dos imágenes es el error cuadrático medio. Toda la teoría detrás de este método se apoya en la operación *diferencia*, que nos da una medida de cómo de lejos dos valores están. La diferencia o resta también suele denominarse distancia por esta misma razón. Como la diferencia en sí suele ser sensible a ruido en la señal, se eleva al cuadrado para eliminar signos negativos, así como para saturar las diferencias más grandes y no tanto las diferencias pequeñas.

(a) Reconstrucción correcta (b) Reconstrucción incorrecta

Figura 7.1: Ejemplo de los resultados obtenidos con una imagen vacía y una con un ciervo. Se puede apreciar cómo en la reconstrucción con el ciervo presente no se distingue nada, ni el ciervo ni la escena original.

En nuestro problema, este método se aplica píxel a píxel:

$$MSE = \frac{1}{m \cdot n} \sum_{i,j=0}^{i=m,j=n} (P_{i,j} - P'_{i,j})^2 \tag{7.1}$$

Ecuación 7.1: Cálculo del MSE para un par de imágenes

donde P es la imagen de entrada, original y P' es la imagen predicha por el modelo, i y j son las coordenadas de los diferentes píxeles, m es el número de filas de la imagen y n, el de columnas. $P_{i,j}$ es el pixel (i, j) de la imagen P.

Esto nos da una aproximación de cómo de diferentes son las imágenes. Aun así, es un método demasiado básico y no suficientemente representativo como para seguirlo a ciegas.

Índice de similitud estructural, SSIM

El SSIM (Structural Similarity Index) [49] es un cálculo que trata de resolver todos los problemas con los que nos encontramos utilizando MSE para averiguar la diferencia entre dos imágenes. Se pueden estudiar al detalle sus fundamentos matemáticos e implementación en la referencia provista, ya que entrar en detalle acerca de la elaboración de este algoritmo escapa al ámbito de este trabajo.

Este proceso tiene en cuenta la estructura de la imagen, como su nombre indica, y hace una evaluación a nivel mucho más global que el MSE, cuya evaluación es local para cada píxel.

Este índice puede aportarnos más información y más útil de cara a nuestro problema. Dados estos datos, nuestro objetivo es encontrar un valor de MSE bajo y un SSIM alto. Esto es lo que define a una foto vacía. La pregunta inversa también es válida.

7.1.3. Selección de un umbral

MSE	SSIM	Tag
63.655816	47.456846	1
105.411323	41.589356	1
79.244656	43.246608	1
81.933648	69.382547	0
82.625814	64.246800	0

Cuadro 7.1: Ejemplo de 5 muestras de los datos obtenidos para nuestro conjunto de imágenes mixto. Se ejecutaron las predicciones en el modelo y, con las imágenes de salida, se calcularon los factores de MSE y SSIM.

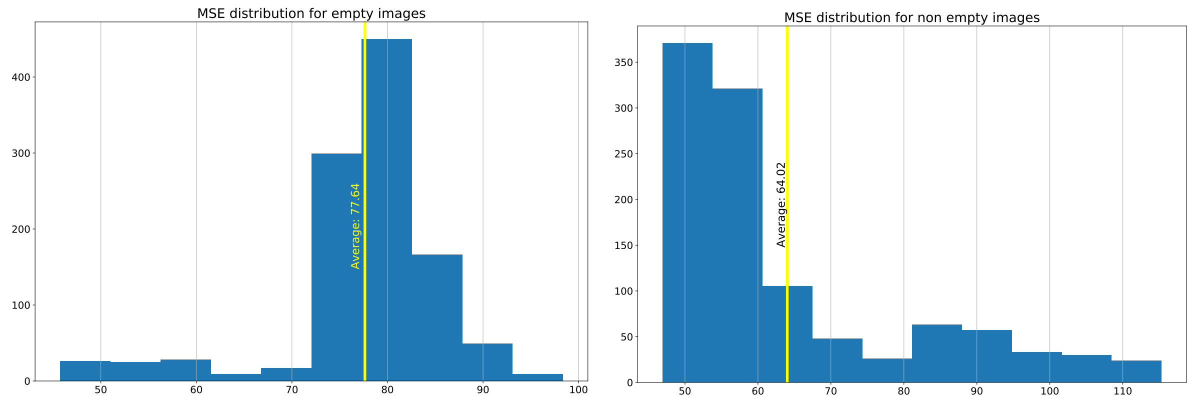
Selección manual

A partir de este punto, tenemos un conjunto de datos de 2156 filas con el formato que vemos en el extracto del cuadro 7.1. Para cada imagen hemos generado sus correspondientes errores y su etiqueta. A partir de aquí, la idea es determinar qué valores hacen que la etiqueta sea 0 o 1 y, consecuentemente, nos informen acerca de la presencia o ausencia de un animal en ella.

Para establecer manualmente este umbral, debemos hacer una exploración exhaustiva de los datos, y no hay mejor herramienta para ello que las gráficas.

En los histogramas de las figuras 7.2 y 7.3 podemos apreciar la distribución del número de fotografías cuyo error cae en dicho rango. Esto nos da una vista rápida de dónde se hallan la mayoría de las fotografías. Lo ideal sería encontrar un valor a partir del cual los dos histogramas se separasen lo suficiente como para poder considerarlo determinante.

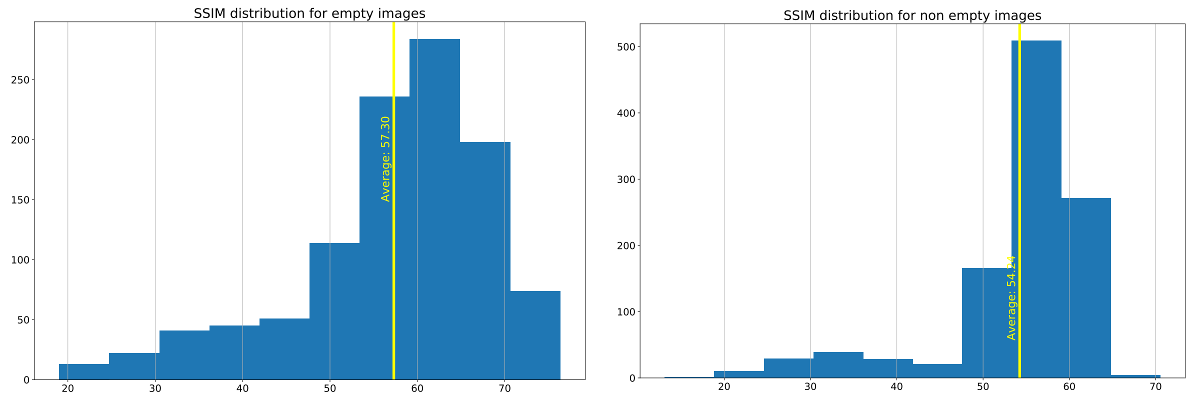
Como podemos apreciar, todos los histogramas, a excepción del 7.3b, están bastante distribuidos a lo largo del espectro. Esto son malas noticias, porque indica que



(a) Histograma de distribución del MSE de las imágenes vacías

(b) Histograma de distribución del MSE de las imágenes no vacías

Figura 7.2: Resumen de los histogramas del MSE



(a) Histograma de distribución del SSIM de las imágenes vacías

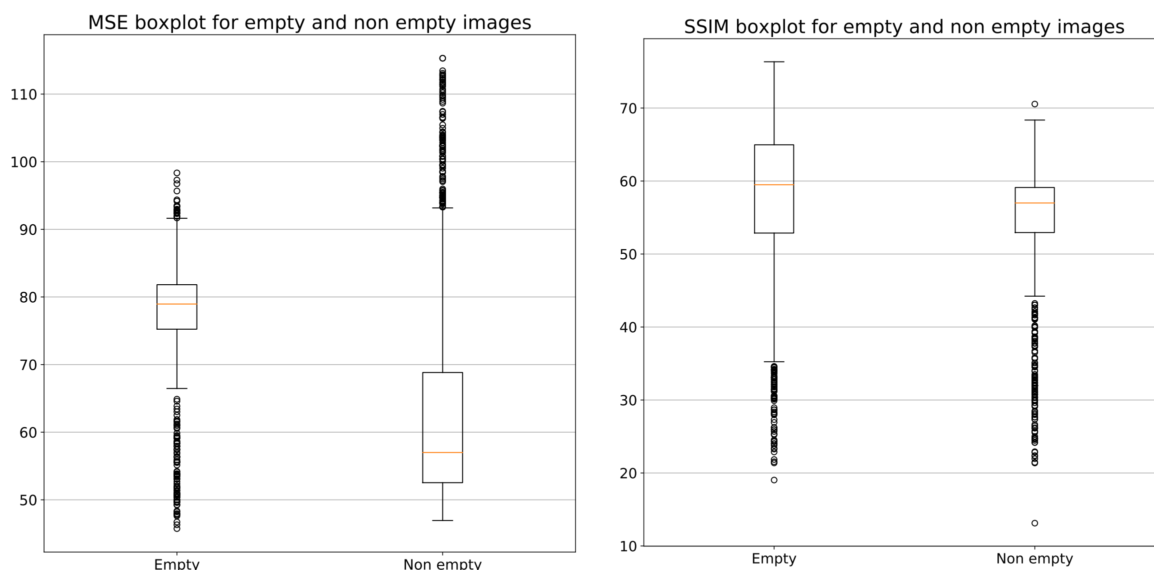
(b) Histograma de distribución del SSIM de las imágenes no vacías

Figura 7.3: Resumen de los histogramas del SSIM

la valoración hecha por los métodos utilizados no es suficientemente determinante ni robusta.

Para obtener una perspectiva alternativa y confirmar los puntos anteriormente mencionados, podemos ver la distribución de los datos en diagramas de caja en la figura 7.4. En muchos de los casos, los valores son considerados como atípicos por salirse de la norma del factor del 1.5 del valor intercuartílico. Pero si nos fijamos en la anchura de las cajas y los bigotes, podemos ver una distribución muy similar a la presente en los histogramas.

A raíz de estas imágenes, una posible solución sería evaluar la función umbral a partir del mínimo valor del rango que más fotografías contiene, como el valor aproximado de 53 que puede apreciarse en 7.3b. Es decir, si el valor de SSIM de la imagen es menor o igual a un 53 %, la imagen no se ha reconstruido lo suficientemente bien, por lo que inferimos que es una fotografía de un animal.



(a) Diagrama de caja para la distribución del MSE

(b) Diagrama de caja para la distribución del SSIM

Figura 7.4: Resumen de los diagramas de caja y bigotes

Pero la decisión de este umbral es muy pobre, ya que no tenemos en cuenta que, aunque el SSIM es más preciso y elaborado que el MSE, no es perfecto.

Fotografías nocturnas

Un aspecto importante a tener en cuenta es la naturaleza de las fotografías. De las 37000 que poseemos vacías, fácilmente más de un 90 % fueron hechas de día. De las fotografías precatalogadas como contenedoras de algún animal, cerca de un 90 % fueron hechas por la noche.

Este factor es tremendamente importante a la hora de calcular el error de una fotografía.

La razón es sencilla: las zonas oscuras de las esquinas superiores, como se puede apreciar en la figura 7.5, serán muy similares en la reconstrucción, ya que son todo valores muy cercanos a negro. Esto baja mucho el MSE y sube el SSIM, porque los valores en ambas imágenes son, en realidad, muy parecidos. Además, el AE tiene mucha más *práctica* codificando imágenes diurnas, así que las fotos nocturnas puede que se reconstruyan simplemente peor, en el mejor de los casos.

Esta es la razón por la que vemos el comportamiento tan aparentemente anómalo en el histograma 7.2b. Se supone que si estamos en la categoría de imágenes no vacías, el error debería ser alto al haber presencia de animales y ser la reconstrucción

Figura 7.5: Ejemplo de una fotografía vacía por la noche

muy pobre. Pero el histograma muestra un valor medio radicalmente más bajo que en las imágenes no vacías.

Esto es simplemente debido a las razones expuestas anteriormente: hay muchísimas más imágenes nocturnas que diurnas de animales, y las imágenes nocturnas se reconstruyen *mejor*, siendo muy estrictos. Lo que nos indica ese histograma en realidad es que un gran porcentaje de imágenes son nocturas, no que las imágenes con animales se reconstruyan mucho mejor que las vacías.

Teniendo en cuenta estos diagramas e hipótesis, la conclusión es que determinar un umbral definitivo en base a estos datos es algo bastante poco apropiado. Los datos no nos aportan suficiente consistencia como para poder determinar dicho umbral. La solución que se propone en la siguiente sección es utilizar ambos valores en vez de uno solo, ya que en realidad los dos aportan información. Calcular el factor común que nos dice qué valores debe tener una fotografía para pertenecer a una categoría o a otra puede ser una muy efectiva alternativa.

Selección automática: árboles de decisión

Dados estos dos valores, nuestro trabajo es encontrar el patrón que define que una imagen tenga un animal o no. Averiguar patrones en números es un trabajo que un ordenador puede ejecutar de forma espléndida. Utilizando árboles de decisión, podemos encontrar dicho patrón y predecir la presencia de un animal.

(a) **MSE**: 85.56. **SSIM**: 64.95 % (b) **MSE**: 95.78. **SSIM**: 22.19 %

Figura 7.6: Ejemplo de los resultados obtenidos y sus correspondientes índices.

Un árbol de decisión [50] trata de generar una predicción Y basándonos en una serie de características $X_1, X_2, \ldots, X_n$. En cada nodo aplicamos una evaluación de los parámetros que en dicho nodo estén presentes, por ejemplo el parámetro X_i. Dependiendo del resultado, nos moveremos a la izquierda o a la derecha del árbol. Iterando sobre este proceso, llegaremos a un nodo hoja, donde efectuaremos una predicción. Esta predicción agrega todos los valores de entrenamiento que alcanzan dicha hoja.

El proceso básico de construcción de un árbol de decisión es el siguiente:

- Seleccionar el mejor atributo siguiendo el proceso de Medida de Selección de Atributos (ASM for Attribute Selection Measures) [51].
- Genera un nodo de decisión de ese atributo y particiona el conjunto de datos en dos subconjuntos de menor tamaño.
- Se construye el árbol repitiendo este proceso recursivamente para cada nodo hijo hasta que todos las muestras pertenezcan a la misma categoría, nos quedemos sin categorías en las que dividir o nos quedemos sin muestras.

Este proceso nos ahorra tener que escoger un umbral manual a partir del cual podríamos considerar que existe un animal o no. Dicho umbral además sería muy poco representativo y probablemente tuviera una tasa de acierto inutilizable, como ya se analizó en la sección anterior.

En nuestro caso, tenemos los atributos MSE y SSIM y una etiqueta, con un 1 si existe animal o un 0 si no lo hubiera. Utilizando la librería *sklearn*, que posee una

implementación de la mayoría de algoritmos de ML, podemos crear un árbol de decisión y entrenarlo de forma casi instantánea.

Un valor a tener en cuenta en la optimización de hiperparámetros de este modelo es la profundidad. Si no se especifica una profundidad, los nodos se expanden hasta que se cumplan las condiciones mencionadas, pero esto puede causar un severo caso de *overfitting* en el modelo. Para ello podemos limitar la profundidad del árbol.

7.1.4. Flujo final de procesamiento

En esta subsección haremos un breve resumen del flujo de procesamiento que se ha acometido en todo el proyecto para llegar al resultado que tenemos actualmente. La figura 7.7 muestra un breve resumen gráfico del mismo.

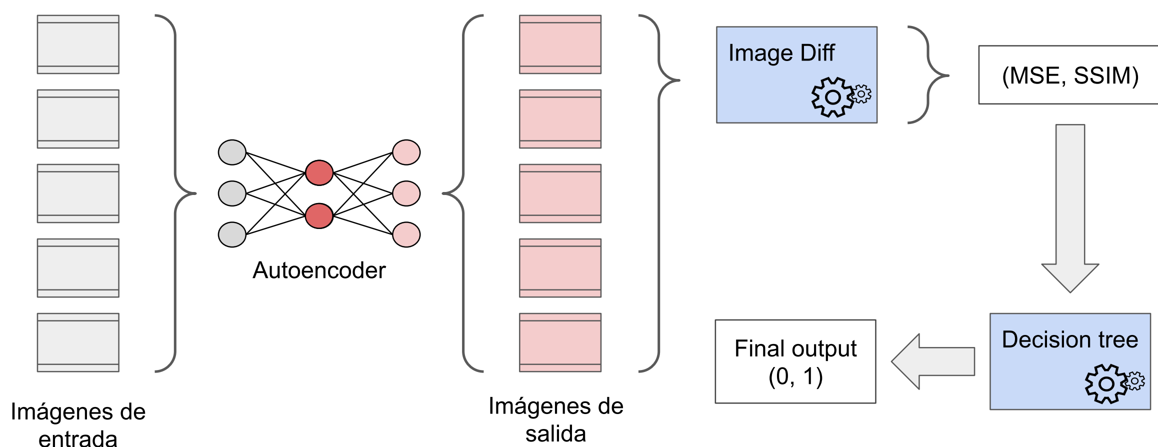


Figura 7.7: Gráfico del flujo final del proyecto [Elaboración propia]

En primer lugar, obtenemos el dataset de la fuente que sea, en nuestro caso, del *Caltech Camera Trap Image Dataset*. Dichas imágenes vienen ya previamente etiquetadas, al ser un conjunto de datos de entrenamiento.

Las imágenes se preprocesan, haciéndolas más pequeñas, normalizando sus valores y tomando la sección que nos interesa.

Posteriormente, se pasan al AE. Ya entrenado, es capaz de reconstruir las imágenes vacías de forma bastante precisa. Este nos devuelve la predicción que ha hecho para cada imagen.

Los pares de imágenes original-predicción se pasan al módulo *Image Diff*, que calcula el MSE y el SSIM de dicho par de imágenes. Esto nos devuelve un conjunto de pares MSE, SSIM para cada imagen de entrada.

Para finalizar, esos dos parámetros son la entrada del árbol de decisión, que determina en base a ellos si hay un animal o no. Finalmente, se nos devuelve dicho valor.

7.1.5. Resultados

Límite de niveles	Precisión
0 (sin límite)	88 %
1	84 %
2	89 %
5	90 %
7	89 %

Cuadro 7.2: Muestra de evolución de la precisión con respecto al límite de niveles en el conjunto de evaluación. Se aprecia que el mejor resultado se encuentra en 5 niveles.

En el cuadro 7.2 vemos la evolución del árbol con diferentes profundidades. Estas pruebas se hicieron sobre el conjunto de datos mencionado en la tabla 7.1. De esas 2156 entradas se tomó un 70 % para el conjunto de datos de entrenamiento/validación del árbol y un 30 % para el conjunto de evaluación.

Con cinco niveles, el árbol de decisión acierta un 90 % de las veces, lo cual es un resultado reseñable. Haber ajustado el parámetro de niveles ha sido una gran idea, ya que, aunque pudiera parecer que no ofrecerle límite le dejaría más libertad para generar otros posibles condicionales, como vemos realmente sufre de *overfitting*, ya que al exponerlo a los datos de evaluación, que son los que se muestran en el cuadro, baja en precisión, indicativo de que no generaliza bien.

7.2. Resultados del algoritmo de CV

Resumiremos aquí los resultados del algoritmo de visión artificial para poder hacer una mejor comparativa entre los mismos.

7.2.1. Resultados

Para evaluar los resultados, se probaron distintos umbrales para poder apreciar la evolución de la precisión. Concretamente, los umbrales utilizados son 10, 15, 20, 25 y 30.

Como se puede ver en la figura 7.8, el sistema funciona sorprendentemente bien, alcanzando más de un 90 % de precisión en los umbrales 10 y 15. El conjunto de datos utilizado fue creado utilizando los mismos procedimientos que con el dataset de la red neuronal, pero no se garantiza que sean las mismas fotografías exactamente ya que el listado de archivos de las funciones internas del lenguaje es arbitrario, no sigue un orden específico. Aún así, siguen siendo 2156 fotografías de las cuales 1078 están precatalogadas como contenedoras de animales, y el resto, como vacías.

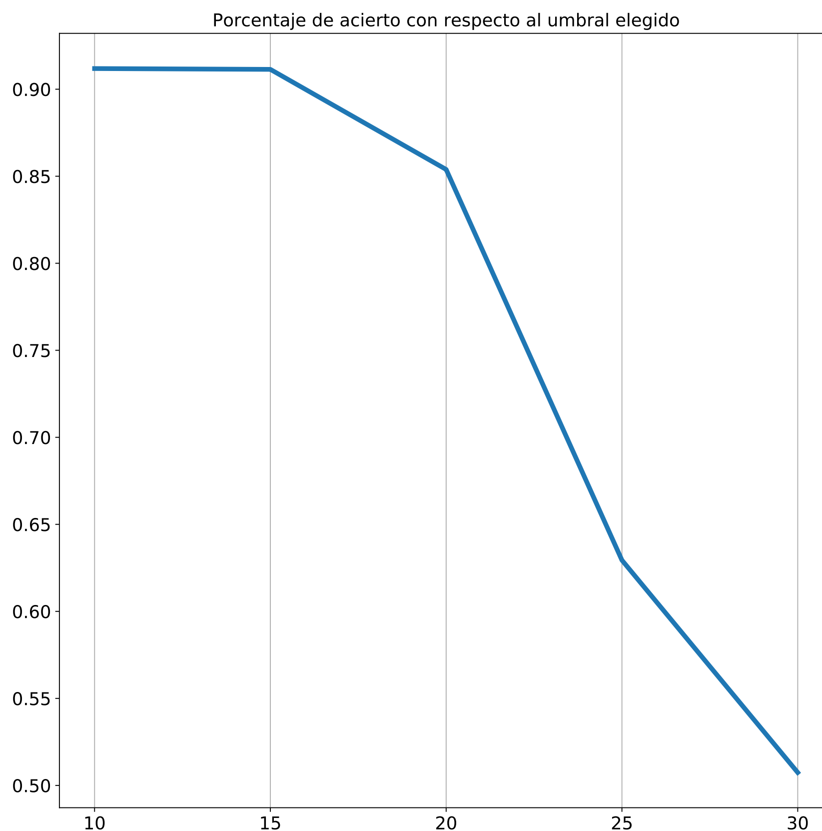


Figura 7.8: Evolución de la precisión en función del umbral

Conforme aumentamos el umbral, la curva desciende rápidamente. Llegados a 30 % o superando esa cifra, el comportamiento del algoritmo se resume en decir que cualquier fotografía tiene un animal, por lo que el mínimo teórico que se puede alcanzar es un 50 %, ya que el dataset creado para esta prueba posee una proporción 50/50 de fotografías vacías/no vacías.

En la figura 7.9 tenemos un resumen de la gráfica 7.8 pero detallada en cada iteración con matrices de confusión. El valor ideal de estas gráficas, como referencia, sería obtener un valor de 1078 en las dos casillas de la diagonal principal y 0 en el resto.

Las dos primeras matrices de confusión, 7.9a y 7.9b nos muestran lo cerca que ha estado el algoritmo de este valor ideal. También se aprecia la clara tendencia a los falsos positivos, no teniendo demasiado problema en los falsos negativos. La última matriz 7.9e nos enseña que pasado dicho umbral, el único veredicto de salida es cata-

logar la foto como no vacía, de ahí que aparentemente tenga un 50 % de probabilidad de acierto. Estrictamente hablando ha acertado la mitad de las fotos, pero, evidentemente, el funcionamiento del algoritmo a partir de ese punto es completamente inútil.

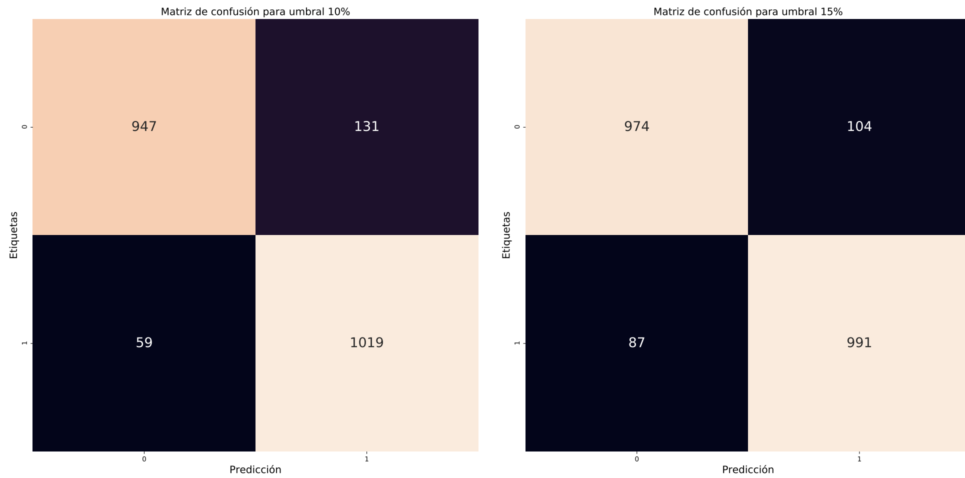
7.2.2. Observaciones y conclusiones

Para finalizar, se compartirán en esta subsección algunas observaciones recogidas a raíz de experimentar con estas técnicas que merecen ser mencionadas.

En algunos casos, plantas que estén en primer plano y generen grupos de píxeles blancos pasarán el umbral, categorizándose la fotografía como no vacía, pero es algo de lo que prácticamente no podemos encargarnos.

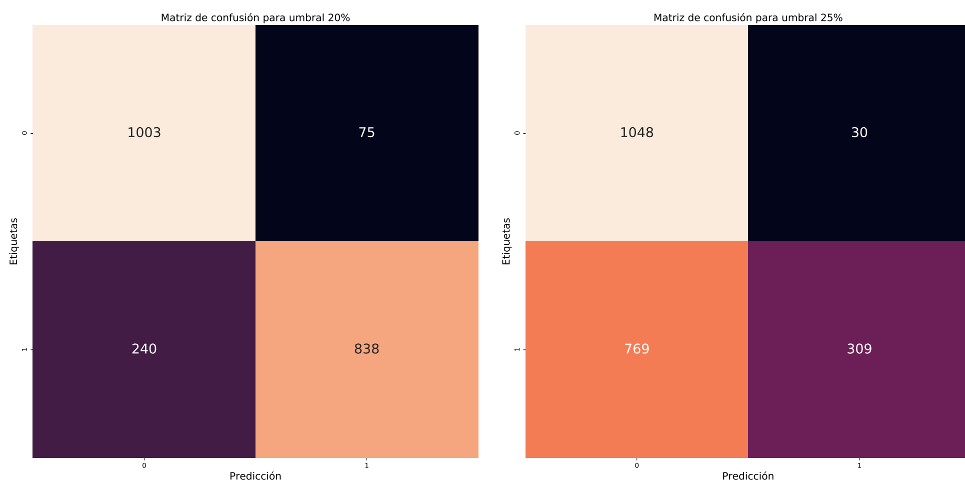
También, a veces algunos animales no son captados del todo por la cámara, o están tan lejos que no son correctamente iluminados, por lo que los píxeles que los representan son realmente similares a los de su entorno. Dichas imágenes muy probablemente sean catalogadas como vacías incorrectamente.

Estos factores y probablemente otros que no haya sido capaz de detectar contribuirían al mal funcionamiento del algoritmo, pero también se está hablando de casos particularmente extremos y poco favorables. Aún así, parece haber alcanzado unos resultados bastante similares a los de las redes neuronales, lo cual es, cuanto menos, sorprendente.



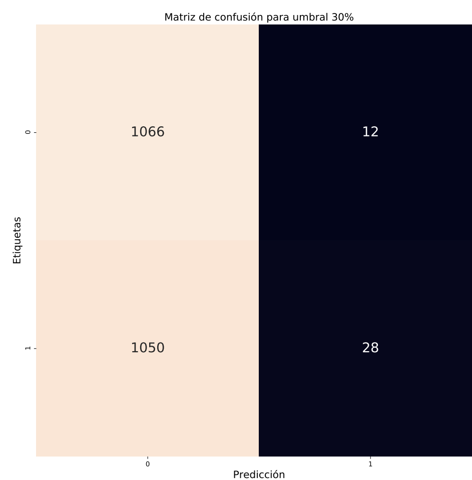
(a) Matriz de confusión para umbral de 10 %

(b) Matriz de confusión para umbral de 15 %



(c) Matriz de confusión para umbral de 20 %

(d) Matriz de confusión para umbral de 25 %



(e) Matriz de confusión para umbral de 30 %

Figura 7.9: Diferentes matrices de confusión en función del umbral

Capítulo 8

Conclusiones, comparativa y propuestas

En este capítulo discutiremos acerca de los resultados obtenidos, los experimentos realizados y se añade una pequeña reflexión personal. Finalmente, se exponen algunas posibles propuestas o mejoras que podrían acometerse para perfeccionar el resultado en el futuro.

A lo largo del proceso de experimentación hemos logrado incrementar enormemente la calidad y precisión del AE, teniendo en cuenta los resultados iniciales en comparación con los finales. Gran parte de la mejora vino por el aumento de magnitud, y esto es muy probablemente extrapolable a una red con más capas y que fuera entrenada en el conjunto de datos completo.

A raíz de ahí, todo era cuestión de elaborar métodos de cálculo de diferencia de imágenes. Aún así, este *dataset* no era el más apropiado, porque un gran porcentaje de las fotografías con animales eran nocturnas y no había muchas en comparación. Los demás conjuntos de imágenes sufrían de problemas similares.

Afortunadamente, la obtención de dos cálculos diferentes para cada imagen en conjunción con la inclusión de un árbol de decisión ha parecido ser la clave del éxito. Alrededor de un 90 % de acierto en el subconjunto de datos de evaluación es, sin duda, un resultado reseñable.

A la hora de extrapolar esto a tareas más grandes, lo ideal sería entrenar un modelo más profundo y sobre todo el conjunto. Así, la capacidad de reproducción del mismo sería muchísimo más alta y los valores de MSE y SSIM, consecuentemente, más representativos. Es una tarea que, sin embargo, se escapa de nuestra capacidad de cómputo, además de que carece de sentido cuando el conjunto de experimentación ya nos ha aportado prácticamente toda la información que necesitábamos.

El AE ha evolucionado adecuadamente conforme hemos hecho cambios, pero el proceso de entrenamiento es tedioso. Para cada entrenamiento desde el primero hasta el aumento de magnitud, la duración era de tres horas, concretamente rondaba siempre las 3 horas y 15 minutos, dependiendo del modelo. La duración del entrenamiento del modelo de mayor magnitud fue de algo más de 20 horas. Todo esto, contando con que el equipo del que se dispone posee una tarjeta gráfica.

Además, algo curioso es que los datos provistos por la red neuronal no eran directamente interpretables. Hemos tenido que hacer uso de otras técnicas de ML para poder realmente obtener una respuesta, lo cual es destacable dada la inversión de tiempo realizada.

El algoritmo de visión artificial ha realizado el exacto mismo trabajo utilizando una fracción muy pequeña del tiempo y de los recursos, obteniendo unos resultados muy similares. De hecho, el algoritmo de CV procesó todas las imágenes para cada umbral, utilizando todos los umbrales provistos, calculó las matrices de confusión y las guardó en algo más de 4 minutos. Y ni siquiera se hicieron mejoras como la paralelización del procesamiento. Estos datos aportan un poco de perspectiva acerca de cómo de exigente es entrenar una red neuronal.

Este punto nos invita a la reflexión, cuando muchas de las veces se piden ingenieros de DL porque se vuelca toda la esperanza y responsabilidad del objetivo en una red neuronal, que, en los casos profesionales, requieren de un despliegue de medios muy considerable para poder ser viables. A veces no es necesario implementar una red neuronal e, insisto, todo lo que ello conlleva.

Esto no significa que los modelos de DL deban descartarse radicalmente, pero sí que debe hacerse un buen análisis del problema y evaluar si realmente merece la pena.

8.1. Propuestas de futuro

Algunas de las propuestas más atractivas para la puesta en escena de la gran utilidad que nuestro sistema posee es la creación de un software que automáticamente analizara un gran conjunto de imágenes, o la inclusión del modelo en la arquitectura de las propias cámaras de fototrampeo.

8.1.1. Desarrollo de software

La principal idea que tenía presente a la hora de desarrollar este proyecto y poder guiarme por una motivación algo más tangible era la impementación de un software que hiciera uso de los modelos y tecnologías aquí usados.

El programa pediría al usuario que especificara la ruta de un archivo o carpeta de archivos y el programa, automáticamente, ordena los archivos de entrada en carpetas pertinentes. Me imaginaba a cualquiera de los operarios, biólogos, zoólogos o cualquier profesional que haga uso de estos dispositivos arrastrando una carpeta al programa y el programa creando una carpeta destino con todas las imágenes clasificadas pertinenemente, en cuestión de minutos. Es, desde luego, una sensación satisfactoria.

8.1.2. Implementación en las cámaras

Otra idea es que las cámaras estuviesen dotadas de dispositivos de inferencia como los que propone *Tensorflow*.

Esto es posible gracias a las tecnologías que la empresa Coral[1] ha dispuesto al público en colaboración con Google.

Figura 8.1: Placa PCIe Coral Edge TPU

Una vez entrenado el modelo y hechos todos los cálculos necesarios para alcanzar la ratio de acierto requerido, este pequeño dispositivo puede integrarse sin problema en las cámaras de fototrampeo. El modelo entrenado se instala y compila en la placa y ahora la cámara está dotada de todo el poder de inferencia que hemos desarrollado, pero directamente en acción.

A la hora de tomar una fotografía, la imagen se pasa a la placa de inferencia y ésta determinará si debe o no ser descartada, al instante.

Esta solución resuelve muchos problemas. En primer lugar, ya no se tendría que lidiar con tantísima información inútil en el laboratorio, y no tendrían que almacenarse tantas imágenes vacías a la espera de ser catalogadas. Además, esto alargaría tremendamente la vida útil de los dispositivos de almacenamiento de las cámaras, ya que no estarían llenos de fotografías vacías y siempre habría espacio para otra fotografía de un animal. Asimismo, la revisión periódica de los mismos podría disminuirse, ya que no se llenarían tan rápido.

Esto se puede mejorar incluso más si, como se mencionó en la sección 1.1.1, estas disponen de conexión a una red de algún tipo. Las cámaras podrían dejarse casi completamente desatendidas e irían tomando y enviando solo las fotografías útiles directamente al laboratorio.

[1] https://coral.ai/

Apéndice A

Planificación temporal

ESTIMACIÓN TEMPORAL

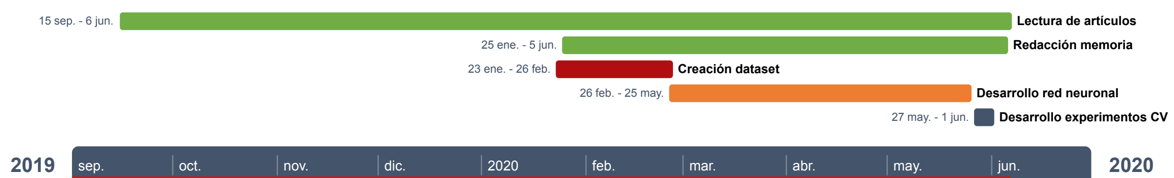


Figura A.1: Estimación temporal del proyecto.

Aquí se presenta un diagrama en el que se estima la duración de cada una de las subtareas que componen la totalidad de este proyecto. Se empezó a investigar acerca de esta disciplina a partir del momento en el que se estipuló el acuerdo con los tutores, 15 de septiembre aproximadamente. A partir de ahí, se empezó a trabajar realmente en la redacción de la memoria y el desarrollo del software paralelamente durante el segundo cuatrimestre, teniendo en cuenta también que se estaban cursando las tres últimas asignaturas del curso. Por ello, la distribución temporal es aproximada.

Apéndice B

Ficheros de código

En este apéndice se incluirá el código en python de los diferentes modelos entrenados así como aquellos otros ficheros o recursos que sean relevantes. Se incluyen aquí para no interrumpir con el flujo del documento. El código del proyecto puede encontrarse en GitHub: `https://github.com/jesi-rgb/Convolutional-Autoencoder`

```
info
{
  Required
  "version" : str,
  "description" : str,

  Optional
  "year" : int,
  "contributor" : str
  "date_created" : datetime
}
image
{
  Required
  "id" : str,
  "file_name" : str,

  Optional
  "width" : int,
  "height" : int,
  "rights_holder" : str,
  "datetime": datetime,
  "seq_id": str,
  "seq_num_frames": int,
  "frame_num": int
  "location": str,
  "corrupt": bool
}
category
{
  "id" : int,
  "name" : str
}
annotation
{
  "id" : str,
  "image_id" : str,
  "category_id" : int,
  "bbox": [x,y,width,height],
  "sequence_level_annotation" : bool
}
```

Listing 1: Ejemplo de la estructura del archivo JSON, usando el estándar COCO [1]

```python
def build_autoencoder(shape):
    input_layer = Input(shape=shape)
    x = Conv2D(16, (3, 3), activation='relu', padding='same')(input_layer)
    x = MaxPooling2D((2, 2), padding='same')(x)
    x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D((2, 2), padding='same')(x)
    x = Conv2D(4, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D((2, 2), padding='same')(x)
    x = Conv2D(4, (3, 3), activation='relu', padding='same')(x)
    encoded = MaxPooling2D((2, 2), padding='same')(x)

    x = Conv2D(4, (3, 3), activation='relu', padding='same')(encoded)
    x = UpSampling2D((2, 2))(x)
    x = Conv2D(4, (3, 3), activation='relu', padding='same')(x)
    x = UpSampling2D((2, 2))(x)
    x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
    x = UpSampling2D((2, 2))(x)
    x = Conv2D(16, (3, 3), activation='relu', padding='same')(x)
    x = UpSampling2D((2, 2))(x)
    decoded = Conv2D(shape[len(shape)-1], (3, 3), activation='sigmoid',
    padding='same')(x)

    autoencoder = Model(input_layer, decoded)
    autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

    return autoencoder
```

Listing 2: Implementación del autoencoder sacado de la página de Keras.

```python
def build_autoencoder(shape):
    input_layer = Input(shape=shape)
    x = Conv2D(16, (3, 3), activation='relu', padding='same')(input_layer)
    x = MaxPooling2D((2, 2), padding='same')(x)

    x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D((2, 2), padding='same')(x)

    x = Conv2D(4, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D((2, 2), padding='same')(x)

    encoded = Dense(2304, activation='relu')(x)

    x = Conv2D(4, (3, 3), activation='relu', padding='same')(encoded)
    x = UpSampling2D((2, 2))(x)

    x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
    x = UpSampling2D((2, 2))(x)

    x = Conv2D(16, (3, 3), activation='relu', padding='same')(x)
    x = UpSampling2D((2, 2))(x)

    decoded = Conv2D(shape[len(shape)-1], (3, 3), activation='sigmoid',
    padding='same')(x)

    autoencoder = Model(input_layer, decoded)
    autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

    return autoencoder
```

Listing 3: Implementación del autoencoder, ligeramente modificado para nuestro problema.

```python
def build_autoencoder(shape):
    input_layer = Input(shape=shape)
    x = Conv2D(16, (3, 3), activation='relu', padding='same')(input_layer)
    x = MaxPooling2D((2, 2), padding='same')(x)

    x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D((2, 2), padding='same')(x)

    x = Conv2D(4, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D((2, 2), padding='same')(x)

    encoded = Dense(2304, activation='relu')(x)

    x = Conv2D(4, (3, 3), activation='relu', padding='same')(encoded)
    x = UpSampling2D((2, 2))(x)

    x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
    x = UpSampling2D((2, 2))(x)

    x = Conv2D(16, (3, 3), activation='relu', padding='same')(x)
    x = UpSampling2D((2, 2))(x)

    decoded = Conv2D(shape[len(shape)-1], (3, 3), activation='linear',
    padding='same')(x)

    autoencoder = Model(input_layer, decoded)
    autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

    return autoencoder
```

Listing 4: Implementación del autoencoder con función de activación linear.

```python
def build_autoencoder(shape):
    input_layer = Input(shape=(shape[1], shape[2], 1))
    x = Conv2D(16, (3, 3), activation='relu', padding='same')(input_layer)
    x = MaxPooling2D((2, 2), padding='same')(x)

    x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D((2, 2), padding='same')(x)

    x = Conv2D(4, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D((2, 2), padding='same')(x)

    encoded = Dense(2304, activation='relu')(x)

    x = Conv2D(4, (3, 3), activation='relu', padding='same')(encoded)
    x = UpSampling2D((2, 2))(x)

    x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
    x = UpSampling2D((2, 2))(x)

    x = Conv2D(16, (3, 3), activation='relu', padding='same')(x)
    x = UpSampling2D((2, 2))(x)

    decoded = Conv2D(shape[len(shape)-1], (3, 3), activation='sigmoid',
    padding='same')(x)

    autoencoder = Model(input_layer, decoded)
    autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

    return autoencoder
```

Listing 5: Implementación del autoencoder en escala de grises. La especificación de la unidimensionalidad se hace en la *input shape*, al inicio.

```
def build_autoencoder(shape):
    input_layer = Input(shape=shape)
    x = Conv2D(16, (3, 3), activation='relu', padding='same')(input_layer)
    x = MaxPooling2D((2, 2), padding='same')(x)

    x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D((2, 2), padding='same')(x)

    x = Conv2D(4, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D((2, 2), padding='same')(x)

    encoded = Dense(2304, activation='relu')(x)

    x = Conv2DTranspose(4, (3, 3), activation='relu', padding='same')(encoded)
    x = UpSampling2D((2, 2))(x)

    x = Conv2DTranspose(8, (3, 3), activation='relu', padding='same')(x)
    x = UpSampling2D((2, 2))(x)

    x = Conv2DTranspose(16, (3, 3), activation='relu', padding='same')(x)
    x = UpSampling2D((2, 2))(x)

    decoded = Conv2DTranspose(shape[len(shape)-1], (3, 3), activation='sigmoid',
    padding='same')(x)

    autoencoder = Model(input_layer, decoded)
    autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

    return autoencoder
```

Listing 6: Implementación del autoencoder con el uso de la capa `Conv2DTranspose`.

```python
def build_autoencoder(shape):
    input_layer = Input(shape=shape)
    x = Conv2D(256, (3, 3), activation='relu', padding='same')(input_layer)
    x = MaxPooling2D((2, 2), padding='same')(x)

    x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D((2, 2), padding='same')(x)

    x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D((2, 2), padding='same')(x)

    x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D((2, 2), padding='same')(x)


    flat = Flatten()(x)
    encoded = Dense(4608, activation='relu')(flat)
    reshaped = Reshape((9, 16, 32))(encoded)


    x = Conv2DTranspose(32, (3, 3), activation='relu', padding='same')(reshaped)
    x = UpSampling2D((2, 2))(x)

    x = Conv2DTranspose(64, (3, 3), activation='relu', padding='same')(x)
    x = UpSampling2D((2, 2))(x)

    x = Conv2DTranspose(128, (3, 3), activation='relu', padding='same')(x)
    x = UpSampling2D((2, 2))(x)

    x = Conv2DTranspose(256, (3, 3), activation='relu', padding='same')(x)
    x = UpSampling2D((2, 2))(x)

    decoded = Conv2DTranspose(shape[len(shape)-1], (3, 3), activation='sigmoid',
    padding='same')(x)

    autoencoder = Model(input_layer, decoded)
    autoencoder.compile(optimizer='adam', loss='mse')
```

Listing 7: Implementación del autoencoder en su versión final, compilado con MSE.

Bibliografía

[1] Dan Morris and Siyu Yang. COCO Camera Traps format. Consultado el: 24 de febrero de 2020.

[2] Francesco Rovero, Mathias Tobler, and James Sanderson. Camera trapping for inventorying terrestrial vertebrates. *Manual on field recording techniques and protocols for All Taxa Biodiversity Inventories and Monitoring*, 01 2010.

[3] Nils J. Nilsson. Introduction to machine learning. An early draft of a proposed textbook, 1996.

[4] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[5] Artem Oppermann. https://www.deeplearning-academy.com/p/ai-wiki-machine-learning-vs-deep-learning.

[6] Haohan Wuhan and Bhiksha Raj. On the origin of deep learning. 2017.

[7] Alexander Bain. Mind and body. the theories of their relation. 1873.

[8] Donald Olding Hebb. The organization of behavior. *The american journal of psychology*, 63:633, 1950.

[9] Warren McCulloch and Walter Pitts. A logical calculus nervous activity. *The american journal of psychology*, 52:99–115, 1943.

[10] Sebastian Ruder. An overview of gradient descent optimization algorithms., 2016. cite arxiv:1609.04747Comment: Added derivations of AdaMax and Nadam.

[11] Yann Lecun. *A theoretical framework for back-propagation*. IEEE Computer Society Press, 1992.

[12] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning. *CoRR*, abs/1811.03378, 2018.

[13] Sepp Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. 04 1991.

[14] Lu Lu, Yeonjong Shin, Yanhui Su, and George Em Karniadakis. Dying relu and initialization: Theory and numerical examples. *ArXiv*, abs/1903.06733, 2019.

[15] Shaeke Salman and Xiuwen Liu. Overfitting mechanism and avoidance in deep neural networks. *CoRR*, abs/1901.06566, 2019.

[16] Linda Shapiro and George Stockman. *Computer Vision*. 03 2000.

[17] N. Otsu. A threshold selection method from gray-level histograms. *IEEE Transactions on Systems, Man, and Cybernetics*, 9(1):62–66, 1979.

[18] António Anjos and Hamid Shahbazkia. Bi-level image thresholding - a fast method. volume 2, pages 70–76, 01 2008.

[19] Mehmet Sezgin and Bülent Sankur. Survey over image thresholding techniques and quantitative performance evaluation. *Journal of Electronic Imaging*, 13(1):146 – 165, 2004.

[20] Stephen M. Pizer, E. Philip Amburn, John D. Austin, Robert Cromartie, Ari Geselowitz, Trey Greer, Bart Ter Haar Romeny, and John B. Zimmerman. Adaptive histogram equalization and its variations. *Comput. Vision Graph. Image Process.*, 39(3):355–368, September 1987.

[21] Karel Zuiderveld. *Contrast Limited Adaptive Histogram Equalization*, page 474–485. Academic Press Professional, Inc., USA, 1994.

[22] 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, nov 2018.

[23] Hervé Abdi and Lynne J. Williams. Principal component analysis. *WIREs Computational Statistics*, 2(4):433–459, 2010.

[24] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[25] Jakub Kvita. Visualizations of RNN units, 2016. Consultado el 1 de abril de 2020.

[26] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets, 2014.

[27] Github de StyleGAN.

[28] Github de deepfake.

[29] Kevin Schawinski, Ce Zhang, Hantian Zhang, Lucas Fowler, and Gokula Krishnan Santhanam. Generative adversarial networks recover features in astrophysical images of galaxies beyond the deconvolution limit. *Monthly Notices of the Royal Astronomical Society: Letters*, 467(1):L110–L114, 01 2017.

[30] Xintao Wang, Ke Yu, Shixiang Wu, Jinjin Gu, Yihao Liu, Chao Dong, Chen Change Loy, Yu Qiao, and Xiaoou Tang. ESRGAN: Enhanced Super-Resolution Generative Adversarial Networks. *arXiv e-prints*, page arXiv:1809.00219, September 2018.

[31] OpenCV. How to Use Background Subtraction Methods.

[32] Soharab Hossain Shaikh, Khalid Saeed, and Nabendu Chaki. *Moving Object Detection Using Background Subtraction*, pages 15–23. Springer International Publishing, Cham, 2014.

[33] Yalın BAS¸TANLAR Ula¸s TEKEL˙I. Elimination of useless images from raw camera-trap data. *Turk J Elec Eng & Comp Sci*, pages 11–13, 2018.

[34] Lauren Barghout and Jacob Sheynin. Real-world scene perception and perceptual organization: Lessons from Computer Vision. *Journal of Vision*, 13(9):709, jul 2013.

[35] Richard Nock and Frank Nielsen. Statistical region merging. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(11):1452–1458, 2004.

[36] Ron Ohlander, Keith Price, and D. Raj Reddy. Picture segmentation using a recursive region splitting method. *Computer Graphics and Image Processing*, 8(3):313 – 333, 1978.

[37] Alex Clark andFredrik Lundh. pillow.readthedocs.io. Consultado el: 13 de marzo de 2020.

[38] Adrian Rosebrock. github.com/jrosebr1/imutils. Consultado el: 13 de marzo de 2020.

[39] Intel. opencv.org. Consultado el: 13 de marzo de 2020.

[40] Theanos and Google. tensorflow.org. Consultado el: 15 de marzo de 2020.

[41] Facebook. pytorch.org. Consultado el: 15 de marzo de 2020.

[42] David Cournapeau. scikit-learn.org. Consultado el: 15 de marzo de 2020.

[43] Sara Beery, Grant Van Horn, and Pietro Perona. Recognition in terra incognita. In *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part XVI*, pages 472–489, 2018.

[44] Norm Estabrook, Mauricio de los Santos, and Gary Moore. Azure Copy, oct 2019. Consultado el: 24 de febrero de 2020.

[45] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

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

[47] Francois Chollet. Building autoencoders in keras.

[48] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[49] Zhou Wang, Alan Conrad Bovik, Hamid Rahim Sheikh, and Eero P. Simoncelli. Image quality assessment: From error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

[50] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA, 1984.

[51] R Aruna Devi and K Nirmala. Construction of Decision Tree : Attribute Selection Measures. *International Journal of Advancements in Research & Technology*, 2(4):343–347, 2013.