UNIVERSITAT POLITÈCNICA DE VALÈNCIA

_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería de Telecomunicación

Detección precoz de sarna en rebecos mediante redes neuronales convolucionales

Trabajo Fin de Máster

Máster Universitario en Ingeniería de Telecomunicación

AUTOR/A: Martínez Ordás, María

Tutor/a: Guerola Navarro, Vicente

CURSO ACADÉMICO: 2022/2023

Resumen

La sarna sarcóptica, causada por el ácaro Sarcoptes scabiei, es actualmente la enfermedad más grave que afecta al rebeco. Este parásito provoca inflamación, prurito intenso, alopecia, lesiones cutáneas y delgadez extrema en el rebeco afectado, provocando en muchas ocasiones la muerte del animal. Esta enfermedad es altamente contagiosa por lo que, si no se detecta a tiempo, puede propagarse rápidamente no solo entre rebecos sino también a otras especies, incluida el ser humano. Considerando la propia naturaleza esquiva de estos animales y, por lo tanto, la dificultad de seguimiento de su salud, surge la idea de crear una herramienta automatizada que nos permita detectar sarna y, de esta manera, poder parar su propagación a tiempo. En este trabajo se desarrollará un método para la detección automática de sarna en rebecos mediante técnicas basadas en inteligencia artificial. Para ello se creará y entrenará una red neuronal convolucional (CNN) que sea capaz de percibir las diferencias en la piel y pelo de los rebecos enfermos con un alto nivel de precisión.

Resum

La sarna sarcóptica, causada per l'àcar Sarcoptes scabiei, és actualment la malaltia més greu que afecta l'isard. Aquest paràsit provoca inflamació, pruïja intensa, alopècia, lesions cutànies i primesa extrema en l'isard afectat, provocant en moltes ocasions la mort de l'animal. Aquesta malaltia és altament contagiosa pel que, si no es detecta a temps, pot propagar-se ràpidament no sols entre isards sinó també a altres espècies, inclosa l'ésser humà. Considerant la pròpia naturalesa esquiva d'aquests animals i, per tant, la dificultat de seguiment de la seua salut, sorgeix la idea de crear una eina automatitzada que ens permeta detectar sarna i, d'aquesta manera, poder parar la seua propagació a temps. En aquest treball es desenvoluparà un mètode per a la detecció automàtica de sarna en isards mitjançant tècniques basades en intel·ligència artificial. Per a això es crearà i entrenarà una xarxa neuronal convolucional (CNN) que siga capaç de percebre les diferències en la pell i pèl dels isards malalts amb un alt nivell de precisió.

Abstract

Sarcoptic mange, caused by the Sarcoptes scabiei mite, is currently the most serious disease affecting chamois. This parasite causes inflammation, intense pruritus, alopecia, skin lesions and extreme thinness in the affected chamois, often leading to the animal's death. This disease is highly contagious and, if not detected in time, can spread rapidly not only among chamois but also to other species, including humans. Considering the elusive nature of these animals and, therefore, the difficulty of monitoring their health, the idea arises to create an automated tool that allows us to detect scabies and, thus, stop its spread in time. In this project, a method for the automatic detection of mange in chamois will be developed using techniques based on artificial intelligence. To do this, a convolutional neural network (CNN) will be created and trained to be able to perceive the differences in the skin and hair of sick chamois with a high level of accuracy.

A mi madre y mi hermana, por animarme y apoyarme incondicionalmente todos estos años.

A mi padre por creer en mí y aportarme sus conocimientos en el campo de la veterinaria, sin él este trabajo no hubiera sido posible.

A Juan Carlos Peral Sánchez, Director Técnico de Reservas de Caza de León y a Francisco Rojo Vázquez, Catedrático de Parasitología de la Facultad de Veterinaria de León, quiénes me proporcionaron las imágenes y la información necesaria para el desarrollo de este proyecto.

A Vicente Guerola Navarro, mi tutor, por su motivación y por estar siempre disponible para responder a mis preguntas.

Índice general

Índice de figuras

Índice de tablas

Listado de siglas empleadas

CNN Convolutional Neural Networks o Redes Neuronales Convolucionales.

CNTK Cognitive Toolkit.

FN False Negative.

FP False Positive.

MSE Mean Squared Error o Error Cuadrático Medio.

ODS Objetivos de Desarrollo Sostenible.

ReLU Rectified Linear Unit.

SGD Stochastic Gradient Descent o Descenso de Gradiente Estocástico.

TN True Negative.

TP True Positive.

URL Uniform Resource Locator.

VGG Visual Geometry Group.

Parte I

Memoria

Capítulo 1

Introducción

1.1. Motivación

La sarna es una enfermedad devastadora que afecta a gran cantidad de animales, incluido el ser humano. Esta enfermedad, causada por ácaros, provoca lesiones cutáneas, irritación, pérdida de pelo y deterioro de la condición física del animal, entre otros síntomas. Se trata de una enfermedad altamente contagiosa, por lo que su detección temprana es crucial.

Dentro de los animales afectados, los rebecos son especialmente vulnerables a este ácaro. Cada vez que surge una nueva infección estos animales sufren, se debilitan y sus poblaciones disminuyen rápidamente. Como se trata de animales silvestres y esquivos, la detección de la enfermedad se complica, lo que conlleva graves consecuencias.

Por esta razón, surge la idea de crear una red neuronal convolucional que nos permita detectar los patrones típicos del pelo y la piel de los rebecos enfermos. Esta tecnología permitiría iniciar rápidamente el tratamiento adecuado para contrarrestar los efectos de la enfermedad.

El campo de la inteligencia artificial es sumamente interesante y está experimentando grandes avances. Su aplicación en el contexto de la salud animal, no solo supone una contribución a la protección de los rebecos, sino también un apoyo al medio ambiente. Además, este proyecto tiene el potencial de impulsar el avance y la aplicación de técnicas de inteligencia artificial en el ámbito de la salud animal, abriendo nuevas oportunidades para la investigación y el desarrollo de soluciones innovadoras.

1.2. Objetivos

Los objetivos principales de este Trabajo de Fin de Máster son:

- **Investigación en el campo de la Inteligencia Artificial:** La realización de este trabajo ha implicado una meticulosa investigación en el campo de la inteligencia artificial, centrándonos especialmente en el estudio de las redes neuronales artificiales. Para el desarrollo de un proyecto de esta magnitud es necesario un conocimiento profundo del ámbito al que pertenece. Además, este trabajo nos ha ofrecido la oportunidad de aplicar y ampliar los conocimientos adquiridos durante nuestra formación académica y el máster. Asimismo, este proyecto

nos ha permitido ampliar nuestras habilidades en Python, una herramienta fundamental en el contexto de las redes neuronales artificiales.

- **Desarrollo de una red neuronal para la detección de sarna en rebecos:** El objetivo principal de este Trabajo de Fin de Máster ha sido la implementación de una red neuronal que nos permita la detección temprana de sarna en estos animales. Como se ha comentado anteriormente, los rebecos son gravemente afectados por la enfermedad y se necesita de una herramienta que ayude a detener su propagación. Debido a las circunstancias que afectan a estos animales, la creación de una red neuronal que detecte los patrones de la piel y el pelaje en los especímenes enfermos es la mejor solución. Por lo tanto, la implementación de una red neuronal que cumpla con este cometido es el objetivo más importante de este proyecto.
- **Contribución a la detección de enfermedades en animales y conservación de la fauna:** Mediante el desarrollo de esta red neuronal buscamos realizar una contribución en el ámbito de la salud animal y la conservación de la fauna. La detección precoz de enfermedades como la sarna es crucial a la hora de realizar el tratamiento y prevenir la propagación de la enfermedad. Esta aportación no solo tendrá efectos positivos en la población de los rebecos sino que también beneficiará a otros animales, incluido el ser humano, ayudando de esta manera a mantener el equilibrio ecológico de su hábitat natural. Además, la creación de esta red neuronal puede abrir nuevas oportunidades para aplicar técnicas de inteligencia artificial en la detección de enfermedades en animales. La inteligencia artificial ya está siendo ampliamente utilizada en el ámbito de la salud humana, pero en el contexto animal todavía es limitada. Es necesario expandir su uso para conseguir garantizar el bienestar y cuidado de todas las especies de nuestro planeta.

1.3. Contribución a los Objetivos de Desarrollo Sostenible (ODS)

Los **Objetivos de Desarrollo Sostenible (ODS)**, también llamados Objetivos Globales, fueron aprobados por unanimidad por la Asamblea General de las Naciones Unidas el 25 de septiembre de 2015. Estos objetivos forman parte de la Agenda 2030 para el Desarrollo Sostenible, un plan de acción que tiene como objetivo abordar los desafíos mundiales y promover el bienestar de las personas, el planeta, la prosperidad y la paz universal [1].

Los objetivos de este trabajo se alinean con tres ODS:

- **Objetivo 3: Salud y Bienestar.** El objetivo principal de este proyecto es la detección y prevención de la propagación de la sarna en los rebecos, para evitar su sufrimiento y garantizar su bienestar y salud. Además, al prevenir y tratar la enfermedad, se reduce la probabilidad de que la sarna se transmita a otros animales, tanto silvestres como domésticos y, por ende, a los humanos. Por lo tanto, ayuda a mantener la salud y bienestar tanto de los animales como de los seres humanos.
- **Objetivo 15: Vida de Ecosistemas Terrestres.** Los efectos de la sarna pueden ser catastróficos en las poblaciones de rebecos y otros animales. Al tratar de evitar la propagación de esta enfermedad se vela por la conservación de la fauna y se contribuye al mantenimiento del equilibro de los ecosistemas terrestres.

- **Objetivo 9: Industria, Innovación e Infraestructuras.** La aplicación de tecnologías de inteligencia artificial, como las redes neuronales artificiales, en el campo de la salud animal representa un importante avance tecnológico y científico en este ámbito. Esta perspectiva novedosa proporciona una técnica innovadora para la detección de enfermedades en animales.

Capítulo 2

Marco Teórico

2.1. Sarna en Rebecos

2.1.1. Introducción

Desde tiempos inmemoriales se conocía la sarna tanto en animales como en el hombre. Ya en el libro del Levítico del Antiguo Testamento se prohíbe el uso de animales sarnosos en las ofrendas divinas.

Pero, a pesar de que la enfermedad era conocida desde la antigüedad, no fue hasta 1687 que se identificó el ácaro Sarcoptes scabiei como agente causal de esta enfermedad. Hasta entonces se creía que las enfermedades eran causadas por la alteración de los "humores" (Teoría de Galeno).

La acción del ácaro sobre la epidermis del animal afectado causa graves trastornos, comenzando por prurito intenso, malestar, aspecto muy deteriorado, caquexia y provoca la muerte de hasta el 90 % de los miembros del rebaño.

2.1.2. Etiología

El ácaro **Sarcoptes scabiei** es un parásito microscópico, que hace surcos en la piel de la persona o animal afectado.

El ciclo biológico de estos ácaros es directo (se transmite de uno a otro por contacto directo), y se completa en menos de un mes. Todas las fases de su ciclo se realizan sobre la piel del hospedador afectado.

La hembra fecundada excava galerías subepidérmicas en las que deposita los huevos, 2-6 diarios hasta poner un máximo de 25-30. Cada hembra excava una única galería tortuosa que aumenta 2-3 mm al día, hasta un máximo de 35-50 mm, alimentándose de la propia piel. Los huevos se incuban 7 días y nacen las larvas, las cuales pueden salir al exterior o hacer galerías perpendiculares a la de su madre. A los 9 días mudan a ninfas y en los próximos 3-6 días hacen dos mudas y se convierten en adultos. Los machos son menos numerosos y nómadas, deambulando por la superficie de la piel en busca de hembras que fecundar. El ciclo completo finaliza en 8-14 días [2].

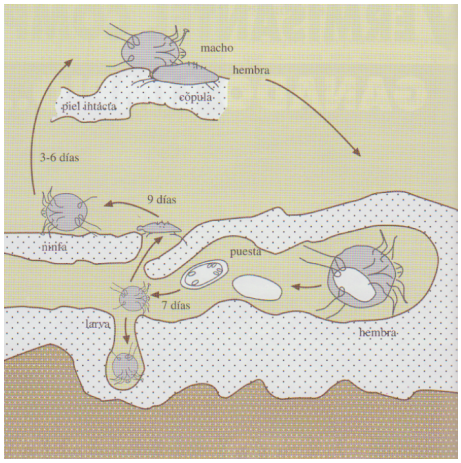


Figura 2.1: Ciclo biológico de Sarcoptes scabiei.

En seis meses una pareja de S. scabiei podrían tener seis generaciones, llegando a nacer un millón de hembras y medio millón de machos.

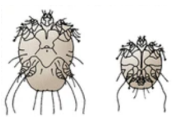
	Demodex	*Chorioptes*	*Sarcoptes*	*Psoroptes*
Forma	Alargado, con estrías transversales	Ovalado	Globoso, cutícula estriada con cerdas, espinas y escamas triangulares	Ovalado con estriaciones muy finas y espinas dorsales
Extremo anterior	Muy corto y unido al tórax	Largos (menos que *Psoroptes*) y redondeado	Corto y cuadrado con dos cerdas verticales	Largo y cónico sin cerdas verticales
Patas	Muy cortas, como muñones	Largas, todos los pares de patas sobresalen del cuerpo	Cortas, solo sobresalen del cuerpo el 1º y 2º par	Largas, todos los pares de patas sobresalen del cuerpo
Terminación en patas en uñas o ventosas	Uñas	Ventosas con pedúnculo corto no articulado en los pares de patas	Ventosas con pedúnculo largo no articulado en los pares de patas	Ventosas con pedúnculo largo articulado en los pares de patas
♀ & ♂				

Tabla 2.1: Características de los distintos géneros de agentes productores de sarnas en pequeños rumiantes.

2.1.3. Epidemiología

Los ácaros de la sarna son bastante específicos, siendo el Sarcoptes scabiei el que principalmente afecta al rebeco [3] [4].

La transmisión más común es a través de contacto directo entre las pieles del animal afectado y el sano. A medida que aumenta la población en un rebaño de rebecos la transmisión es mucho más rápida, lo que significa que es más necesaria la prevención. También es posible la transmisión a través de materiales cercanos, como puede ser camas, rocas o ramas, puesto que estos ácaros pueden vivir varios días fuera del hospedador, pero este tipo de transmisión es menos frecuente [2].

La alimentación inadecuada, la suciedad, el estrés, los esfuerzos excesivos, el frío y la lluvia aumentan la predisposición a padecer la enfermedad, lo que significa que aumenta la probabilidad de infección en los meses de otoño e invierno.

2.1.4. Patología

Los ácaros actúan irritando la piel al perforar la epidermis, incluidas las pezuñas. A esto se une la reacción del hospedador al recibir la perforación y la respuesta a los restos metabólicos que produce el parásito: empieza con nódulos, vesículas, pústulas, costras, descamación, hiperqueratosis, paraqueratosis, alopecias y adelgazamiento progresivo que puede desembocar en caquexia y terminando en muchos casos en muertes [2] [5] [6]. En la figura 2.2 se muestra la progresión de la sarna en los rebecos afectados [7].

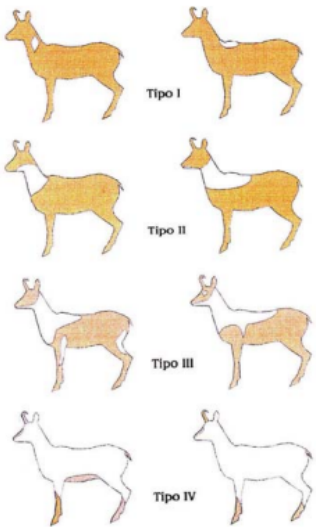


Figura 2.2: Categorías de rebecos con sarna según el porcentaje de superficie corporal con lesiones cutáneas.

En sucesivas reinfestaciones se suele desarrollar cierta inmunidad protectora y reduce la afección.

2.1.5. Diagnóstico

En rumiantes domésticos el método de diagnóstico escogido es el microscópico después del raspado cutáneo. Este método permite identificar el ácaro a través del microscopio fácilmente [2].

Además, si previamente se ha realizado la observación de lesiones y los datos epidemiológicos, como la introducción de nuevos animales o casos anteriores de sarna, resulta de gran ayuda para el diagnóstico.

En rebecos, dada su dificultad para atraparlos y lo nerviosos que son (lo cual produce bajas por el estrés de estar encerrados), la observación de lesiones debería ser el método de elección. Si esa observación fuera precoz comenzaríamos antes el tratamiento y, de esta manera, los animales sufrirían menos y se evitaría que la infección se propagase.

2.1.6. Tratamiento

El éxito de un tratamiento antiácaros está condicionado, principalmente, a que sea un tratamiento en las primeras fases de infestación. De ello depende que no existan bajas y de que el sufrimiento de los animales sea menor.

El producto utilizado en rebecos será por vía oral, a través de pienso medicado. Hay que tener en cuenta que estos productos no son ovicidas (no tiene la capacidad de eliminar los huevos de los ácaros), por lo que el tratamiento no puede ser muy corto. Se suelen utilizar avermectinas, por ser un tratamiento oral [2] [8].

2.1.7. Prevención

Hasta hace pocos años, el interés general y científico por la sanidad de los animales silvestres era mínimo, y solo ha aumentado cuando se ha considerado que la salud de estos animales es importante para la salud de los animales domésticos y para la salud humana: One Health [9] [10].

Los animales silvestres no pueden aislarse, viven sin fronteras y en lugares abiertos. Es por esto que muchas enfermedades importantes que afectan a los animales domésticos y a las personas permanecen activas, ya que sobreviven en estos animales y, al entrar en contacto con animales domésticos o con el hombre, hacen aparecer de nuevo la enfermedad.

En España ha habido muchos brotes importantes de sarna en animales salvajes de nuestros parques naturales, sobre todo en rebeco y en cabra montesa, lo que ha provocado gran mortalidad en sus rebaños [6].

Se considera que la sarna en rebeco suele estar de forma enzoótica (es una enfermedad endémica en estos animales y se presenta de manera recurrente), pero con graves brotes epizoóticos que provocan una mortalidad de hasta el 90 % del rebaño afectado. Esto significa que en momentos de aumento de la población, frío, humedad, aumento de la suciedad y peor alimentación hay que tener una prevención especial, comenzando el tratamiento de forma inmediata ante los primeros síntomas. Las alopecias y el mal aspecto general deberían ser los primeros síntomas que podrían indicarnos un comienzo de la enfermedad.

En este trabajo, utilizando técnicas de inteligencia artificial, conseguiremos detectar esas alopecias y estados corporales deficientes desde los primeros estadios, para así comenzar un tratamiento

temprano y adecuado que consiga mantener el rebaño de rebecos en buen estado y disminuir el sufrimiento, enfermedad y muertes que causan esta grave enfermedad.

(a) **(b)**

Figura 2.3: Rebecos con Sarna.

2.2. Redes neuronales artificiales

2.2.1. Inteligencia Artificial, Machine Learning y Deep Learning

En la actualidad, la **Inteligencia Artificial (IA)** se ha convertido en un campo de estudio y aplicación de gran relevancia. Su objetivo principal es desarrollar sistemas capaces de llevar a cabo tareas que requieren de inteligencia humana, como el reconocimiento de patrones, la toma de decisiones y el aprendizaje [11] [12].

Dentro de la IA, el **Machine Learning** o Aprendizaje Automático ha ganado especial atención. Este enfoque permite a las máquinas aprender de manera autónoma a partir de los datos, sin requerir una programación explícita. El Machine Learning ha demostrado ser una herramienta poderosa para resolver problemas complejos en diversas áreas [11] [12].

Uno de los subcampos más destacables del Aprendizaje Automático es el **Deep Learning** o Aprendizaje Profundo. El Deep Learning se basa en el desarrollo de **redes neuronales artificiales profundas**, las cuales están inspiradas en la estructura y el funcionamiento del cerebro humano. Estas redes están compuestas por múltiples capas de neuronas interconectadas y, mediante el Aprendizaje Profundo, tienen la capacidad de aprender de manera jerárquica, capturando características y representaciones complejas a partir de los datos [11] [12].

A diferencia de las redes neuronales tradicionales, que suelen tener una o dos capas ocultas, el Deep Learning emplea redes neuronales más profundas con un mayor número de capas ocultas. Esta estructura aumenta la capacidad del modelo para aprender representaciones de características más abstractas y complejas, lo cual las habilita para resolver problemas altamente complejos de manera más efectiva.

A continuación, nos centraremos en analizar en detalle las bases de las redes neuronales artificiales

y los conceptos esenciales para entender su funcionamiento en el contexto del Aprendizaje Profundo. Se explicará cómo las redes neuronales tienen la capacidad de aprender de forma autónoma y llevar a cabo tareas complejas como el reconocimiento de patrones, la clasificación de datos y la generación de contenido. Al comprender estos principios, obtendremos una perspectiva más clara sobre el potencial y la aplicabilidad de las redes neuronales en el campo de la Inteligencia Artificial.

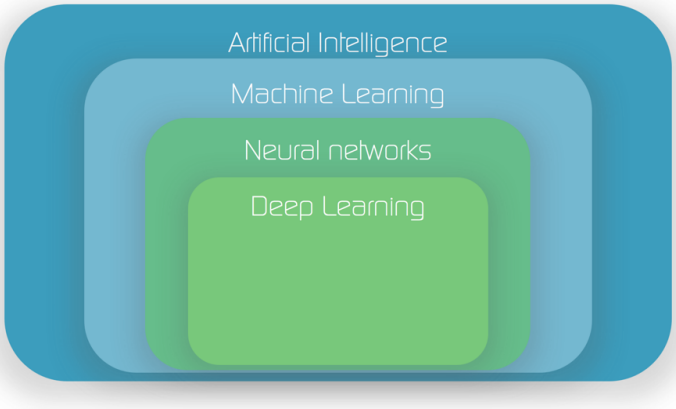


Figura 2.4: Relación entre IA, Machine Learning, Redes Neuronales y Deep Learning.

2.2.2. Fundamentos de las redes neuronales

Una red neuronal artificial es un sistema informático compuesto por un conjunto de unidades conectadas llamadas **neuronas** (o nodos) que se organizan en lo que llamamos **capas**. Este modelo está basado en la organización y funcionamiento del cerebro humano, donde las neuronas biológicas se comunican a través de sinapsis [13].

Los datos se propagan a través de la red, empezando por la capa de entrada y pasando por las capas ocultas hasta llegar a la capa de salida. A medida que los datos se propagan, las neuronas se activan si la entrada que reciben supera un cierto umbral.

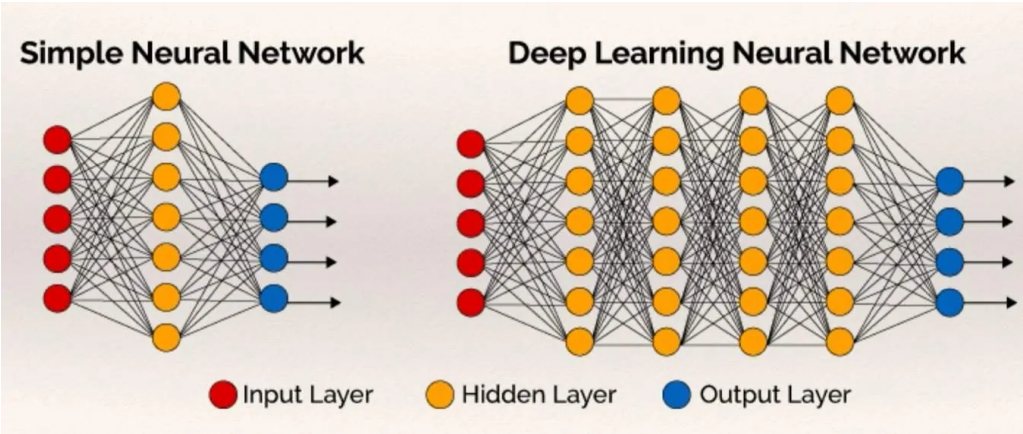


Figura 2.5: Arquitectura básica de red neuronal artificial.

Cada conexión entre dos neuronas tiene un **peso** asociado que representa la fuerza de la cone-

xión entre los dos nodos. Las neuronas (excepto las de la capa de entrada) calcularán una suma ponderada de cada una de las conexiones entrantes.

$$w_0x_0 + w_1x_1 + ... + w_Nx_M \tag{2.1}$$

Esta suma se pasará a través de una función de activación, que realiza algún tipo de transformación en la suma dada. El resultado de la función se convierte en la entrada para los nodos de la siguiente capa. Este proceso continúa hasta llegar a la capa de salida [14] .

$$y = \phi(w_0x_0 + w_1x_1 + ... + w_Nx_M) \tag{2.2}$$

El número de neuronas de la capa de salida depende del número de posibles clases de salida o predicciones que tengamos.

2.2.2.1. Función de activación

En una red neuronal artificial, una **función de activación** define como será la salida de un nodo dada una entrada o un conjunto de entradas. Si nos interesase transmitir la información sin modificaciones utilizaríamos la función de identidad pero, en general, la mayoría de las funciones de activación no son lineales, lo que permite que nuestras redes neuronales sean capaces de resolver problemas cada vez más complejos.

Las funciones de activación más usadas son [14] [15]:

- **Función Escalón:** propaga un 0 si el valor de x es negativo, o un 1 si es positivo. No hay casos intermedios y sirve para clasificar de forma muy estricta.

$$f(x) = \begin{cases} 0 & \text{si } x < 0 \\ 1 & \text{si } x \geq 0 \end{cases} \tag{2.3}$$

- **Función Rectificadora (ReLU):** transforma los valores introducidos anulando los valores negativos y dejando los positivos tal y como entran. Se trata de la más utilizada en la actualidad debido a su simplicidad y su eficiencia.

$$f(x) = max(0, x) = \begin{cases} 0 & \text{si } x < 0 \\ x & \text{si } x \geq 0 \end{cases} \tag{2.4}$$

- **Función Sigmoide:** transforma los valores introducidos a una escala (0,1), donde los valores altos tienden de manera asintótica a 1 y los valores muy bajos tienden de manera asintótica a 0. Suele emplearse en modelos de clasificación binaria.

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

- **Función Tangente Hiperbólica:** transforma los valores introducidos a una escala (-1,1), donde los valores altos tienden de manera asintótica a 1 y los valores muy bajos tienden de manera asintótica a -1.

$$f(x) = \frac{2}{1 + e^-2x} - 1 \tag{2.6}$$

- **Función SoftMax:** transforma las salidas a una representación en forma de probabilidades, de tal manera que el sumatorio de todas las probabilidades de las salidas de 1. Se suele utilizar en las última capas en modelos de clasificación multiclase y permite que la salida de la red neuronal pueda interpretarse como una distribución de probabilidad sobre las diferentes clases de salida.

$$f(x) = \frac{e_j^Z}{\sum_{k=1}^{K} e^{Z_k}} \tag{2.7}$$

2.2.3. Entrenamiento de modelos de redes neuronales

El proceso de entrenamiento es una de las etapas más importantes en la creación de modelos de aprendizaje automático, en la cual el modelo aprende a partir de los datos de entrenamiento que se le proporcionan. Este proceso implica ajustar los pesos del modelo según una función de pérdida, iterando a través de los datos de entrenamiento. Con esto se consigue que el modelo aprenda a hacer predicciones precisas en datos nuevos.

Para que este proceso sea efectivo se necesita contar con un **conjunto de datos de entrenamiento** de alta calidad y representativo del problema a resolver, que cubra todas las posibles situaciones que el modelo podría enfrentar en la vida real.

Además, se tienen que seleccionar cuidadosamente los **hiperparámetros** del modelo, como la tasa de aprendizaje, el tamaño del lote y el número de épocas. Estas variables son parámetros que no se aprenden durante el proceso de entrenamiento, sino que se establecen antes de comenzar el entrenamiento y se ajustan para mejorar el rendimiento del modelo, por lo tanto, deben seleccionarse minuciosamente [16].

Por último, es importante seleccionar un buen **algoritmo de optimización**, puesto que puede marcar una gran diferencia en la eficacia y eficiencia del proceso de entrenamiento del modelo.

2.2.3.1. Función de pérdida

Durante el entrenamiento, los pesos de la red neuronal se ajustan para **minimizar una función de pérdida** que mide la discrepancia entre la salida predicha y la salida verdadera. La elección de la función de pérdida es importante y puede variar según el problema. Una función de pérdida común es el error cuadrático medio (MSE), que mide la diferencia al cuadrado entre la predicción de salida proporcionada y la etiqueta o valor real [13].

2.2.3.2. Backpropagation

Backpropagation es un algoritmo utilizado en el entrenamiento de redes neuronales artificiales. Su objetivo es ajustar los pesos y los sesgos de una red neuronal para minimizar la función de pérdida asociada con el problema que se está tratando de resolver [17].

El proceso de backpropagation comienza con una fase de propagación hacia delante donde los datos de entrada se pasan a través de la red neuronal y se generan las predicciones. Al finalizar, se calcula la función de pérdida como se comenta en el apartado anterior.

Una vez realizados los cálculos comienza la fase de retropropagación donde se calculan las derivadas parciales (gradiente) de la función de pérdida con respecto a cada uno de los pesos y sesgos de la red neuronal utilizando la regla de la cadena, la cual permite determinar cómo los cambios en una función compuesta se propagan a través de las funciones internas y externas. Las derivadas parciales se utilizan para ajustar los pesos y sesgos de la red neuronal utilizando un algoritmo de optimización. Dos de los algoritmos más utilizados son los siguientes [18]:

- **SGD (Descenso de Gradiente Estocástico):** es un método de optimización que trata de minimizar la función de pérdida. Durante cada iteración, se calcula el gradiente de un lote de imágenes de entrenamiento y se multiplica por la tasa de aprendizaje, ajustándose de esta manera los pesos de cara a la siguiente iteración. Este proceso de actualización de los pesos consigue minimizar la función de pérdida global al repetirse en múltiples lotes de entrenamiento.
- **Adam:** es otro método de optimización que combina los conceptos del algoritmo de descenso de gradiente estocástico (SGD) y del método de momento adaptativo (AdaGrad). Al igual que el método anterior, el algoritmo Adam ajusta los pesos del modelo a partir del gradiente de la función de pérdida pero también mantiene una estimación del momento del gradiente. Esta diferencia permite acelerar la convergencia en espacios de parámetros con curvas de pérdida más complejas. Esto quiere decir que las tasas de aprendizaje se ajustarán de manera automática para cada peso individual en función de las estimaciones del primer y segundo momento del gradiente. Por esta razón, Adam converge más rápidamente hacia el mínimo de la función de pérdida que SGD.

La retropropagación es un proceso iterativo y se repite muchas veces hasta conseguir ajustar gradualmente los pesos de la red.

2.2.3.3. Overfitting y Underfitting

El **overfitting** (o sobreajuste) es un fenómeno que se produce cuando el modelo de aprendizaje automático proporciona predicciones precisas para los datos de entrenamiento, pero no para datos nuevos. Esto se debe a que el modelo ha memorizado las características específicas del conjunto de entrenamiento, pero no ha aprendido a generalizar.

Durante el entrenamiento, se puede comprobar si el modelo está sobreajustándose, fijándonos en métricas como la precisión y la pérdida en los conjuntos de entrenamiento y validación. Si las métricas del modelo son buenas con el conjunto de entrenamiento pero no con el conjunto de validación quiere decir que el modelo realmente no clasifica bien y se está produciendo overfitting.

El modelo ha aprendido muy bien las características del conjunto de entrenamiento, pero si se le muestran datos que se desvían ligeramente de los datos utilizados durante el entrenamiento, es incapaz de generalizar y predecir con precisión el resultado.

Las técnicas más destacadas que nos permiten reducir el overfitting son [13] [19] [20]:

- **Agregar más datos al conjunto de entrenamiento:** de esta manera nuestros datos de entrenamiento serán más diversos y el modelo tendrá menos probabilidades de sobreajustarse.

- **Data Augmentation:** se trata de una técnica que implica la generación de nuevas instancias de datos a partir de los datos existentes mediante la aplicación de transformaciones y manipulaciones. Estas transformaciones pueden incluir rotaciones, zoom, desplazamientos y cambio de color, entre otros, que alteran la apariencia de los datos originales. El objetivo del data augmentation es aumentar la cantidad y la variedad de datos de entrenamiento disponibles para un modelo de aprendizaje automático, lo que puede mejorar su capacidad para generalizar y hacer predicciones precisas sobre datos nuevos y desconocidos.
- **Reducir la complejidad del modelo:** eliminando algunas capas del modelo o reduciendo el número de neuronas de las capas.
- **Dropout:** añadir dropout a un modelo significa que este ignorará aleatoriamente un subconjunto de neuronas de una capa determinada durante el entrenamiento, es decir, descartará los nodos de la capa. Esto evitará que los nodos descartados participen en la predicción de los datos. De esta manera, se consigue que el modelo aprenda a generalizar mejor datos que no ha visto anteriormente.
- **Regularización:** es una técnica que introduce una penalización en la función de pérdida durante el entrenamiento para controlar la complejidad del modelo. La regularización más común es la regularización L1 y L2, que penaliza los valores altos de los pesos del modelo. Esto ayuda a evitar que los pesos se vuelvan demasiado grandes y dominen la función de pérdida, lo que puede llevar al sobreajuste.

Por otro lado, si el modelo no se ajusta lo suficiente tenemos **underfitting**. El underfitting se presenta cuando la precisión de entrenamiento del modelo es baja y/o las pérdidas son altas. En este caso, el modelo no es capaz de clasificar los datos de entrenamiento adecuadamente, lo que sugiere que no podrá hacer predicciones precisas en datos no vistos previamente. Para conseguir reducir el underfitting se realizarán los cambios opuestos a los del overfitting.

2.2.4. Redes neuronales convolucionales (CNN)

Las **CNNs (Convolutional Neural Networks o Redes Neuronales Convolucionales)** son un modelo de aprendizaje profundo que se utiliza para procesar datos con un patrón reticular, como las imágenes y los vídeos.

Está inspirado en la organización de la corteza visual animal y está diseñado para aprender de forma automática y adaptativa jerarquías espaciales de características, desde patrones de bajo a alto nivel. Una CNN se compone normalmente de tres tipos de capas: convolución, agrupación y capas totalmente conectadas. Las capas de convolución y agrupación extraen las características, mientras que la capa totalmente conectada transforma las características extraídas en resultados finales.

Las CNNs se utilizan para diversas tareas como clasificación de imágenes, detección de objetos, segmentación semántica, reconocimiento facial y generación de imágenes. Esto las convierte en una herramienta de gran importancia en el campo del aprendizaje automático y la inteligencia artificial. Además, se aplican en una amplia variedad de ámbitos y aplicaciones, como medicina, reconocimiento de voz, robótica, industria automotriz y análisis de vídeo, entre otras. Estas redes destacan puesto que han demostrado un rendimiento excepcional al capturar características espaciales y una alta capacidad de adaptabilidad [21].

En el contexto de este trabajo, las CNNs se emplearán para la detección de sarna utilizando imágenes como referencia. Su capacidad para extraer características visuales relevantes de las imágenes permitirá identificar patrones asociados a la presencia de sarna y facilitará el proceso de diagnóstico.

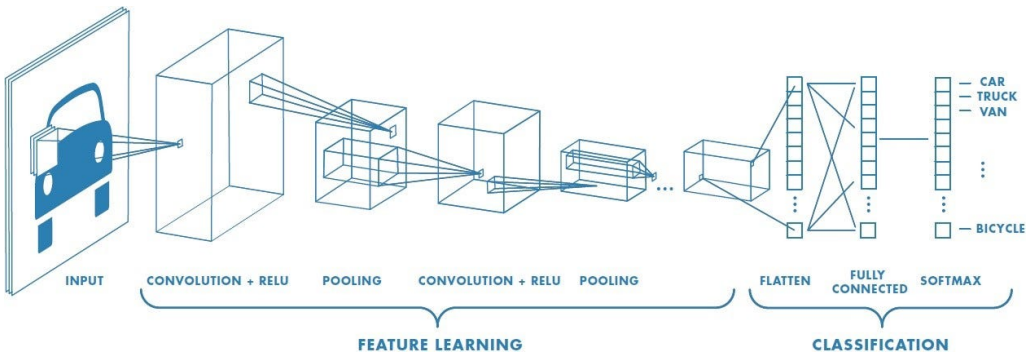


Figura 2.6: Arquitectura de una CNN.

2.2.4.1. Capa de convolución

La **capa de convolución** es un componente fundamental de la arquitectura de las CNNs. Realiza la extracción de características mediante una combinación de operaciones lineales y no lineales, incluyendo la operación de convolución y la función de activación.

Estas capas aplican filtros o kernels a la imagen de entrada. El filtro se va desplazando por la imagen, realizando una operación de convolución en cada posición. El resultado final de esta operación es el mapa de características, mapa de activación o característica convolucionada. Cada filtro extrae una característica específica, como bordes, formas o texturas, y el mapa de características resultante muestra la presencia o ausencia de esa característica en diferentes partes de la imagen [21][22].

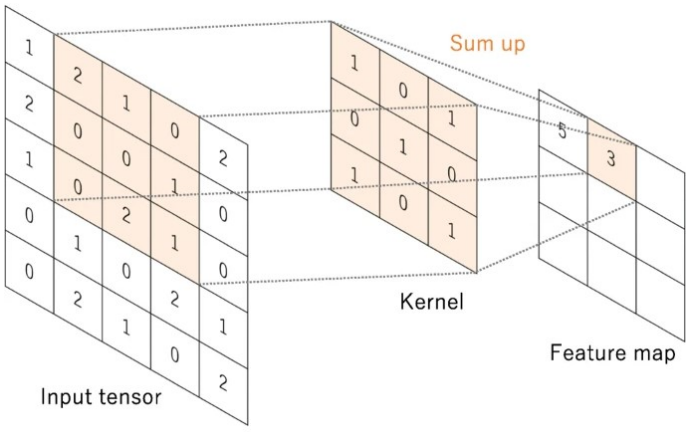


Figura 2.7: Convolución.

Después de cada operación de convolución, la CNN aplica una función de activación no lineal, como la función rectificadora (ReLU), al mapa de características. Al introducir no linealidad en el

modelo, se amplia su capacidad para aprender representaciones más sofisticadas y expresivas de los datos.

A medida que se avanza en la red neuronal convolucional, las capas de convolución se apilan, lo que permite que la red aprenda características cada vez más complejas a partir de la imagen de entrada.

2.2.4.2. Capa de agrupamiento

Las **capas de agrupamiento o pooling** realizan una operación de reducción de muestreo en los mapas de características generados por las capas de convolución. Se utiliza para reducir la dimensionalidad en el plano de los mapas de características y lograr una invariancia de traslación a pequeños desplazamientos y distorsiones [21].

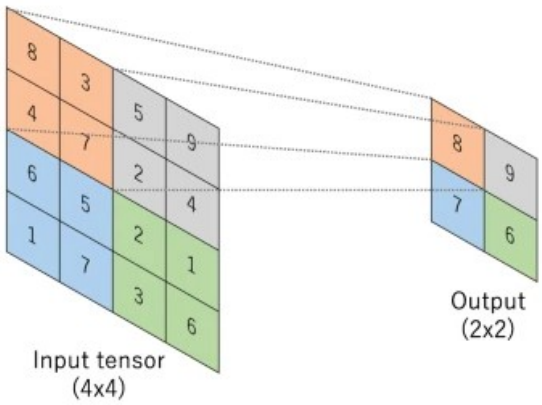


Figura 2.8: Pooling.

La forma más popular de operación de agrupamiento es el **Max Pooling**. Esta operación divide los mapas de características de entrada en regiones no superpuestas y selecciona el valor máximo de cada una, conservando solamente las características más destacadas de cada región. Con este proceso se reduce el tamaño del mapa de características así como la cantidad de parámetros "aprendibles" posteriores de la red [22].

Otra operación de agrupamiento común es el **Global Average Pooling**, que reduce el tamaño del mapa de características a una matriz de 1x1 simplemente tomando la media de todos los elementos de cada mapa de características. Esta operación suele aplicarse una sola vez antes de las capas totalmente conectadas [22].

Aunque se pierde mucha información en la capa de pooling, también tiene una serie de ventajas para la CNN, como la reducción de la complejidad y del costo computacional, mejora de la eficiencia y reducción del riesgo de overfitting.

2.2.4.3. Capa totalmente conectada

La **capa totalmente conectada**, también llamada capa densamente conectada, se establece después de las capas de convolución o agrupamiento y conecta todas las neuronas de una capa con todas

las neuronas de la capa siguiente. Estas capas utilizan las características extraídas a través de las capas anteriores y sus diferentes filtros para realizar la tarea de clasificación [22].

Por lo general, la última capa totalmente conectada tiene el mismo número de neuronas de salida que de clases o predicciones y está seguida de una función no lineal, como Sigmoid para clasificaciones binarias y SoftMax para clasificaciones multiclase.

2.2.5. Transfer Learning

El **Transfer Learning**, también conocido como Aprendizaje por Transferencia, es una técnica ampliamente utilizada en el campo del aprendizaje automático y las redes neuronales. Esta técnica consiste en reutilizar un modelo previamente entrenado y todo el conocimiento adquirido en un nuevo problema relacionado. En lugar de comenzar el proceso de entrenamiento desde cero, se reutilizan los pesos y patrones aprendidos en las capas iniciales y medias del modelo pre-entrenado, mientras reentrenamos o adaptamos las capas posteriores para la tarea en cuestión [23].

El transfer learning presenta ciertas ventajas respecto a crear una red neuronal desde cero [24]:

- **Escasez de datos:** En muchos casos es difícil recopilar un conjunto de datos lo suficientemente grande y diverso para entrenar una red neuronal desde cero. El transfer learning permite utilizar modelos pre-entrenados que han aprendido características generales en grandes conjuntos de datos, por lo tanto, al reutilizar el modelo no se necesitará contar con un conjunto de datos elevado y se evitará el problema de overfitting.
- **Eficiencia computacional:** Entrenar una red neuronal desde cero puede requerir una gran cantidad de recursos computacionales, incluyendo tiempo y potencia de cálculo. Al aprovechar un modelo pre-entrenado, podemos ahorrar tiempo y recursos al iniciar el entrenamiento desde un punto inicial más cercano a la solución óptima.
- **Mejora del rendimiento:** Al utilizar transfer learning, es posible obtener un rendimiento superior al que tendríamos si empleásemos una red neuronal desde cero, especialmente si se cuenta con cantidades de datos escasos. Los modelos pre-entrenados tienen una comprensión más amplia de características y patrones, lo cual puede llevar a una mejor generalización y resultados más precisos en la tarea específica que estamos abordando.

El transfer learning **en redes convolucionales** tiene un impacto significativo en el proceso de entrenamiento y en el rendimiento final del modelo. Cuando se utiliza transfer learning, las capas anteriores de una red convolucional pre-entrenada ya han aprendido a detectar características generales en imágenes, como bordes, texturas, formas y patrones simples. Además, como estos modelos suelen haber sido entrenados en conjuntos de datos masivos que contienen una amplia variedad de imágenes, ya han adquirido un conocimiento profundo sobre las características visuales comunes en imágenes del mundo real. Todo esto acelera el proceso de entrenamiento y mejora el rendimiento del modelo, ya que las capas posteriores solo necesitan aprender a relacionar estas características generales con la tarea específica, en lugar de comenzar desde cero. Por lo tanto, al aprovechar estas características previamente aprendidas, el modelo puede adaptarse rápidamente a nuevas tareas, reconocer patrones visuales relevantes y tendrá una gran capacidad de generalización sin requerir una cantidad masiva de datos de entrenamiento.

Capítulo 3

Materiales y Métodos

3.1. Dataset empleado

La obtención de imágenes de rebecos con sarna se realizó en colaboración con el Director Técnico de Reservas de Caza de León Juan Carlos Peral Sánchez, quien proporcionó las imágenes necesarias para esta investigación. Estas imágenes son difíciles de encontrar en Internet, por lo que su contribución fue fundamental. Por otro lado, las imágenes de rebecos sin sarna se consiguieron a través de la búsqueda de imágenes de alta calidad por Internet [25] [26] [27].

Es importante que las imágenes del dataset tengan alta calidad y estén en el formato correcto. Además, tienen que ser representativas de las situaciones en las que estos animales se encontrarían. Por ello, se comenzó con aproximadamente 400 imágenes de cada categoría (rebecos con sarna y rebecos sin sarna) y se redujo su número a alrededor de 365-370 imágenes de cada una. Se buscó un equilibrio entre ambas categorías o clases, es decir, que la cantidad de imágenes en cada categoría sea similar, y así garantizar que el modelo pueda clasificar correctamente.

El dataset se dividió en: entrenamiento, validación y test. El conjunto de entrenamiento comprende el 70 % de las imágenes y se utiliza para entrenar el modelo. El conjunto de validación, que representa el 20 %, se utiliza para evaluar el rendimiento del modelo durante el entrenamiento. Y, por último, el conjunto de prueba o test, que corresponde al 10 % y se utiliza para evaluar el modelo una vez que ha sido entrenado y comprobar su capacidad para realizar predicciones correctas en imágenes que no ha visto previamente.

Figura 3.1: Algunas imágenes de rebeco con sarna del dataset.

Figura 3.2: Algunas imágenes de rebeco sin sarna del dataset.

3.2. Herramientas y tecnologías utilizadas

3.2.1. Google Colab

Google Colaboratory, también conocido como **Google Colab**, es un entorno gratuito de Jupyter Notebooks que permite ejecutar código en la nube, sin necesidad de instalar software en tu computadora [28]. Ofrece potentes recursos computacionales como GPUs y CPUs de forma gratuita, posibilitando el entrenamiento de redes neuronales complejas con grandes conjuntos de datos, independientemente del equipo con el que se cuente. Al aprovechar los recursos de Google, se consigue acelerar el proceso de creación y desarrollo de los proyectos de aprendizaje automático, sin que esto afecte al rendimiento de los procesadores locales ni que se genere un aumento de la temperatura en la computadora [29].

Figura 3.3: Logo Google Colab [28].

Google Colab tiene preinstaladas herramientas y bibliotecas de Python muy útiles para el aprendizaje automático, como TensorFlow, Keras o PyTorch. Además, Colab se integra con Google Drive, permitiendo el almacenamiento de notebooks y archivos en esta plataforma. También es posible compartir dichos archivos y colaborar a tiempo real con otros usuarios en proyectos de trabajo en equipo.

La versión gratuita de Google Colab presenta limitaciones en el tiempo de ejecución así como de almacenamiento, lo que puede afectar al desarrollo de proyectos complejos y extensos.

3.2.2. Google Drive

Google Drive es un servicio de almacenamiento en la nube ofrecido por Google [30]. Esta plataforma proporciona un espacio de almacenamiento en la nube donde los usuarios pueden guardar sus archivos, como documentos, hojas de cálculo, presentaciones, imágenes y vídeos, garantizando su disponibilidad y seguridad.

Figura 3.4: Logo Google Drive [30].

Google Drive tiene sincronización automática, lo que significa que todo el contenido se almacena y se guarda en los servidores de Google de manera inmediata. Por lo tanto, se podrá acceder a la última versión de cada archivo desde cualquier dispositivo que tenga acceso a él.

Además, esta plataforma se integra con otras herramientas y servicios de Google, como Google Docs o Google Slides. Esto nos permite cargar y editar documentos e imágenes almacenados en Google Drive directamente desde Google Colab, lo cual agiliza el proceso de tratamiento de datos y ayuda con el desarrollo de modelos de aprendizaje automático [31].

3.2.3. Python

Para el desarrollo de la red neuronal he utilizado el lenguaje de programación **Python**. Este lenguaje interpretado de alto nivel se ha convertido en el más utilizado en el campo del aprendizaje automático y la inteligencia artificial [32].

Figura 3.5: Logo Python [32].

Dado que Python es un lenguaje de programación muy legible y fácil de usar, la creación de redes neuronales personalizadas y la implementación de algoritmos de aprendizaje profundo se convierte en una tarea más sencilla [33].

Python ofrece una amplia selección de bibliotecas y frameworks especializados aplicables al campo de aprendizaje automático, como TensorFlow, Keras, PyTorch y scikit-learn. Estas opciones reducen la complejidad de tareas como la manipulación de datos, el entrenamiento de modelos y la evaluación de resultados, puesto que proporcionan una amplia variedad de herramientas y funcionalidades para el desarrollo de modelos de redes neuronales.

3.2.4. TensorFlow

TensorFlow es una biblioteca de aprendizaje automático de código abierto desarrollada por Google que permite la creación y entrenamiento de redes neuronales y otros modelos de aprendizaje automático. Se trata de una herramienta muy útil debido a su flexibilidad, escalabilidad y facilidad de uso. Tensorflow es compatible con otros lenguajes de programación, como Python, y con otras bibliotecas de aprendizaje automático [34].

Además, existen gran cantidad de recursos disponible para todos los usuarios, como documentación, tutoriales y ejemplos de código que pueden ser muy útiles para aprender a trabajar con esta biblioteca [35].

Figura 3.6: Logo TensorFlow [35].

Tensorflow desempeña un papel fundamental en el campo del aprendizaje automático puesto que proporciona las herramientas necesarias para crear y entrenar modelos de manera eficiente. Esta biblioteca permite a los desarrolladores automatizar tareas y procesos y mejorar la eficiencia en diversas áreas, sin tener que de preocuparse por la infraestructura subyacente o el desarrollo de algoritmos complejos.

3.2.5. Keras

Keras es una biblioteca de aprendizaje profundo de código abierto en Python que se ejecuta sobre frameworks de aprendizaje automático como TensorFlow, Theano y Cognitive Toolkit (CNTK) [36]. Destaca por su simplicidad y eficiencia en la creación y entrenamiento de modelos de aprendizaje profundo, especialmente en redes neuronales convolucionales.

Keras resalta por su diseño modular, rápido y fácil de usar. Esta biblioteca ofrece una API consistente, simple y extensible en la cual definir y manipular modelos de redes neuronales.

Figura 3.7: Logo Keras [36].

Además, al igual que TensorFlow, Keras cuenta con una gran comunidad de usuarios y desarrolladores que proporcionan soporte y recursos útiles, como tutoriales, ejemplos de código y modelos pre-entrenados.

3.2.5.1. Keras Tuner

El ajuste de los hiperparámetros puede resultar complicado en el contexto del aprendizaje profundo. Encontrar manualmente las combinaciones óptimas de hiperparámetros a base de prueba y error es un proceso lento, subóptimo e ineficiente. Una alternativa a este proceso es utilizar algoritmos de búsqueda de hiperparámetros.

Keras Tuner es una biblioteca de Python que proporciona herramientas para optimizar automáticamente los hiperparámetros de los modelos de aprendizaje automático desarrollados con Keras. Con esta biblioteca se pueden aplicar diferentes estrategias de búsqueda de hiperparámetros, como la búsqueda aleatoria y la búsqueda en hiperredes.

La búsqueda aleatoria selecciona combinaciones de hiperparámetros de forma aleatoria en un rango predefinido y evalúa el rendimiento del modelo. Por otro lado, la búsqueda en hiperredes utiliza algoritmos más sofisticados, como el algoritmo genético, para encontrar las combinaciones óptimas de hiperparámetros de una manera más eficiente que la búsqueda exhaustiva [37].

La utilización de este tipo de algoritmos agiliza y automatiza el proceso de ajuste de hiperparámetros, optimizándose más rápidamente el rendimiento de los modelos desarrollados.

3.2.6. Grid Search

Grid Search, al igual que Keras Tuner, es un enfoque de optimización utilizado para ajustar los hiperparámetros de los modelos de aprendizaje automático. Consiste en explorar exhaustivamente todas las combinaciones posibles de valores de hiperparámetros dentro de un rango predefinido [38].

Para ello, se crea una cuadrícula o ”grid”que abarca todos los valores posibles para cada hiperparámetro que se desee ajustar y se entrena y evalúa el modelo con cada combinación. Se seleccionará la combinación que produce el mejor rendimiento según una determinada métrica de evaluación, como la precisión o la pérdida.

Aunque Grid Search es una técnica sencilla y ampliamente utilizada, puede ser computacionalmente costosa cuando se trata de optimizar muchas combinaciones de hiperparámetros. Sin embargo, tiene la ventaja de ser exhaustivo y garantiza que se evalúen todas las combinaciones posibles [39].

3.3. Modelos pre-entrenados evaluados

Por último, vamos a comentar algunos modelos pre-entrenados que son comúnmente utilizados en el ámbito de las redes neuronales convolucionales. Todos ellos han sido entrenados en el conjunto de datos ImageNet, que consta de millones de imágenes etiquetadas en diversas categorías. Por estas razones, han sido seleccionados con el fin de encontrar el modelo que mejor se adapte al problema que trata este proyecto.

- **DenseNet121:** DenseNet121 destaca por su estructura densamente conectada, donde cada capa está conectada directamente a todas las capas subsiguientes. Esta estructura fomenta el flujo de información y la reutilización de características en toda la red, consiguiendo así un mayor aprovechamiento de los datos y una mejor generalización. Además de Densenet121,

existen otras variantes de la arquitectura DenseNet que se diferencian en profundidad y número de capas, como DenseNet201, DenseNet169, etc [40] [41].

- **VGG16:** Este modelo pre-entrenado se caracteriza por tener una estructura profunda con 16 capas, compuesta principalmente por capas convolucionales y capas completamente conectadas. Este modelo es famoso por su simplicidad y facilidad de comprensión, pero manteniendo un gran rendimiento en diversas tareas de clasificación de imágenes. Existe otra versión del modelo llamada VGG19, que es una extensión del modelo VGG16 y comparte una estructura similar, pero con una mayor profundidad debido a la inclusión de más capas convolucionales [42][43].

- **MobileNet-v2:** MobileNet-v2 es una arquitectura de red neuronal de 53 capas de profundidad diseñada específicamente para aplicaciones con recursos computacionales limitados. Se caracteriza por el uso de operaciones de convolución separables en profundidad, que reducen drásticamente el número de parámetros y la carga computacional sin comprometer significativamente el rendimiento. La versión original MobileNet también utiliza convoluciones separables y es eficiente en tamaño y recursos pero tiene menor precisión [44] [45].

- **ResNet-50:** Este modelo es parte de la familia de modelos ResNet (Redes Residuales) que introdujeron el concepto de bloques residuales. Estos bloques permiten que la red aprenda la diferencia entre las características originales y las características residuales, facilitando el entrenamiento de redes neuronales más profundas. ResNet-50 es conocido por su capacidad para manejar imágenes de alta resolución. A partir de la arquitectura original ResNet-50, se han desarrollado varias variantes que difieren en la profundidad y estructura de la red, como ResNet-18, ResNet-101, etc [46] [47].

La selección de un modelo pre-entrenado u otro dependerá de diferentes factores, como el contexto del problema, el tamaño del dataset, los recursos computacionales y las necesidades específicas del proyecto. En este trabajo se han realizado diversas pruebas y experimentos con todos los modelos para determinar cuál es el que se adapta mejor al problema en cuestión. En el siguiente capítulo se comentarán los resultados obtenidos.

Capítulo 4

Desarrollo y Resultados

4.1. Carga y preprocesamiento del dataset

El primer paso de este proyecto ha sido la carga y tratamiento del dataset. Hemos subido las imágenes a Google Drive separadas en carpetas según su categoría (Rebeco Con Sarna y Rebeco Sin Sarna) y hemos accedido a ellas a través de Google Colab, donde las almacenaremos en una carpeta llamada "dataset".

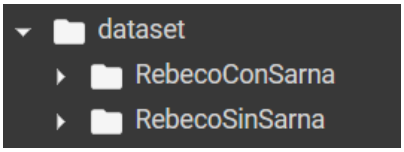


Figura 4.1: Estructura del directorio dataset.

Utilizamos la función `image_dataset_from_directory` que se encargará de escanear el directorio y crear un conjunto de datos de imágenes asignando automáticamente las etiquetas correspondientes de acuerdo al subdirectorio al que pertenecen (Rebeco Con Sarna y Rebeco Sin Sarna). La asignación de las etiquetas facilitará el uso de los datos para el entrenamiento de los modelos. Las imágenes tendrán un tamaño de 224x224 píxeles y un tamaño de lote de 16, por lo tanto, durante el entrenamiento se utilizarán lotes de 16 imágenes a la vez. El tamaño de lote es uno de los hipérparametros que más tarde ajustaremos para optimizar el modelo.

A continuación, creamos un iterador que nos permite acceder a los datos del conjunto de datos de imágenes y obtenemos el siguiente lote de datos del iterador.

```
#Cargamos los datos
import numpy as np
from matplotlib import pyplot as plt

dataset = tf.keras.utils.image_dataset_from_directory('dataset',
    image_size=(224, 224), batch_size=16)

dataset_iterator = dataset.as_numpy_iterator()
batch = dataset_iterator.next()
```

Aplicamos una transformación a los datos del dataset empleando la función `map`, donde se utiliza una función `lambda` que recibe dos argumentos, `x` (imágenes) e `y` (etiquetas), y se normaliza los valores de los píxeles dividiendo entre 255.

```
#Escalamos los datos
data = data.map(lambda x, y: (x / 255, y))
```

Por último, dividimos el dataset en entrenamiento (70 %), validación (20 %) y test (10 %), como comentábamos en Materiales y Métodos.

```
#Dividimos los datos
train_size = int(len(data) * 0.7)
val_size = int(len(data) * 0.2) + 1
test_size = int(len(data) * 0.1)

train = data.take(train_size)
val = data.skip(train_size).take(val_size)
test = data.skip(train_size + val_size).take(test_size)
```

4.2. Elección del modelo

Una vez realizado el preprocesamiento de los datos, pasamos a intentar seleccionar el modelo que utilizaremos. En el Marco Teórico, se discutió la elección de utilizar redes neuronales convolucionales (CNN) para este proyecto, dado que son altamente adecuadas para el procesamiento de imágenes y vídeos. A partir de ahí, surge la cuestión de si es necesario crear una red neuronal desde cero o emplear la técnica de transfer learning. En caso de crear una red neuronal desde cero se ha de escoger minuciosamente determinados parámetros, como el número de capas o de neuronas, hasta conseguir los mejores resultados. Por otro lado, tenemos la opción de reutilizar un modelo pre-entrenado que ya está optimizado y adaptarlo a nuestras necesidades.

A continuación, hablaremos sobre las métricas a través de las cuales determinaremos cuál es el mejor modelo y de los resultados y conclusiones obtenidas con cada modelo probado.

4.2.1. Métricas de evaluación

En este proyecto se realizaron pruebas con múltiples redes neuronales hasta encontrar el modelo que más se adapta al caso en cuestión. A la hora de evaluar y comparar modelos de redes neuronales, existen varias métricas y elementos clave en los que debemos fijarnos para determinar cuál es el mejor. Una forma muy útil de observar el rendimiento del modelo durante el entrenamiento es representar las **funciones de pérdida y precisión**, tanto para el conjunto de entrenamiento como para el de validación. Esto nos permite ver como va evolucionando el modelo en cada época de entrenamiento y observar si se esta produciendo overfitting.

Una vez que hemos entrenado el modelo, es importante comprobar si es capaz de clasificar correctamente imágenes que nunca ha visto antes. Para llevar a cabo esta evaluación, utilizamos el dataset de prueba o test. Existe una herramienta ampliamente utilizada para representar los resultados obtenidos en la predicción: la **Confusion Matrix o Matriz de Confusión**. Esta matriz es una

tabla que muestra la cantidad de ejemplos clasificados correctamente e incorrectamente para cada clase [48].

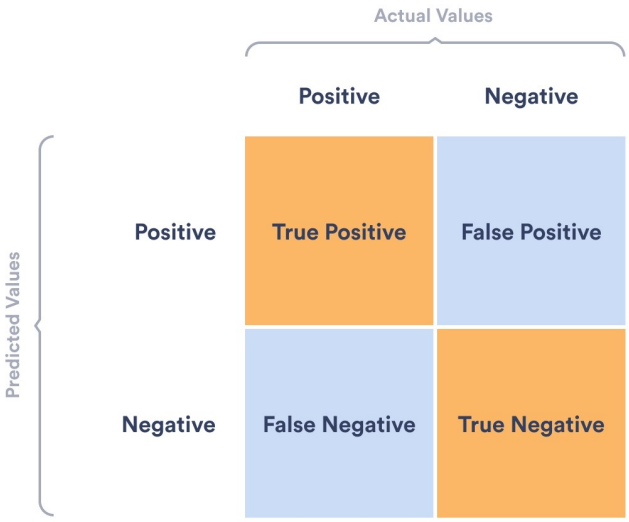


Figura 4.2: Matriz de Confusión.

En la matriz de confusión, las columnas representan las clases reales, mientras que las filas representan las clases predichas por el modelo. Si consideramos que tenemos un modelo binario, donde 0 indica que el rebeco tiene sarna (positivo) y 1 indica que el rebeco no tiene sarna (negativo), el resultado de cada celda quiere decir lo siguiente:

- **Verdaderos Positivos (True Positive, TP):** El modelo predice correctamente que el rebeco de la imagen tiene sarna.
- **Verdaderos Negativos (True Negative, TN):** El modelo predice correctamente que el rebeco no tiene sarna.
- **Falsos Positivos (False Positive, FP):** El modelo predice incorrectamente que el rebeco tiene sarna cuando en realidad no la tiene.
- **Falsos Negativos (False Negative, FN):** El modelo predice incorrectamente que el rebeco no tiene sarna cuando en realidad sí la tiene.

La distribución de estos valores en la matriz nos permite sacar conclusiones sobre el rendimiento del modelo. Si la mayoría de los resultados se encuentran en la diagonal principal de la matriz (TP y TN) significa que nuestro modelo está realizando predicciones precisas.

Una vez obtenida la matriz de confusión, podemos realizar cálculos adicionales que nos permiten evaluar con mayor exactitud la capacidad predictiva de nuestro modelo [48]:

- **Precisión:** La precisión determina el porcentaje de predicciones positivas que son realmente positivas. Una mayor precisión indica que el modelo tiene menos falsos positivos, lo que

significa que tiene una menor tendencia a clasificar incorrectamente un rebeco sin sarna como un rebeco con sarna.

$$Precision = \frac{TP}{TP + FP} \tag{4.1}$$

- **Recall o Sensibilidad:** El recall calcula el porcentaje de casos positivos que se han predicho correctamente, es decir, la tasa de verdaderos positivos. Un mayor recall indica que el modelo tiene menos falsos negativos, lo que significa que tiene una menor tendencia a clasificar incorrectamente un rebeco con sarna como un rebeco sin sarna. En nuestro caso, es más importante tener un recall elevado dado que tener falsos negativos supone no detectar la enfermedad a tiempo y que se extienda rápidamente.

$$Recall = \frac{TP}{TP + FN} \tag{4.2}$$

- **F1 Score:** El F1 Score es una medida que combina la precisión y el recall mediante la media armónica de ambos valores. Toma en cuenta tanto los falsos positivos como los falsos negativos, lo que lo convierte en una métrica adecuada para conjuntos de datos desequilibrados. Proporciona un equilibrio entre la precisión y el recall y es útil para evaluar el rendimiento general del modelo de clasificación.

$$F1Score = \frac{2}{\frac{1}{Precision} + \frac{1}{Recall}} = \frac{2 * Precision * Recall}{Precision + Recall} \tag{4.3}$$

4.2.2. Comparativa de los diferentes modelos

Ahora que ya sabemos como comparar los modelos para determinar el mejor de ellos, vamos a comentar los resultados obtenidos tras su entrenamiento. Para asegurarnos que la comparativa es justa, se han empleado los mismos hiperparámetros en cada uno de los modelos:

Batch Size	Learning Rate	Épocas	Algoritmo
16	0.001 (default)	10	Adam

Tabla 4.1: Valores iniciales.

Los cinco modelos probados y entrenados con nuestro dataset son:

- **Red neuronal personalizada:** Primero se comenzó creando una red neuronal convolucional desde cero utilizando un modelo secuencial. Un modelo secuencial es aquel en el que las capas se agregan secuencialmente una después de la otra. En este caso, el modelo consta de tres capas convolucionales (`Conv2D()`) seguidas de capas de pooling (`MaxPooling2D()`), una capa de aplanamiento (`Flatten()`) y dos capas densamente conectadas (`Dense()`). La capa de aplanamiento transforma una matriz multidimensional en un vector unidimensional para que los datos puedan ser procesados por capas densamente conectadas. Se utiliza la función de activación ReLU en las capas convolucionales y densamente conectadas, excepto

en la capa de salida, donde se aplica la función Sigmoid para la clasificación binaria. En total, se trata de una red neuronal de nueve capas.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D,
    Dense, Flatten

modelo1 = Sequential()

modelo1.add(Conv2D(16, (3,3), 1, activation='relu',
    input_shape=(224,224,3)))
modelo1.add(MaxPooling2D())

modelo1.add(Conv2D(32, (3,3), 1, activation='relu'))
modelo1.add(MaxPooling2D())

modelo1.add(Conv2D(16, (3,3), 1, activation='relu'))
modelo1.add(MaxPooling2D())

modelo1.add(Flatten())

modelo1.add(Dense(256, activation='relu'))
modelo1.add(Dense(1, activation='sigmoid')) #Salida sigmoidal
    para dos clases
```

Compilamos el modelo empleando el optimizador Adam, la función de pérdida `binary_crossentropy` y la métrica de evaluación `accuracy`, es decir, configuramos el modelo para que utilice el optimizador Adam durante el entrenamiento y calcule la pérdida y la precisión durante la evaluación. Seguidamente entrenamos al modelo utilizando los datos de entrenamiento `train` durante 10 épocas y se evalúa el rendimiento en el conjunto de validación `val`. La variable `hist` guardará el historial de entrenamiento. Utilizaremos este mismo código para compilar y entrenar todos los modelos.

```
modelo1.compile('adam', loss='binary_crossentropy', metrics=['
    accuracy'])
hist = modelo1.fit(train, epochs=10, validation_data=val)
```

En la figura 4.3 se representan los resultados de pérdida y precisión obtenidos durante la fase de entrenamiento. Estas gráficas nos permiten seguir la evolución del modelo, observando si es capaz de aprender de los errores y adaptarse. Las líneas azules representan los valores obtenidos durante el entrenamiento con el dataset de entrenamiento, mientras que las líneas naranjas se corresponden a los resultados obtenidos con el conjunto de validación, que se evalúa al finalizar cada época. Es importante comparar los resultados de ambos conjuntos, ya que si hay una gran diferencia entre ellos, podría indicar la presencia de overfitting. En otras palabras, si el modelo tiene buenos resultados en el conjunto de entrenamiento pero tiene un rendimiento considerablemente inferior con el conjunto de validación, significa que el modelo está memorizando los datos de entrenamiento y no está aprendiendo a generalizar bien.

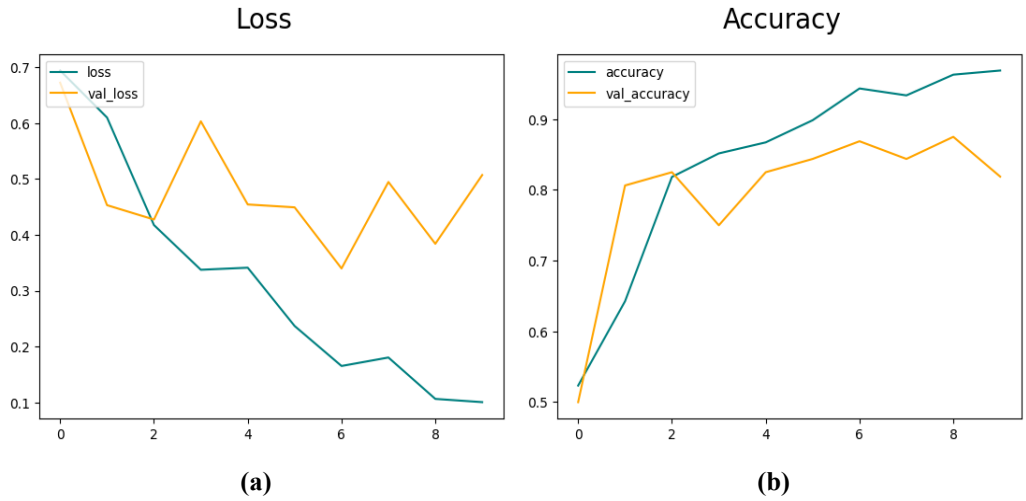
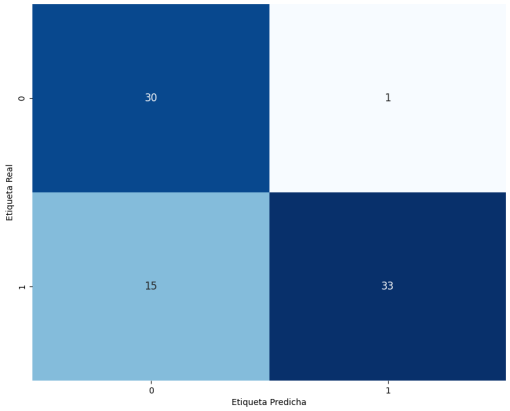


Figura 4.3: Función de pérdida y precisión de la red neuronal personalizada.

En la figura 4.4 observamos la matriz de confusión obtenida con esta red personalizada. Esta matriz se consigue al utilizar el modelo entrenado para predecir y clasificar imágenes que no ha observado durante el entrenamiento o la validación. Como mencionábamos anteriormente, un buen desempeño del modelo se ve reflejado en valores altos de TP y TN en la diagonal principal. Sin embargo, en este caso, se han clasificado correctamente 30 verdaderos positivos (rebeco con sarna) y 33 verdaderos negativos (rebeco sin sarna), pero existen 15 falsos negativos y 1 falso positivo. La presencia de muchos falsos negativos puede indicar limitaciones en la capacidad del modelo para reconocer y capturar patrones específicos relacionados con la sarna, lo que reduce la fiabilidad y efectividad del modelo.



Loss	0.318
Accuracy	0.841
Validation Loss	0.478
Validation Accuracy	0.795
Precision	0.970
Recall	0.687
F1 Score	0.805

Figura 4.4: Matriz de confusión y resultados empleando la red neuronal personalizada.

Por último, en la tabla observamos la pérdida y la precisión promedio de todas las épocas, tanto con el conjunto entrenamiento como con el de validación. Se observa que los resultados con el dataset de entrenamiento son aceptables, pero en la pérdida de validación es de 0.478, valor que puede ser considerado elevado en una clasificación binaria. En la tabla también se proporcionan los resultados de precisión, recall y F1 score, calculados a partir de la matriz de confusión. La precisión es de 0.97, lo que implica que el 97 % de las muestras clasificadas como positivas por el modelo son realmente positivas, mientras que solo el 3 % son falsos

positivos. El recall es de 0.687, lo cual indica que el 68.7 % de las muestras positivas reales fueron correctamente identificadas por el modelo, mientras que el 31.3 % son falsos negativos. Es importante destacar que si bien la precisión puede ser buena en términos generales, se debe tener en cuenta la presencia de falsos negativos, reflejados en la métrica de recall. El hecho de que haya 15 falsos negativos indica que el modelo no está identificando correctamente algunos casos positivos de sarna en los rebecos, lo cual puede tener consecuencias negativas, como la propagación de la enfermedad y la falta de tratamiento en los animales afectados. Por lo tanto, reducir la tasa de falsos negativos es un aspecto crítico a considerar y a mejorar en la detección.

- **DenseNet121:** Este es el primer modelo pre-entrenado evaluado con nuestro dataset. Para poder utilizar el modelo DenseNet121 en nuestro proyecto tenemos que cargar su arquitectura pre-entrenada tal y como mostramos en el código inferior. Los pesos se inicializan utilizando los pesos obtenidos del conjunto de datos de ImageNet y se excluye la capa densa de salida original para adaptarla a nuestro modelo. Además, le indicamos al modelo que las imágenes que estamos utilizando tienen un tamaño de 224x224 píxeles y son a color.

```
from tensorflow.keras.applications.densenet import DenseNet121
from tensorflow.keras import layers

base_model = DenseNet121(
    weights="imagenet",
    include_top=False,
    input_shape=(224, 224, 3)
)
```

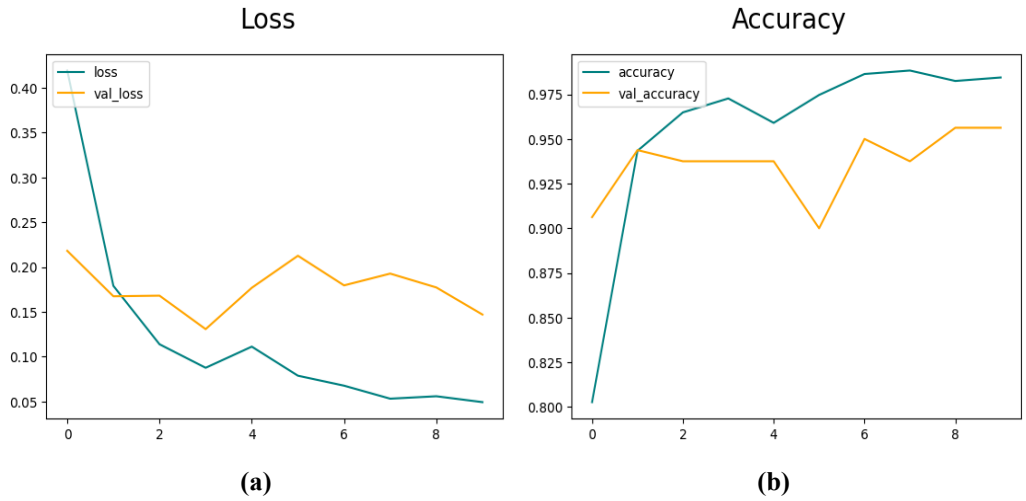
Congelamos todas las capas del modelo original para que no se modifiquen durante el entrenamiento. Esto nos permite aprovechar el conocimiento pre-entrenado de las capas base del modelo.

```
for layer in base_model.layers:
    layer.trainable = False
```

Seguidamente, añadimos las capas necesarias para adaptar el modelo pre-entrenado a nuestro problema en cuestión. Añadimos una capa de pooling para reducir la dimensionalidad de las características extraídas por las capas base, disminuyendo la cantidad de parámetros y capturando solo las características más importantes. Creamos una capa densa con función ReLU, que permite al modelo aprender representaciones más complejas y discriminatorias. Por último, se añade la capa final en la cual se indica que la clasificación es binaria y solo hay dos posibles salidas (Rebeco Con Sarna y Rebeco Sin Sarna).

```
x = layers.GlobalAveragePooling2D()(base_model.output)
x = layers.Dense(256, activation="relu")(x)
outputs = layers.Dense(1, activation="sigmoid")(x)

#Modelo final
modelo2 = tf.keras.Model(inputs=base_model.inputs, outputs=
    outputs)
```

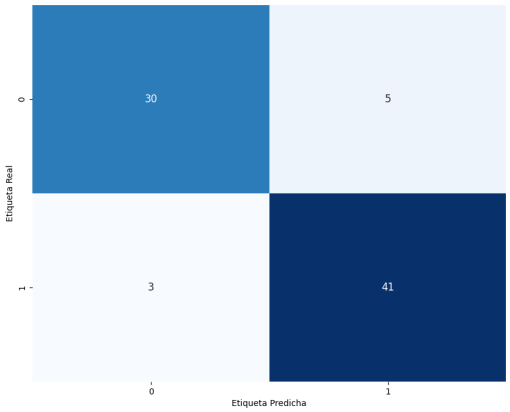
Una vez compilado y entrenado el modelo, obtenemos los resultados mostrados en la figura 4.5.



(a) **(b)**

Figura 4.5: Función de pérdida y precisión del modelo DenseNet121.

El modelo actual muestra una mejora notable en comparación con el modelo anterior. Al observar los valores en la escala vertical de las gráficas de pérdida y precisión, se puede apreciar que tanto el conjunto de entrenamiento como el de validación tienen resultados muy buenos. En la tabla, se puede ver como los valores promedio son muy similares entre el conjunto de entrenamiento y el de validación, lo que indica una menor probabilidad de overfitting. A pesar de ello, es recomendable aplicar técnicas que eviten sobreajuste dado que la curva de precisión con el conjunto de entrenamiento se acerca mucho a 1.



Loss	0.208
Accuracy	0.927
Validation Loss	0.234
Validation Accuracy	0.907
Precision	0.909
Recall	0.857
F1 Score	0.882

Figura 4.6: Matriz de confusión y resultados empleando DenseNet121.

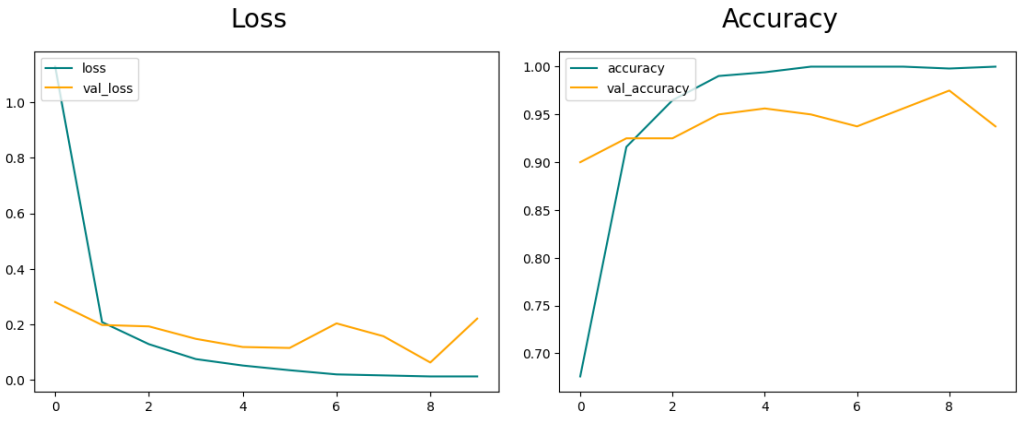
En relación a la matriz de confusión, los resultados son bastante buenos, ya que solo ocho imágenes han sido clasificadas incorrectamente. Esto se refleja en los altos valores de precisión, recall y F1 score.

- **VGG16:** La importación y uso del modelo VGG16 es parecida al modelo anterior. Comenzamos creando una instancia del modelo VGG16 pre-entrenado, especificando que se carguen los pesos pre-entrenados del modelo y que se excluya la capa de clasificación final, ya que se reemplazará con capas personalizadas. Congelamos los pesos de todas las capas del modelo

para evitar que se modifican durante el entrenamiento y agregamos las capas necesarias para adaptar el modelo a nuestra tarea de clasificación binaria.

```
from tensorflow.keras.applications.vgg16 import VGG16
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.models import Model

base_model = VGG16(weights='imagenet', include_top=False,
    input_shape=(224, 224, 3))

for layer in base_model.layers:
    layer.trainable = False

x = Flatten()(base_model.output)
x = Dense(256, activation='relu')(x)
x = Dense(1, activation='sigmoid')(x)

#Modelo final
modelo3 = Model(inputs=base_model.input, outputs=x)
```

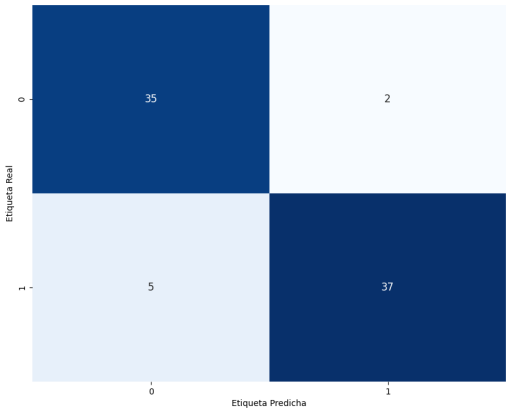
Al evaluar el rendimiento del modelo VGG16, se llegaron a las siguientes conclusiones:



(a) **(b)**

Figura 4.7: Función de pérdida y precisión del modelo VGG16.

El modelo VGG16 obtiene unos resultados excelentes, los mejores hasta el momento. La pérdida de entrenamiento y validación es bastante baja, ambos alrededor de 0.168, y la precisión es muy alta, alcanzando 0.954 y 0.941, respectivamente. Al igual que el modelo anterior, sería recomendable aplicar técnicas que eviten la probabilidad de que aparezca overfitting. La matriz de confusión tiene unos resultados prometedores, con solo siete imágenes clasificadas de manera incorrecta. La precisión es de 0.949, el recall es 0.881, y el F1 score es de 0.914.



Loss	0.168
Accuracy	0.954
Validation Loss	0.169
Validation Accuracy	0.941
Precision	0.949
Recall	0.881
F1 Score	0.914

Figura 4.8: Matriz de confusión y resultados empleando VGG16.

- **MobileNet-v2:** Para importar este modelo se ha decidido tomar un enfoque diferente. En lugar de cargar el modelo directamente desde la biblioteca de TensorFlow.keras, se ha utilizado TensorFlow Hub, una plataforma que ofrece gran cantidad de modelos reutilizables para tareas de aprendizaje automático. En este caso, hemos utilizado la URL del modelo MobileNet-v2 para importarlo. El enlace empleado apunta a una versión que ya no incluye la capa de clasificación final. Además, hemos desactivado la capacidad de entrenamiento de las capas del modelo y, por último, hemos añadido una capa densa que permite la clasificación binaria.

```
import tensorflow_hub as hub

url = "https://tfhub.dev/google/tf2-preview/mobilenet_v2/
    feature_vector/4"
mobilenetv2 = hub.KerasLayer(url, input_shape=(224,224,3))

mobilenetv2.trainable = False

#Modelo final
modelo4 = tf.keras.Sequential([
    mobilenetv2,
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

Los resultados obtenidos al analizar el funcionamiento del modelo MobileNet-v2 se muestran en las figuras 4.9 y 4.10. En la tabla podemos ver los valores promedio, los cuales son bastante buenos: pérdida en torno a 0.2 y precisión de aproximadamente 0.9, en ambos conjuntos de datos. En la matriz de confusión podemos ver las predicciones realizadas por el modelo. En general, los resultados son satisfactorios pero inferiores a los dos modelos anteriores.

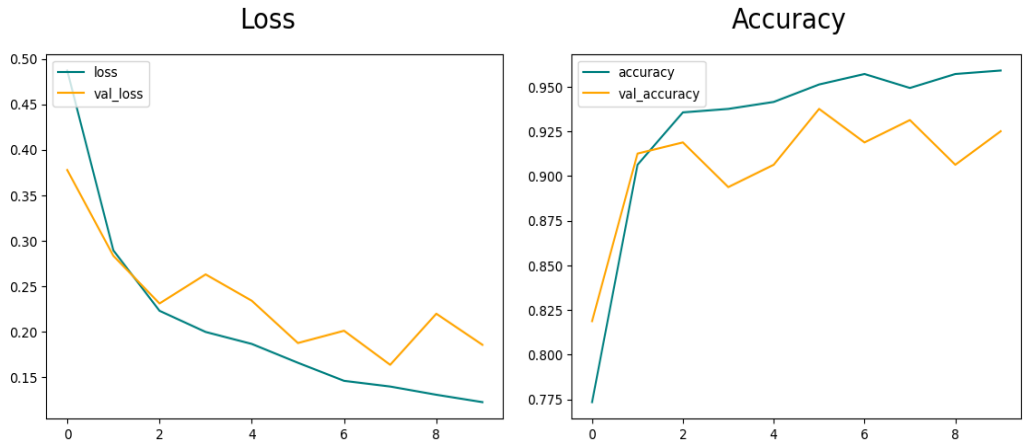
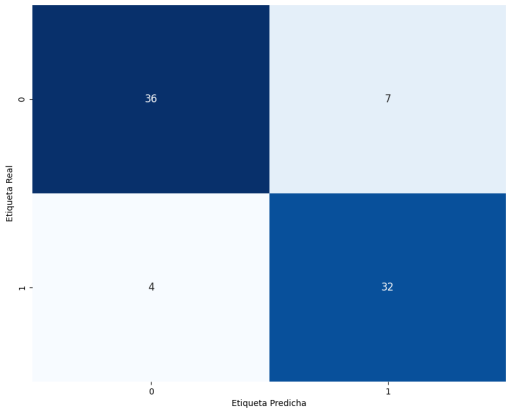


Figura 4.9: Función de pérdida y precisión del modelo MobileNet-v2.



Loss	0.209
Accuracy	0.927
Validation Loss	0.234
Validation Accuracy	0.907
Precision	0.821
Recall	0.889
F1 Score	0.853

Figura 4.10: Matriz de confusión y resultados empleando MobileNet-v2.

- **ResNet-50:** Para la utilización del modelo ResNet-50 hemos utilizado una técnica similar a la del modelo DenseNet121. Hemos creado una instancia del modelo pre-entrenado, reutilizando los pesos del modelo base al congelarlos y lo hemos adaptado para una clasificación binaria.

```
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.layers import Dense,
    GlobalAveragePooling2D
from tensorflow.keras.models import Model

base_model = ResNet50(weights='imagenet', include_top=False,
    input_shape=(224, 224, 3))

for layer in base_model.layers:
    layer.trainable = False
```

```
x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(256, activation='relu')(x)
predictions = Dense(1, activation='sigmoid')(x)

#Modelo final
modelo5 = Model(inputs=base_model.input, outputs=predictions)
```

Por último, comentaremos las conclusiones extraídas sobre el modelo ResNet-50.

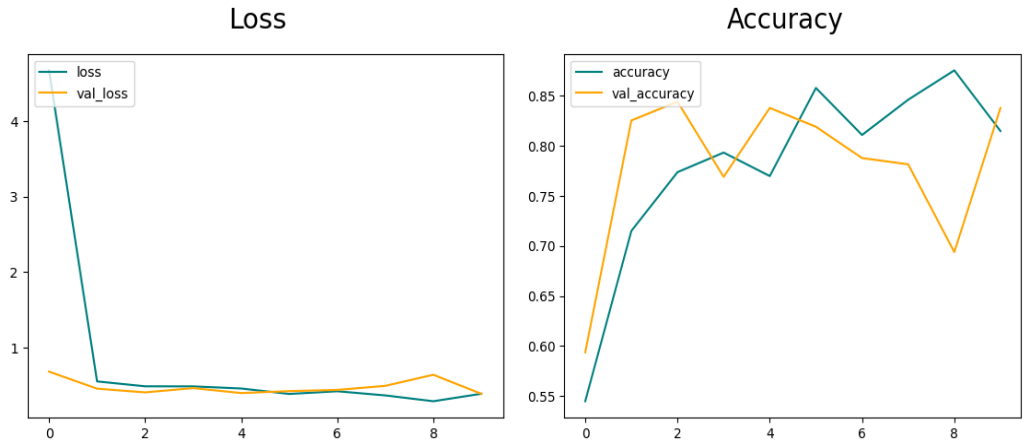
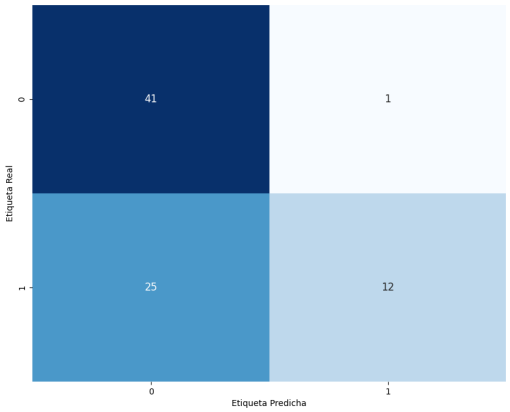


Figura 4.11: Función de pérdida y precisión del modelo ResNet-50.



Loss	0.855
Accuracy	0.779
Validation Loss	0.484
Validation Accuracy	0.779
Precision	0.923
Recall	0.324
F1 Score	0.480

Figura 4.12: Matriz de confusión y resultados empleando ResNet-50.

Al observar las gráficas, podemos notar que la pérdida de entrenamiento y de validación tienen un patrón similar en términos de evolución a lo largo de las épocas. Sin embargo, al examinar su valor promedio, se puede apreciar que ambos presentan valores elevados: 0.855 para la pérdida de entrenamiento y 0.484 para la pérdida de validación. Es importante destacar que el promedio de pérdida del conjunto de entrenamiento es significativamente más alto debido a una pérdida inicial elevada en la primera época. Pero, aunque la pérdida disminuye en etapas posteriores, sigue siendo alta en comparación con otros modelos. Además, la

precisión obtenida es inferior a la lograda anteriormente.

En cuanto a la matriz de confusión y los resultados de precisión y recall, se observa un desempeño mixto del modelo. Por un lado, se logran altos valores de precisión, lo cual significa que hay pocos falsos positivos. Sin embargo, el recall es relativamente bajo, lo que implica que el modelo tiene dificultades para detectar todas las instancias positivas reales. Estos resultados indican que el modelo no está prediciendo correctamente muchas instancias de rebecos con sarna.

Una vez analizado el rendimiento de todos los modelos, podemos concluir que, en general, es más beneficioso utilizar modelos pre-entrenados en lugar de crear una red neuronal desde cero. Esto se debe a que estos modelos han sido entrenados en conjuntos de datos masivos y diversos, lo que les permite capturar características generales y relevantes para una amplia gama de tareas de visión por computadora y ya han pasado por un proceso de ajuste fino en tareas similares, lo que les permite adaptarse mejor a nuestro problema específico.

Entre todos los modelos evaluados, el VGG16 y el DenseNet121 son los que más destacan. Me he decantado finalmente por seleccionar el modelo VGG16 para el desarrollo de este proyecto porque que ha demostrado ser el más prometedor en términos de pérdida y precisión y tiene unos resultados excelentes en la matriz de confusión. Basándonos en su buen rendimiento y capacidad de generalización, el **modelo VGG16** se convierte la opción más adecuada para abordar nuestro problema.

4.3. Optimización del modelo escogido

Una vez seleccionado el modelo VGG16, vamos a intentar conseguir los mejores resultados posibles utilizando diferentes métodos de optimización.

Los resultados de este modelo son excelentes: pérdida muy baja y precisión muy alta. Cuando los resultados son demasiado buenos y las curvas de precisión se acercan demasiado a 1, puede surgir la duda de si se esta produciendo overfitting. Por esta razón, se decidió aplicar diferentes técnicas que ayuden a reducir la probabilidad de que aparezca. Comenzamos con una de las técnicas mencionadas anteriormente en el apartado de Overfitting y Underfitting: añadir **data augmentation**. Data augmentation suponer aplicar diferentes transformaciones a las imágenes como las siguientes:

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=10,
    width_shift_range=0.1,
    height_shift_range=0.1,
    zoom_range=[0.8, 1.2],
    brightness_range=[0.8, 1.2],
    horizontal_flip=True,
    vertical_flip=True,
    validation_split=0.2
)
```

```
#Dataset de entrenamiento y validación
train = datagen.flow_from_directory('/content/dataset/',
    target_size=(224,224), batch_size=16, shuffle=True, subset='
    training', class_mode='binary')

val = datagen.flow_from_directory('/content/dataset/', target_size
    =(224,224), batch_size=16, shuffle=True, subset='validation',
    class_mode='binary')
```

En este ejemplo se muestra una rotación aleatoria de la imagen, desplazamientos horizontales y verticales aleatorios, zoom, cambio del brillo y volteo horizontales y verticales aleatorios. Además, se normaliza el valor de los píxeles dividiéndolos por 255 y se divide el conjunto de datos en entrenamiento (80 %) y validación (20 %). Para poder aplicar data augmentation se necesita emplear `ImageDataGenerator`, una clase de la biblioteca `tensorflow.keras.preprocessing.image`. `ImageDataGenerator` solo puede dividir los datos en entrenamiento (`train`) y validación (`val`), por lo que no tendremos dataset de test. Se realizaron diferentes pruebas variando las transformaciones aplicadas y sus valores, pero al final se concluyó que se obtienen mejores predicciones **sin** aplicar data augmentation.

Por otro lado, se debían optimizar los valores de los **hiperparámetros**: batch size o tamaño de lote, learning rate o tasa de aprendizaje (lr) y número de épocas, así como seleccionar el mejor **algoritmo de optimización**.

Para la selección de los hiperparámetros se realizaron múltiples pruebas con todas las posibles combinaciones. Debido a que la modificación manual es un proceso demasiado lento, se decidió utilizar dos técnicas que aceleraron el proceso: Grid Search y Keras Tuner, de las cuales hablamos en el apartado de Materiales y Métodos.

Para utilizar **Grid Search**, primero tenemos que importar las bibliotecas `GridSearchCV` y `KerasClassifier`. Definimos la variable `param_grid` que contiene los diferentes valores de los hiperparámetros que se probarán durante la búsqueda en cuadrícula. En este caso, vamos a intentar optimizar los valores de `batch_size`, `learning_rate` y `epochs`.

```
from sklearn.model_selection import GridSearchCV
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier

param_grid = {
    'batch_size': [8, 16, 32, 64],
    'learning_rate': [0.001, 0.01, 0.1],
    'epochs': [5, 10, 20, 25]
}
```

A continuación, definimos una función para establecer el modelo que queremos optimizar. Le pasamos un valor de `learning_rate` como argumento para configurar el optimizador de la función pero, una vez comience la búsqueda, se irá ajustando a los valores definidos en `param_grid`.

```
from tensorflow.keras.applications.vgg16 import VGG16
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.models import Model

def create_vgg_model(learning_rate=0.001):
```

```
base_model = VGG16(weights='imagenet', include_top=False,
    input_shape=(224, 224, 3))

for layer in base_model.layers:
    layer.trainable = False

x = Flatten()(base_model.output)
x = Dense(1, activation='sigmoid')(x)

modelo = Model(inputs=base_model.input, outputs=x)

optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
modelo.compile(optimizer=optimizer, loss='binary_crossentropy',
    metrics=['accuracy'])

return modelo
```

Envolvemos el modelo de Keras en un objeto de Scikit-learn para poder utilizar el modelo dentro de la función `GridSearchCV`. Creamos un objeto de `GridSearchCV` con `cf` como estimador, especificando que use el modelo de la función, los diferentes valores definidos en `param_grid` y que se utilice la precisión como métrica de evaluación. El parámetro `cv=3` significa que se va a realizar una validación cruzada de 3 pliegues, es decir, que se dividan los datos en 3 conjuntos de entrenamiento y prueba diferentes, de esta manera la evaluación será más robusta. Es necesario comentar que para el uso de Grid Search se debe utilizar `ImageDataGenerator`.

Por último, se llama al método `fit` para ejecutar la búsqueda en cuadrícula. Para ello, le proporcionamos las imágenes y sus etiquetas correspondientes.

```
model = KerasClassifier(build_fn=create_vgg_model)
clf = GridSearchCV(
       estimator=model, param_grid=param_grid, scoring='accuracy',
           cv=3)

clf.fit(images, labels)
```

Una vez finalizado la búsqueda, podemos analizar sus resultados a través de los siguientes atributos: `clf.cv_results_` que almacena todos los resultados de cada combinación probada, `clf.best_params_` que nos devuelve la combinación óptima de hiperparámetros que tiene el mejor rendimiento y `clf.best_score_` que nos devuelve el puntaje promedio obtenido por el mejor modelo en términos de la métrica de evaluación, que es la precisión en este caso.

La ejecución de la búsqueda tomó mucho tiempo, por lo tanto se consiguieron sacar algunas conclusiones, pero no se pudo probar todas las combinaciones que se muestran en `param_grid`. Por esta razón, se decidió probar otra técnica: **Keras Tuner**. Además, también se quería aplicar otras técnicas que redujesen la probabilidad de overfitting y optimizarlas a través Keras Tuner.

Las técnicas de reducción de overfitting utilizadas fueron el **dropout** y la **regularización L2**. El dropout se añadió justo antes de la última capa de nuestro modelo VGG16 y la regularización en la

capa densa con función de activación ReLU. Estas modificaciones se aplicaron al modelo antes de iniciar la optimización de Keras Tuner y se observó una mejora de los resultados. Por lo tanto, había que conseguir encontrar la combinación de ambas que produjese resultados óptimos. Para este cometido se emplearía Keras Tuner, quien también se encargaría de optimizar los hiperparámetros `learning_rate` y `batch_size`.

Para conseguir todo esto comenzamos definiendo una clase llamada `MyHyperModel`, que hereda de `kt.HyperModel`, la clase base para la definición de hipermodelos en Keras Tuner. Dentro de esta clase, definimos el método `build()` que nos construye el modelo de la red neuronal. Aquí es donde añadimos el código del modelo VGG16 con las modificaciones propias del dropout y la regularización. Para la optimización hemos declarado tres posibles valores de regularización L2, dos de `learning_rate` y hemos definido un rango de valores para el dropout. No podemos probar demasiadas combinaciones dado que el proceso se haría demasiado largo.

A continuación, se define el método `fit()` para entrenar el modelo. En este método se declaran los tres posibles valores de `batch_size` que vamos a utilizar en la optimización.

```
import tensorflow as tf
from tensorflow.keras.applications.vgg16 import VGG16
from tensorflow.keras.layers import Dense, Flatten, Dropout
from tensorflow.keras.models import Model
from tensorflow.keras.regularizers import l1, l2
import tensorflow_hub as hub
import keras_tuner as kt

class MyHyperModel(kt.HyperModel):
    def build(self, hp):

        base_model = VGG16(weights='imagenet', include_top=False,
            input_shape=(224, 224, 3))

        for layer in base_model.layers:
            layer.trainable = False

        x = Flatten()(base_model.output)
        x = Dense(256, activation='relu', kernel_regularizer=l2(hp.
            Choice('l2_regularization', values=[0.0, 0.001, 0.01])))(
            x)
        x = Dropout(hp.Float('dropout_rate', min_value=0.1,
            max_value=0.5, step=0.1))(x)
        x = Dense(1, activation='sigmoid')(x)

        model = Model(inputs=base_model.input, outputs=x)
        model.compile(optimizer=tf.keras.optimizers.Adam(hp.Choice('
            learning_rate', values=[0.01, 1e-4])),
                      loss='binary_crossentropy',
                      metrics=['accuracy'])

        return model
```

```
    def fit(self, hp, model, *args, **kwargs):
        return model.fit(*args,
                         batch_size=hp.Choice("batch_size", [8, 16,
                             32]),
                         **kwargs)

tuner = kt.RandomSearch(
    MyHyperModel(),
    objective="val_loss",
    max_trials=25
)
```

Por último, se crea un objeto RandomSearch para la búsqueda a aleatoria de Keras Tuner, que utiliza el hipermodelo personalizado MyHyperModel, especificando el número máximo de pruebas y que el objetivo es la minimización de la pérdida del conjunto de validación. Cuanto más alto es el número de pruebas, más probable es que se prueben todas las combinaciones pero, como esto llevaría mucho tiempo y recursos computacionales, Keras Tuner está diseñado para que si este número no es elevado, se prioricen y ejecuten primero las combinaciones más prometedoras.

Después de aplicar todas estas técnicas, se llegó a la conclusión de que el modelo más optimizado tendría los siguientes valores:

L2	Dropout	Batch Size	Learning Rate	Épocas	Algoritmo
0.001	0.3	32	0.001	7	Adam

Tabla 4.2: Valores optimizados.

Por lo tanto, el modelo final queda de la siguiente manera:

```
from tensorflow.keras.applications.vgg16 import VGG16
from tensorflow.keras.layers import Dense, Flatten, Dropout
from tensorflow.keras.models import Model
from keras.regularizers import l1, l2

base_model = VGG16(weights='imagenet', include_top=False,
    input_shape=(224, 224, 3))

for layer in base_model.layers:
    layer.trainable = False

x = Flatten()(base_model.output)
x = Dense(256, activation='relu', kernel_regularizer=l2(0.001))(x)
    #Añadimos regularización L2
x = Dropout(0.3)(x) #Añadimos dropout
x = Dense(1, activation='sigmoid')(x)

modeloFinal = Model(inputs=base_model.input, outputs=x)
```

Tras optimizar nuestro modelo, obtenemos los resultados de la figura 4.13.

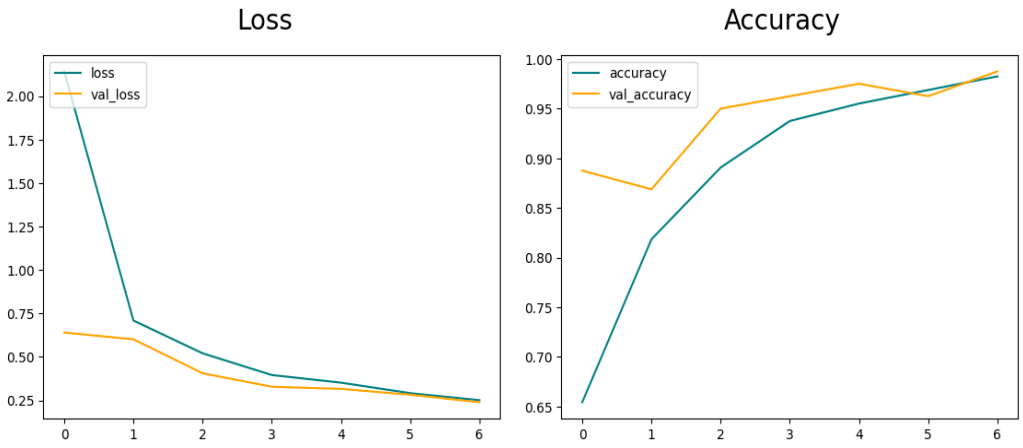


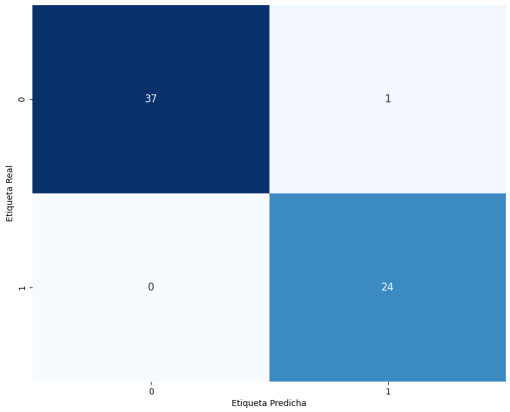
Figura 4.13: Función de pérdida y precisión del modelo VGG16 optimizado.

Las funciones de pérdida y validación han sido optimizadas para obtener los mejores resultados posibles, pero evitando la aparición de overfitting al aplicar dropout y regularización L2. Al aplicar estas técnicas, es común observar un aumento inicial en la pérdida y una reducción de la precisión debido a la mayor variabilidad y restricciones en el ajuste del modelo. Esto explica por qué la pérdida y precisión promedio de la tabla 4.15 pueden parecer peores a las anteriores. Sin embargo, como se observa en la figura 4.13, a medida que el modelo se adapta, los resultados mejoran, consiguiendo una mayor capacidad de generalización y, por lo tanto, convirtiéndose en un mejor clasificador. En la figura 4.14 se muestran los valores específicos de precisión y pérdida durante el proceso de entrenamiento.

```
Epoch 1/7
16/16 [==============================] - 47s 2s/step - loss: 2.1421 - accuracy: 0.6543 - val_loss: 0.6404 - val_accuracy: 0.8875
Epoch 2/7
16/16 [==============================] - 34s 2s/step - loss: 0.7093 - accuracy: 0.8184 - val_loss: 0.6013 - val_accuracy: 0.8687
Epoch 3/7
16/16 [==============================] - 33s 2s/step - loss: 0.5208 - accuracy: 0.8906 - val_loss: 0.4062 - val_accuracy: 0.9500
Epoch 4/7
16/16 [==============================] - 38s 2s/step - loss: 0.3960 - accuracy: 0.9375 - val_loss: 0.3288 - val_accuracy: 0.9625
Epoch 5/7
16/16 [==============================] - 34s 2s/step - loss: 0.3522 - accuracy: 0.9551 - val_loss: 0.3164 - val_accuracy: 0.9750
Epoch 6/7
16/16 [==============================] - 35s 2s/step - loss: 0.2917 - accuracy: 0.9688 - val_loss: 0.2829 - val_accuracy: 0.9625
Epoch 7/7
16/16 [==============================] - 35s 2s/step - loss: 0.2515 - accuracy: 0.9824 - val_loss: 0.2396 - val_accuracy: 0.9875
```

Figura 4.14: Pérdida y precisión del modelo VGG16 optimizado en cada época.

Por último, queda comentar la matriz de confusión. Como podemos observar en la figura 4.15, los resultados son casi perfectos, solo se clasifica mal una de las imágenes. Esto quiere decir que nuestro modelo es capaz de generalizar y realiza predicciones muy precisas. De esta manera, obtenemos una precisión de 0.973, un recall perfecto, con valor de 1.0 y un F1 score de 0.987.



Loss	0.666
Accuracy	0.887
Validation Loss	0.402
Validation Accuracy	0.942
Precision	0.973
Recall	1.0
F1 Score	0.987

Figura 4.15: Matriz de confusión y resultados finales del modelo VGG16 optimizado.

Basándonos en los resultados obtenidos, se puede concluir que el modelo desarrollado ha demostrado un rendimiento sobresaliente en la tarea de clasificación. La alta precisión y el recall perfecto indican que el modelo es capaz de realizar predicciones precisas y de identificar correctamente todas las muestras positivas.

Esto sugiere que el modelo tiene una capacidad excepcional para generalizar a partir de los datos de entrenamiento y realizar predicciones en datos no vistos. Estas características son muy valiosas, especialmente en este proyecto donde la detección correcta de casos positivos de sarna es crítica.

Capítulo 5

Conclusiones y Líneas Futuras

5.1. Conclusiones

Finalmente, comentaremos las conclusiones extraídas al realizar este proyecto, analizando los resultados obtenidos y la relevancia de un trabajo como este en el campo de la inteligencia artificial y de la salud.

La sarna es una enfermedad catastrófica que afecta seriamente a los animales, especialmente a los rebecos. Una vez comienza a extenderse, las poblaciones de rebecos comienzan a disminuir, dejando a los animales que sobreviven muy débiles. Para evitar estas graves consecuencias, se requiere de una herramienta que consiga detectar la enfermedad a tiempo y, así, poder comenzar con el tratamiento de los animales enfermos.

Considerando que el diagnóstico de esta enfermedad se realiza principalmente observando la piel y el pelaje de los rebecos en busca de lesiones cutáneas y la naturaleza esquiva de estos animales, la técnica más apropiada para su detección es el desarrollo de una red neuronal convolucional.

En este trabajo se han creado y entrenado múltiples redes neuronales para cumplir este propósito. Se ha seleccionado y optimizado la más eficiente y se han conseguido unos resultados excelentes. El modelo desarrollado tiene una gran capacidad para generalizar a partir de los datos de entrenamiento y, por lo tanto, realiza predicciones muy precisas y fiables. Esto sugiere que el modelo obtendría unos resultados prometedores una vez se aplique a un entorno real.

Por otro lado, gracias a este proyecto, se han podido poner en práctica los conocimientos y las aptitudes adquiridas en la carrera y el máster. Asimismo, se han conseguido desarrollar nuevas habilidades referidas al campo de la inteligencia artificial que supondrán una ventaja en mi futuro profesional.

Además, como se comentará en el siguiente y último apartado, los resultados obtenidos con este trabajo abren las puertas a diversas líneas de investigación futuras, entre las cuales destaca su adaptación al entorno natural.

Por último, es importante destacar que el desarrollo de un trabajo como este supone una contribución al campo de la salud, tanto animal como humana, y a la protección del medio ambiente y la biodiversidad. Alineado con los Objetivos de Desarrollo Sostenible (ODS) establecidos por las Naciones Unidas, este proyecto busca promover la salud y el bienestar, preservar la biodiversidad

y fomentar prácticas sostenibles en la protección de los animales y su entorno.

5.2. Líneas Futuras

Una vez logrado nuestro objetivo en este proyecto, es importante considerar las posibles líneas futuras en las cuales aplicar lo aprendido:

- **Adaptación al entorno natural:** La finalidad principal de este trabajo es poder crear una red neuronal que nos permitiese detectar sarna en rebecos para que, en un futuro, se pudiera aplicar en su hábitat natural. Por lo tanto, el siguiente paso consiste en establecer las condiciones necesarias para poder implementar el modelo en este contexto. Para conseguirlo, será necesario contar con cámaras de alta calidad y alta resolución que sean capaces de capturar imágenes o vídeos de dichos animales. Es importante que la cámara sea capaz de capturar con alto nivel de detalle el pelo y piel de los rebecos, asegurando así una detección altamente confiable. Las cámaras tendrán que situarse en lugares estratégicos donde es común encontrar estos animales, como bosques o montañas, para que resulten efectivas. Por otro lado, se necesitará un servidor o sistema informático capaz de procesar y analizar las imágenes o vídeos que capture la cámara. Este servidor debe tener alta capacidad de almacenamiento y de procesamiento para poder ejecutar el modelo de red neuronal a tiempo real. Además, necesitaremos una fuente de energía sostenible y una conexión a red estable para transmitir las imágenes o vídeos de la cámara al servidor y para que el servidor sea capaz de procesar los datos.
- **Sistema de notificación:** Una vez detectado un caso positivo de sarna es necesario generar una alarma o notificación dirigida a las personas responsables. Como medio de comunicación se pueden emplear mensajes de texto, correos electrónicos o incluso se podría desarrollar una aplicación móvil para este propósito. Para enviar estas notificaciones en tiempo real, es necesario establecer una conexión entre el sistema de notificación y el servidor. Si se optase por desarrollar una aplicación móvil, se podrían agregar otras funcionalidades, como la visualización en tiempo real de los casos detectados.
- **Dispensador de pienso automático:** Como habíamos comentado anteriormente, la mejor manera para conseguir tratar a los rebecos enfermos es por vía oral. Esto se suele hacer mediante el uso de pienso medicado que este a disposición de los animales. Por lo tanto, se podrían diseñar unos dispensadores automáticos que suministrasen pienso medicado cada vez que se registrase una alerta por un caso positivo de sarna.
- **Mapeo de casos y seguimiento geográfico:** Otra opción interesante sería mapear los casos de sarna detectados y visualizar su distribución geográfica. De esta manera, se conseguiría tener un mayor seguimiento de la enfermedad. Para ello, se podría utilizar la aplicación que acabamos de comentar, en la cual se mostraría un mapa con los casos registrados. Mediante este mapa, sería posible identificar de manera visual las áreas afectadas y utilizar esta información para tomar decisiones de tratamiento específicas. Además, esta nueva perspectiva podría desencadenar la implantación de nuevas estrategias preventivas basadas en los datos recopilados.
- **Adaptación del modelo para otros animales:** En este trabajo nos hemos centrado en la detección de sarna en rebecos, pero esta enfermedad afecta a muchas otras especies salvajes,

como las cabras montesas o los corzos. El modelo desarrollado ha aprendido a identificar patrones de la enfermedad en rebecos y, por lo tanto, se podría adaptar a otros animales. Esto requeriría ajustar y entrenar la red neuronal para que pueda reconocer las características específicas de la sarna en cada especie animal. Para ello, necesitaríamos recopilar imágenes de estos animales, tanto sanos como enfermos, y realizar las modificaciones y pruebas necesarias para que el modelo sea sea capaz de identificar y clasificar esta enfermedad en todas las especies afectadas. De esta manera, podremos contribuir a la identificación temprana y el tratamiento adecuado de la sarna en todos estos animales, generando un impacto más significativo en la salud y el bienestar de la fauna.

- **Adaptación a otras enfermedades:** El modelo creado se podría adaptar para detectar otras enfermedades cutáneas que afectan a animales silvestres, como tiña, dermatitis o tumores cutáneos. Además, también se podría utilizar para detectar heridas por accidente, lo que permitiría proporcionar atención médica al animal herido, evitándose así complicaciones o sufrimiento innecesario.

Bibliografía

[1] *Objetivos y metas de desarrollo sostenible - Desarrollo Sostenible*. https://www.un.org/sustainabledevelopment/es/objetivos-de-desarrollo-sostenible/. (Accedido el 06/16/2023).

[2] Jesus M Perez et al. "PREVENCION y CONTROL DE LA SARCOPTIDOSIS EN LA FAUNA SILVESTRE". En: (1997). (Accedido el 03/25/2023).

[3] *Frontiers | Sarcoptic Mange in Wild Caprinae of the Alps: Could Pathology Help in Filling the Gaps in Knowledge?* https://www.frontiersin.org/articles/10.3389/fvets.2020.00193/full. (Accedido el 03/24/2023).

[4] *A review of sarcoptic mange in North American wildlife - ScienceDirect*. https://www.sciencedirect.com/science/article/pii/S2213224419300902. (Accedido el 03/24/2023).

[5] *La sarna sarcóptica en el ganado ovino | PortalVeterinaria*. https://www.portalveterinaria.com/rumiantes/articulos/11901/la-sarna-sarcoptica-en-el-ganado-ovino.html. (Accedido el 03/24/2023).

[6] Francisco Javier Pérez-Barbería. *El-rebeco-cantabrico-Rupicapra-pyrenaica-parva-Conservacion-y-gestion-de-sus-poblaciones.pdf*. https://www.researchgate.net/profile/Francisco-Perez-Barberia/publication/274389113_El_rebeco_cantabrico_Rupicapra_pyrenaica_parva_Conservacion_y_gestion_de_sus_poblaciones/links/5f476d5ba6fdcc14c5cc7ddd/El-rebeco-cantabrico-Rupicapra-pyrenaica-parva-Conservacion-y-gestion-de-sus-poblaciones.pdf#page=121. (Accedido el 03/22/2023).

[7] Gloria González et al. *III INCIDENCIA DE LA SARNA SARCÓPTICA ESTADO SANITARIO*. https://www.miteco.gob.es/es/parques-nacionales-oapn/publicaciones/edit_libro_06_04_tcm30-100301.pdf. (Accedido el 03/21/2023).

[8] *Sarna sarcóptica en fauna silvestre - Sanidad cinegética | Ciencia y Caza: Tu web de caza, investigación y formación cinegética*. https://www.cienciaycaza.org/sanidad-cinegetica/sarna-sarcoptica-en-fauna-silvestre/43. (Accedido el 03/22/2023).

[9] *Sarna sarcóptica en rebeco - Sanidad cinegética | Ciencia y Caza: Tu web de caza, investigación y formación cinegética*. https://www.cienciaycaza.org/sanidad-cinegetica/sarna-sarcoptica-en-rebeco/34. (Accedido el 03/22/2023).

[10] *One Health (una sola salud) o cómo lograr a la vez una salud óptima para las personas, los animales y nuestro planeta - Blog - ISGLOBAL*. https://www.isglobal.org/healthisglobal/-/custom-blog-portlet/one-health-una-sola-salud-o-como-lograr-a-la-vez-una-salud-optima-para-las-personas-los-animales-y-nuestro-planeta/90586/0. (Accedido el 06/15/2023).

[11] *Demystifying Neural Networks, Deep Learning, Machine Learning, and Artificial Intelligence - Stoodnt*. https://stoodnt.com/blog/ann-neural-networks-deep-learning-machine-learning-artificial-intelligence-differences/. (Accedido el 05/19/2023).

[12] *Artificial Intelligence vs. Machine Learning vs. Deep Learning | Built In*. https://builtin.com/artificial-intelligence/ai-vs-machine-learning. (Accedido el 05/19/2023).

[13] *Deep Learning Fundamentals - Classic Edition - deeplizard*. https://deeplizard.com/learn/playlist/PLZbbT5o_s2xq7LwI2y8_QtvuXZedL6tQU. (Accedido el 03/22/2023).

[14] *Redes neuronales desde cero (I) - Introducción - IArtificial.net*. https://www.iartificial.net/redes-neuronales-desde-cero-i-introduccion/. (Accedido el 03/22/2023).

[15] *Función de activación - Redes neuronales - Diego Calvo*. https://www.diegocalvo.es/funcion-de-activacion-redes-neuronales/. (Accedido el 03/22/2023).

[16] *Hyperparameters in Machine Learning - Javatpoint*. https://www.javatpoint.com/hyperparameters-in-machine-learning. (Accedido el 03/29/2023).

[17] *Machine Learning for Beginners: An Introduction to Neural Networks - victorzhou.com*. https://victorzhou.com/blog/intro-to-neural-networks/#now-what. (Accedido el 04/06/2023).

[18] *Optimizers in Deep Learning: A Comprehensive Guide*. https://www.analyticsvidhya.com/blog/2021/10/a-comprehensive-guide-on-deep-learning-optimizers/. (Accedido el 03/27/2023).

[19] *5 Techniques to Prevent Overfitting in Neural Networks - KDnuggets*. https://www.kdnuggets.com/2019/12/5-techniques-prevent-overfitting-neural-networks.html. (Accedido el 05/07/2023).

[20] *7 Best Techniques To Improve The Accuracy of CNN W/O Overfitting | by Hargurjeet | MLearning.ai | Medium*. https://medium.com/mlearning-ai/7-best-techniques-to-improve-the-accuracy-of-cnn-w-o-overfitting-6db06467182f. (Accedido el 05/07/2023).

[21] *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way | Saturn Cloud Blog*. https://saturncloud.io/blog/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way/. (Accedido el 03/24/2023).

[22] Rikiya Yamashita et al. "Convolutional neural networks: an overview and application in radiology". En: (ago. de 2018). (Accedido el 04/10/2023), págs. 611-629. URL: https://insightsimaging.springeropen.com/articles/10.1007/s13244-018-0639-9.

[23] *Transfer Learning en modelos profundos - Think Big Empresas*. https://empresas.blogthinkbig.com/transfer-learning-en-modelos-profundos/. (Accedido el 04/20/2023).

[24] *What Is Transfer Learning? A Guide for Deep Learning | Built In*. https://builtin.com/data-science/transfer-learning. (Accedido el 04/20/2023).

[25] *Saque la creatividad que lleva dentro con fotografías y vídeos de Alamy*. https://www.alamy.es/. (Accedido el 04/01/2023).

[26] *Fotos de stock, ilustraciones, vectores y clips de vídeo libres de derechos - Getty Images*. https://www.gettyimages.es/. (Accedido el 04/01/2023).

[27] *Más de 1 millón de Imágenes Gratis para Descargar - Pixabay - Pixabay*. https://pixabay.com/es/. (Accedido el 04/01/2023).

[28] *Colaboratory*. https://colab.research.google.com/. (Accedido el 03/19/2023).

[29] *What is Google Colab?* https://educationecosystem.com/blog/what-is-google-colab/. (Accedido el 05/28/2023).

[30] *Google Drive*. https://drive.google.com/. (Accedido el 03/19/2023).

[31] *Google Drive: Almacenamiento Online | Google Workspace*. https://workspace.google.com/intl/es/products/drive/. (Accedido el 03/19/2023).

[32] *Welcome to Python.org*. https://www.python.org/. (Accedido el 04/29/2023).

[33] *¿Qué es Python? - Explicación del lenguaje Python - AWS*. https://aws.amazon.com/es/what-is/python/. (Accedido el 04/29/2023).

[34] *¿Qué es TensorFlow y para qué sirve?* https://www.incentro.com/es-ES/blog/que-es-tensorflow. (Accedido el 03/16/2023).

[35] *TensorFlow*. https://www.tensorflow.org. (Accedido el 03/16/2023).

[36] *Keras: Deep Learning for humans*. https://keras.io/. (Accedido el 03/16/2023).

[37] *KerasTuner*. https://keras.io/keras_tuner/. (Accedido el 05/15/2023).

[38] *sklearn.model_selection.GridSearchCV — scikit-learn 1.2.2 documentation*. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html. (Accedido el 05/09/2023).

[39] *What Is Grid Search?. Explaining How To Obtain Optimal... | by Farhad Malik | FinTechExplained | Medium*. https://medium.com/fintechexplained/what-is-grid-search-c01fe886ef0a. (Accedido el 05/09/2023).

[40] *DenseNet*. https://keras.io/api/applications/densenet/. (Accedido el 04/15/2023).

[41] *Architecture of DenseNet-121*. https://iq.opengenus.org/architecture-of-densenet121/. (Accedido el 04/15/2023).

[42] *VGG16 and VGG19*. https://keras.io/api/applications/vgg/. (Accedido el 04/15/2023).

[43] *Arquitectura VGG16 y VGG19 en Deep Learning*. https://keepcoding.io/blog/arquitectura-vgg16-vgg19-deep-learning/. (Accedido el 04/15/2023).

[44] *MobileNet, MobileNetV2, and MobileNetV3*. https://keras.io/api/applications/mobilenet/. (Accedido el 04/20/2023).

[45] *Review: MobileNetV2 — Light Weight Model (Image Classification) | by Sik-Ho Tsang | Towards Data Science*. https://towardsdatascience.com/review-mobilenetv2-light-weight-model-image-classification-8febb490e61c. (Accedido el 04/20/2023).

[46] *tf.keras.applications.resnet50.ResNet50 | TensorFlow v2.12.0*. https://www.tensorflow.org/api_docs/python/tf/keras/applications/resnet50/ResNet50. (Accedido el 04/30/2023).

[47] *Understanding ResNet50 architecture*. https://iq.opengenus.org/resnet50-architecture/. (Accedido el 04/30/2023).

[48] *Performance Metrics: Confusion matrix, Precision, Recall, and F1 Score | by Vaibhav Jayaswal | Towards Data Science*. https://towardsdatascience.com/performance-metrics-confusion-matrix-precision-recall-and-f1-score-a8fe076a2262. (Accedido el 05/12/2023).