Escuela Técnica Superior de Ingeniería Informática

Ingeniería de la Salud

TRABAJO FIN DE GRADO

Detección de cáncer de piel en imágenes dermatoscópicas usando técnicas de Deep Learning.

Autora:

Ana Caamaño Cundíns

Tutor:

Juan Antonio Nepomuceno Chamorro

Primera convocatoria

Curso: 2019/2020

Resumen

El objetivo de este proyecto es realizar un estudio en *deep learning* y crear una herramienta capaz de detectar cáncer de piel en imágenes dermatoscópicas basada en redes neuronales convolucionales. Proyecto que nace para dar una posible solución a un reto de kaggle a nivel internacional.

Se empezará por indagar en el estudio e investigación de esta temática y estado del arte, donde se explicarán los conceptos fundamentales necesarios para el desarrollo del trabajo y se profundizará en las redes neuronales convolucionales, las grandes protagonistas de este trabajo.

Para poder empezar a desarrollar la idea planteada, se hablará de los datos proporcionados por kaggle, así como de las aportaciones más relevantes de otros participantes del reto que fueron una ayuda clave para iniciar este trabajo. Además, se hará una introducción a los cursos y tutoriales hechos y se expondrán las tecnologías estrella del proceso: keras, tensorflow y Google Colab.

A continuación, se presentará el estudio realizado formado por el diseño de tres modelos diferentes, tres posibles soluciones al reto planteado con distinta arquitectura y técnicas aplicadas, donde se verá cómo al aumentar la complejidad de la red creada y al aplicar distintas técnicas, el efecto en la precisión de cada modelo variará. Se trabará tanto con redes entrenadas desde cero, una sencilla y otra más compleja con aumento de datos, como con una red pre entrenada (VGG16) a la que se la aplicarán las técnicas de extracción de características y ajuste fino.

El siguiente paso será mostrar las distintas pruebas realizadas sobre los tres modelos, indicando las métricas utilizadas, los distintos valores de parámetros y las distintas divisiones del *dataset* de entrada probadas. Una vez conseguidos los mejores resultados de cada diseño, se hará una serie de pruebas definitivas para obtener la mejor versión de cada uno de ellos y se elegirá la mejor solución entre ellas.

Por último, se plasmarán las conclusiones generales del trabajo realizado, tanto los resultados obtenidos como las dificultades superadas durante el desarrollo, y también se propondrán posibles mejoras del proyecto hecho para obtener mejores resultados en un futuro.

Palabras Clave

Deep learning, redes neuronales convolucionales, lesión cutánea, cáncer de piel, keras, tensorflow, entrenamiento de redes neuronales, *transfer learning*.

Abstract

The objective of this project is to create a Deep Learning tool capable of detecting skin cancer in dermoscopic images based on convolutional neural networks. The project born to give a possible solution to a Kaggle challenge at an international level.

It will begin by investigating the study of this theme and state of the art, where the fundamental concepts necessary for the development of the work will be explained and the convolutional neural networks will be explored, the great protagonists of this project.

In order to start developing the proposed idea, we will talk about the data provided by kaggle and the most relevant contributions from other challenge participants who were a key help to start this work. There will be an introduction to the courses and tutorials made and the star technologies of the process will be exposed: keras, tensorflow and GoogleColab.

Then, the study carried out consisting of the design of three different models will be presented, three possible solutions to the challenge posed with different architecture and applied techniques, where you will see how by increasing the complexity of the created network and applying different techniques, the effect on the precision of each model will vary. It will work with networks trained from scratch, one simple and one more complex with data augmentation, and a pre-trained network to which the fine-tuning and feature extraction techniques will be applied.

The next step will be to show the different tests carried out on the three models, indicating the metrics used, the different parameter values and the different divisions of the dataset tested. Once the best results of each design have been achieved, there will be a set of definitive tests to obtain the best version of each one of them and the best solution among them will be chosen.

Finally, the general conclusions of the work carried out will be reflected, the results obtained and the adversities overcome during development, and also possible improvements of the project made to obtain better results in the future will be proposed.

Keywords

Deep Learning, convolutional neural networks, skin lesion, skin cancer, keras, tensorflow, neural network training, transfer learning.

Agradecimientos

En primer lugar, quiero dar las gracias a mi amiga y compañera Marta, ya que su entusiasmo y poder de convicción me empujaron a elegir este trabajo similar al suyo y consiguió que las horas, días y meses de estudio conjunto se pasasen volando. Fuiste calma y paz en los momentos de tormenta. Gracias, amiga.

Luego están ellos, los que protagonizaron la mejor etapa de mi vida, la pequeña familia que esta carrera unió y logró que las horas de estudio fuesen más divertidas que los descansos en un bar, si cabe. Me completasteis como persona, me enseñasteis a ver la vida desde diferentes perspectivas y que la diversidad es realmente maravillosa. Sois ángeles y demonios a partes iguales. Os querré siempre.

Gracias a mis amigos, a los de siempre, a los que una distancia de 800 km no pudo separar, a los que pude ver días contados durante 4 años y pese a eso, consiguieron que nada cambiase. Gracias por darme vida cuando el mundo me agobiaba, por darme empujoncitos de esperanza y felicidad pura cuando más lo necesitaba, por ser cielo cuando huía del infierno.

A mi familia, mi equipo, el que hizo posible todo esto trabajando desde detrás del telón. Gracias por regalarme la mejor oportunidad de mi vida sin poner pegas, por luchar por mis sueños más que yo misma, por apoyarme cuando me derrumbaba y por no presionarme en ningún momento. Supisteis hacerlo mejor que yo, por eso siempre estaré inmensamente orgullosa de vosotros. Os lo debo todo.

Y finalmente, a mi tutor, por hacer esto realidad.

Material adicional

A continuación, se adjunta un enlace correspondiente al repositorio de GitHub donde se podrá encontrar la memoria del proyecto, el *notebook* de trabajo del estudio realizado y, además, un conjunto de *notebooks* y apuntes utilizados para el aprendizaje y el estudio previo:

https://github.com/anacaacun/Deteccion-de-cancer-de-piel-en-imagenes-dermatoscopicas-usando-tecnicas-de-Deep-Learning

El código QR asociado a este enlace:

Índice General

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1. Motivación

En la actualidad, la Inteligencia Artificial, el *deep learning* y sus diferentes aplicaciones tecnológicas en el mundo de la medicina están en auge. Décadas y décadas de desarrollo permitieron que algo tan inimaginable como que unas simples líneas de código pudiesen predecir si un cáncer es benigno o maligno, fuera posible. Estos algoritmos que enseñan a las máquinas a pensar como las personas permiten la creación de avances tecnológicos que facilitan el trabajo del personal sanitario como, por ejemplo, manejar grandes cantidades de historiales médicos u ofrecer diagnósticos automatizados. Por tanto, la motivación de este trabajo es el simple hecho de que, aunque una máquina nunca podrá sustituir a un médico, sí puede ser su aliado y facilitar mucho el trabajo de los profesionales y expertos en el sector de la sanidad. En concreto, este estudio trata la detección de distintas clases de lesiones cutáneas en imágenes dermatoscópicas, usando una de las más famosas aplicaciones de *deep learning*, las redes neuronales convolucionales.

El reto de este trabajo es desarrollar una herramienta que consiga clasificar los distintos tipos de cáncer de piel de forma automática a través del tratamiento de imágenes dermatoscópicas. Se evaluará la calidad de predicción de esta herramienta con el conjunto de imágenes de cáncer de piel proporcionadas (Figura 1.1), imágenes con las cuales se entrenará a los distintos modelos, además de probar distintos valores de parámetros y diferentes arquitecturas en las redes neuronales diseñadas durante el desarrollo.

Este proyecto está enfocado en una competición de kaggle (kaggle, s.f.) a nivel internacional. El reto facilita el conjunto de datos de entrada necesarios para poder llevar a cabo el desarrollo de esta herramienta, además de la posible visualización de otros estudios realizados por más participantes de la competición que permitirán obtener una idea inicial de las técnicas y redes que habrá que emplear.

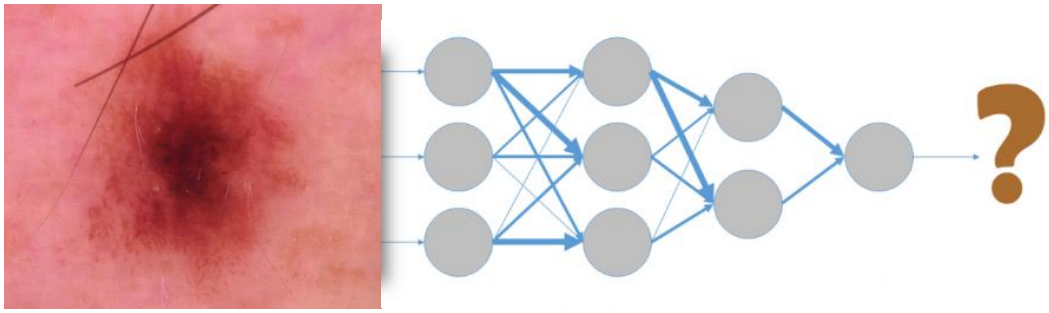


Figura 1.1. Esquema de un clasificador de imágenes en deep learning basado en redes neuronales.

1.2. Objetivos

El objetivo de este trabajo es la creación de una herramienta capaz dar solución al problema planteado por la competición de kaggle, a través del previo estudio de esta temática de *deep learning*. Primero, se hará una investigación de los conceptos fundamentales de las redes neuronales y técnicas aplicadas en ellas, además de la observación de las aportaciones de los demás participantes del reto, con la finalidad de poder construir un modelo inicial con el que realizar los distintos entrenamientos y pruebas. No solo se quiso formar y entrenar un modelo, sino que se consideró interesante construir tres donde se pudiese ver el distinto efecto provocado de las diferentes arquitecturas de red y de las técnicas aplicadas sobre la calidad de precisión de la herramienta. Para cada uno de los tres modelos se harán una serie de pruebas con distintos valores de parámetros y distinta división del conjunto de datos de entrada, con el fin de obtener la combinación óptima con la que se obtengan los mejores resultados. Se observarán los resultados obtenidos y se elegirá la mejor solución para el problema de predicción de cáncer de piel planteado.

Este objetivo general se puede particionar en varios objetivos más concretos que más adelante se dividirán en distintas tareas necesarias para conseguir cada uno de ellos y el tiempo requerido para completarlos. Los objetivos en cuestión son los siguientes:

1. Introducirse en la temática y realizar el estudio del estado del arte: para lograr este objetivo se empezará por hacer una investigación de *deep learning* y de las redes neuronales, además de los conceptos fundamentales de este tema y se profundizará en las redes neuronales convolucionales.
2. Investigación de los *notebooks* proporcionados por los participantes del reto: las soluciones propuestas por los demás competidores servirán de ayuda para poder empezar con el desarrollo de la herramienta.

3. Diseño de modelos: tras la investigación hecha, se construirán 3 modelos distintos con distintas arquitecturas y técnicas aplicadas, partiendo de uno simple (una red entrenada desde cero sencilla) y aumentar en complejidad (pasando por una red entrenada desde cero más compleja con aumento de datos y una red preentrenada) donde probaremos distintos valores de hiperparámetros.
4. Experimentos: se observarán los resultados obtenidos en las distintas pruebas realizadas sobre los tres modelos y se elegirá la herramienta con más porcentaje de precisión y más adecuada para dar solución a nuestro problema.
5. Conclusiones y mejoras: se expondrá un resumen de todo el proceso de trabajo de este proyecto, los resultados conseguidos y las adversidades superadas. Además, se propondrán posibles mejoras en el trabajo realizado para poder obtener resultados de mayor calidad.
6. Redactar el estudio hecho: plasmar por escrito todo lo aprendido y conseguido en este trabajo.

A continuación, se dividirá cada objetivo descrito en una serie de tareas, especificando las fechas en las cuales se realizó cada una de ellas:

OBJETIVO	**TAREA**	**FECHA INICIO**	**FECHA FIN**
Temática y estado del arte		**02/03/2020**	**30/03/2020**
	Investigación en *deep learning*	02/03/2020	06/03/2020
	Investigación de redes neuronales	09/03/2020	13/03/2020
	Aprendizaje de conceptos básicos	09/03/2020	13/03/2020
	Investigación de redes neuronales convolucionales	16/03/2020	20/03/2020
	Aprendizaje de keras y Tensorflow	23/03/2020	30/03/2020
Reto kaggle		**01/04/2020**	**10/04/2020**
	Investigación de los *notebooks* de otros participantes	01/04/2020	06/04/2020
	Preprocesamiento de los datos de entrada proporcionados por el reto	01/04/2020	10/04/2020
Diseño		**10/04/202**	**30/04/2020**
	Búsqueda de un entorno de desarrollo con GPU en la nube	10/04/2020	17/04/2020
	Primeros modelos y simulaciones	24/04/2020	24/04/2020
	Cambios y nuevas simulaciones	24/04/2020	30/04/2020

	Modelos definitivos	30/04/2020	30/04/2020
Experimentos		**01/05/2020**	**22/05/2020**
	Pruebas realizadas sobre el modelo 1	01/05/2020	08/05/2020
	Pruebas realizadas sobre el modelo 2	11/05/2020	15/05/2020
	Pruebas realizadas sobre el modelo 3	18/05/2020	20/05/2020
	Comparar los resultados obtenidos	21/05/2020	22/05/2020
Conclusiones y mejoras		**25/05/2020**	**25/05/2020**
	Analizar los resultados obtenidos y proponer mejoras	25/05/2020	25/05/2020
Redacción de la memoria del proyecto		**15/05/2020**	**09/06/2020**
	Redacción de la memoria del proyecto	15/05/2020	05/06/2020
	Correcciones de la memoria final	08/06/2020	08/06/2020
	Correcciones sobre el *notebook* de trabajo final	08/06/2020	08/06/2020
	Elaboración de un repositorio GitHub y un código QR asociado	05/06/2020	05/06/2020
	Último repaso de la entrega final	08/06/2020	09/06/2020

Tabla 1.1. Tareas realizadas por cada objetivo y fechas en las que se completó cada una de ellas.

1.3. Planificación temporal y estudio de costes.

- **Planificación temporal**

En este apartado se especificará el proceso temporal empleado en cada una de las tareas vistas en el punto anterior (Tabla 1.1.) observando gráficamente estos periodos de tiempo mediante el siguiente diagrama de Gantt:

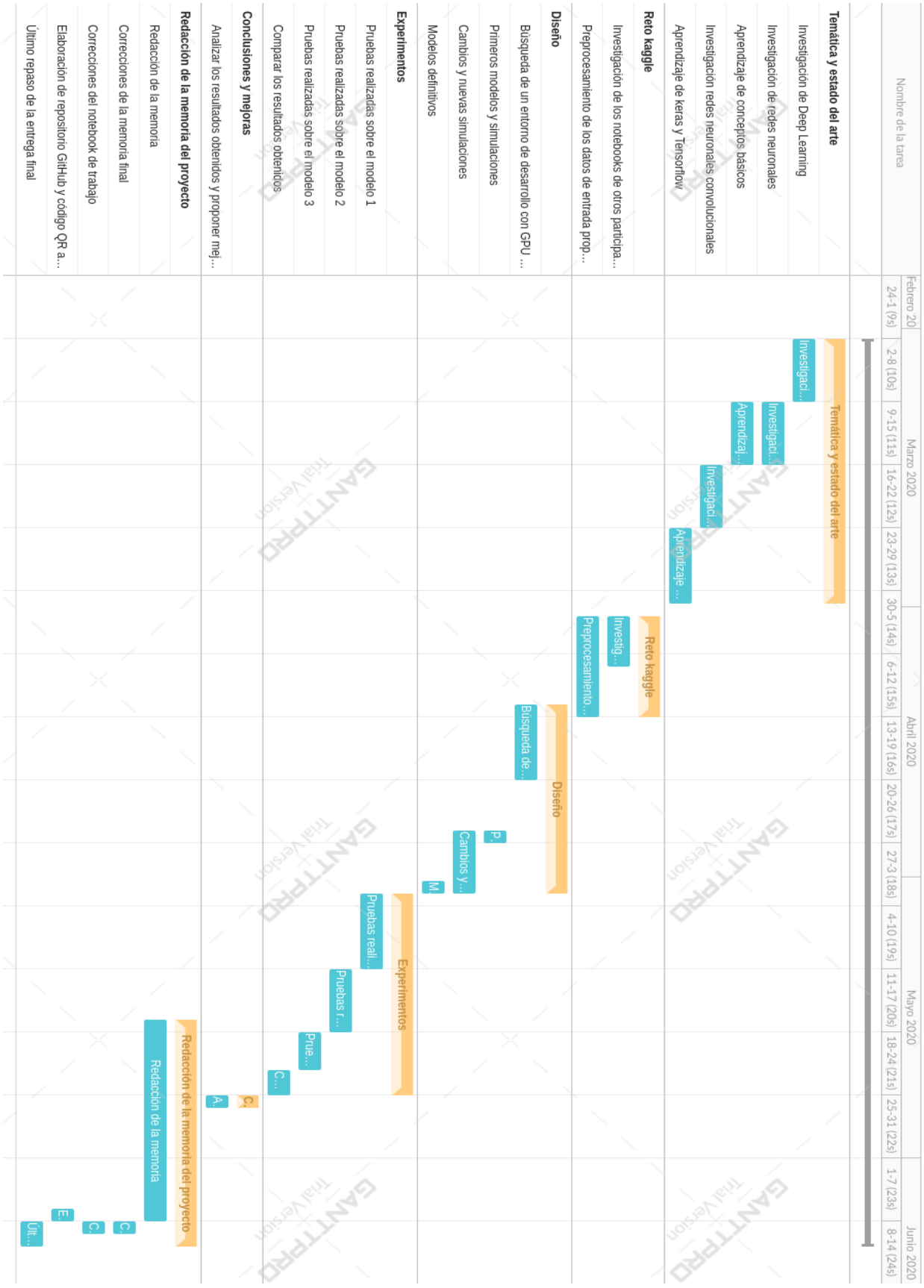


Figura 1.2. Diagrama de Gantt. Planificación temporal del proyecto (Marzo – Junio).

- **Estudio de costes**

A continuación, se detallará el coste de este proyecto en caso de que se hubiese sido desarrollado profesionalmente. Tengamos en cuenta, por tanto, lo siguiente:

- Este proyecto es realizado por una sola persona cuyo nivel profesional equivale a ingeniero junior.
- El proceso temporal de desarrollo de este trabajo abarca unos 4 meses, aproximadamente, desde el 1 de marzo hasta el 10 de junio. Considerando que se siguió una jornada de trabajo similar a una jornada de oficina, trabajando de lunes a viernes con un horario de 09:00h-13:00h y 18:00-21:00h, se puede hacer el cálculo de las horas totales de trabajo realizado.

Empezaremos este estudio de costes por la parte de **recursos humanos**. Como se ha dicho, se pueden calcular las horas de trabajo realizadas aproximadamente. Si consideramos que se ha realizado un total de 73 días laborales y cada uno de los días se trabajó unas 7 horas de media, se estima un total de 511 horas trabajadas en este proyecto. Si consideramos que el coste por hora de un ingeniero junior es de 12€/hora, los gastos en recursos humanos serán de unos **6132€** (12€/h x 511h).

En cuanto a los gastos de **recursos materiales**, tanto software como hardware utilizado, anotaremos el coste del ordenador personal (900€), el entorno de desarrollo Google Colab Pro (20 €), la electricidad consumida (40€) y la conexión a Internet (140€) durante estos 4 meses, lo que hace un coste total de **1100€**

Como conclusión, el **presupuesto** estimado para llevar a cabo este proyecto, sumando los costes de recursos humanos y materiales, significaría una cantidad de siete mil doscientos treinta y dos Euros (**7232€**).

1.4. Estructura de la memoria

La memoria de este trabajo está compuesta por los siguientes capítulos:

- **Capítulo 1**: Introducción al problema, objetivos y organización del TFG.
- **Capítulo 2**: Introducción a la temática del TFG y una breve entrada a las redes neuronales.
- **Capítulo 3**: Se presenta el estado del arte en el que se explican los conceptos más importantes que se usarán a lo largo del proyecto.

- **Capítulo 4**: Se expone el trabajo realizado, los distintos modelos desarrollados y las técnicas aplicadas en cada uno de ellos.
- **Capítulo 5**: Se explican los distintos experimentos realizados y se presentan los resultados de evaluación de dichas pruebas en cada uno de los modelos.
- **Capítulo 6**: Se resumen las conclusiones obtenidas del trabajo en general y se plantean algunas opiniones sobre nuevos propósitos que podrían realizarse en un trabajo futuro.
- **Bibliografía**.

Capítulo 2

Introducción a la temática

2.1. Inteligencia Artificial, Machine Learning y Deep learning

En primer lugar, es necesario definir claramente los conceptos de Inteligencia Artificial, *machine learning* y *deep learning* para ubicarse en la temática del proyecto.

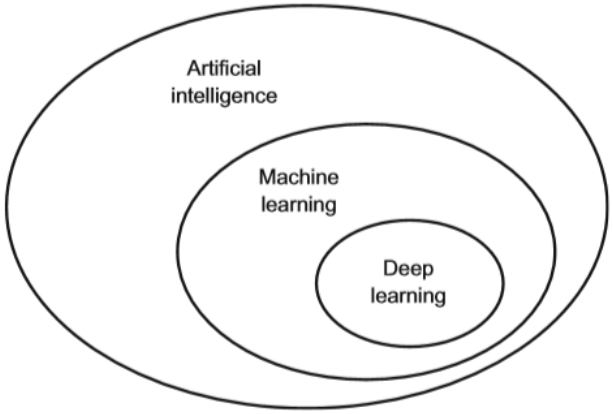


Figura 2.1. Inteligencia Artificial, machine learning y deep learning.

¿Qué es la Inteligencia Artifical?

Por definición, la Inteligencia Artificial (IA) es de "el esfuerzo por automatizar tareas intelectuales que habitualmente son desempeñadas por humanos" y, como podemos ver en la Figura 2.1 (Chollet, 2020), engloba al *machine learning* y al *deep learning*. Con el paso de los años, la IA resultó ser adecuada para problemas lógicos que estaban bien definidos, donde precisar las reglas específicas para manipular este conocimiento era relativamente sencillo, pero demostró ser imposible para determinar estas reglas en problemas que ya eran más complejos como, por ejemplo, la clasificación de imágenes. Surge, entonces, una nueva visión: el *machine learning.*

¿Qué es machine learning?

Este nuevo concepto nace de preguntarnos si los ordenadores o máquinas pueden ir más allá de lo que les ordenamos hacer, es decir, si pueden sorprendernos y aprender por sí mismos. Surge así, un nuevo paradigma de programación: el paso de la programación clásica al "entrenamiento". Veamos la diferencia:

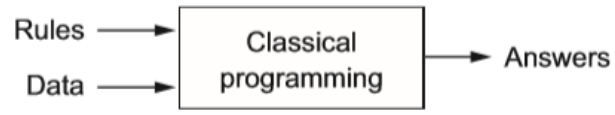


Figura 2.2. Programación clásica.

La programación clásica (Figura 2.2. - (Chollet, 2020)) consiste en generar unas respuestas (*Answers*) a partir de un conjunto de datos (*Data*) y de unas reglas (*Rules*) que introduce el programador.

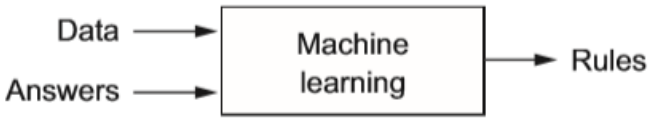


Figura 2.3. machine learning: un nuevo paradigma de programación.

Mientras, el *machine learning* deja atrás esta forma clásica de programar (Figura 2.3. - (Chollet, 2020)) y consigue generar las reglas necesarias a partir del conjunto de datos y las respuestas esperadas para ese conjunto en concreto. Por eso decimos que, en vez de programar, “entrenamos” los datos que tenemos para conseguir el objetivo deseado.

Por otra parte, el *machine learning* se puede dividir en 4 ramas: aprendizaje supervisado, aprendizaje no supervisado, aprendizaje autosupervisado y aprendizaje por refuerzo. Profundizaremos en las dos primeras, veamos la diferencia:

- **Aprendizaje supervisado**.

  Se trata de una modalidad de aprendizaje y de entrenamiento de *machine learning* que se basa en, dado un conjunto de datos, aprender a asignar datos de entrada a objetivos conocidos y suele ser la rama más habitual usada actualmente. Consiste en poder hacer predicciones de forma que un algoritmo determinado combine las distintas respuestas. En este caso, las preguntas son denominadas “características” y las respuestas “etiquetas”. (Zambrano, 2018). Este tipo de aprendizaje se puede dividir principalmente en regresión y clasificación. En este proyecto, nos centraremos en esta última: Clasificación.

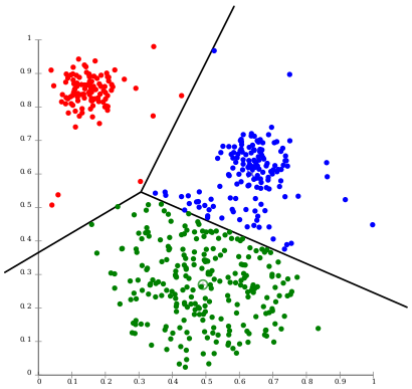


Figura 2.4. Machine learning. Aprendizaje supervisado.Clasificación.

En la clasificación, el algoritmo utilizado encuentra diferentes patrones comunes entre los elementos y su objetivo es clasificarlos en distintas agrupaciones en función de esos patrones y similitudes encontradas (Zambrano, 2018). Por eso, el objetivo es conseguir una relación entre preguntas y respuestas para obtener un resultado aproximado y no calcular de forma precisa a qué grupo pertenece un determinado valor en concreto.

- **Aprendizaje no supervisado:**

  Este tipo se basa en la agrupación y es básico en la analítica de datos. En este caso el algoritmo utilizado, debería catalogar por similitud (Zambrano, 2018). La finalidad del aprendizaje no supervisado es entender mejor las relaciones de los datos que estamos tratando, por eso frecuentemente es un paso previo a resolver un problema de aprendizaje supervisado.

Como conclusión, para aclarar este concepto, el *machine learning* simplemente es una búsqueda de representaciones útiles de datos de entrada

¿Qué es el *deep learning*?

Como sabemos, la IA engloba el *machine learning* y este, al *deep learning* (Figura 2.1 - (Chollet, 2020)), entonces, el *deep learning* simplemente es un subcampo del *machine learning* que consiste en una nueva visión del aprendizaje de representaciones partiendo de datos y acentuando este tipo de aprendizaje en retener capas sucesivas de representaciones que son cada vez más reveladoras y significativas. Por tanto, del "*Deep*" de este concepto hará referencia a la sucesión de capas aprendidas, que marcarán la profundidad de un modelo.

Para poder comprender mejor esta idea de representaciones en capas, vamos a adentrarnos en la teoría de los modelos de *deep learning* conocidos como “redes neuronales”. Aunque este término está relacionado con la neurobiología, los modelos de aprendizaje en *deep learning* no funcionan como lo hace nuestro cerebro, aunque algunos conceptos sí se pudieron plantear al basarse en la forma de entendimiento de este.

A continuación, vamos a ver un ejemplo de una red sencilla formada por cuatro capas sucesivas en la que una imagen de un dígito es transformada en el dígito en cuestión que representa.

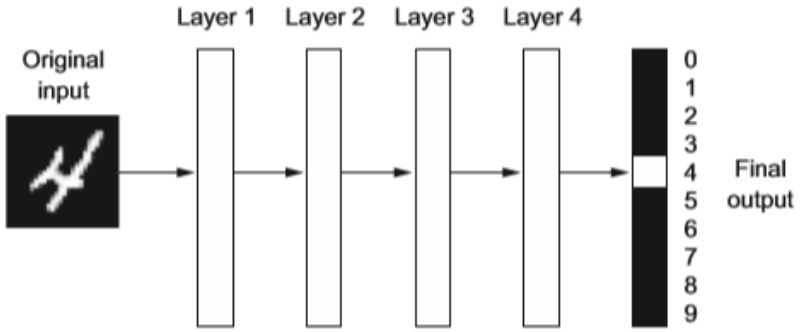


Figura 2.5. Red neuronal profunda. Clasificación de dígitos.

Este es el aspecto que tiene el aprendizaje basado en representaciones llevado a cabo por un algoritmo en *deep learning* (Figura 2.5 - (Chollet, 2020)). Tenemos una imagen de un dígito como entrada que se va transformando en representaciones cada vez más distintas y con más información respecto a lo que queremos saber y obtener como resultado final.

Así definimos el *deep learning*, como un aprendizaje de representaciones en sucesivas capas, como un aprendizaje en diversas fases. La idea es sencilla, pero veamos cómo funciona este mundo.

2.2. Introducción a las redes neuronales

Las primeras ideas básicas sobre redes neuronales aparecieron en los años 50, aunque este enfoque tardó muchos años en arrancar debido a un problema en concreto: entrenar de forma eficaz las redes neuronales grandes. Es entonces, en los años 80, cuando a partir del algoritmo de retropropagación y el descenso de gradiente como optimizador (términos de los que se hablará más adelante) nacen las primeras aplicaciones prácticas de las redes neuronales.

Seguidamente, se hablará de los componentes principales de la anatomía de las redes neuronales: capas, redes, función objetivo y optimizadores.

- Las capas (*layers*) se combinan en un modelo (o red).
- La función de pérdida (*loss function*) define el indicio de retroalimentación que se usa en el aprendizaje de la red.
- Los datos de entrada (*input x*) y sus respectivos objetivos (*true targets*).
- El optimizador (*optimizer*), el cual establece cómo continúa el proceso de aprendizaje.

¿Cómo interaccionan estos elementos?

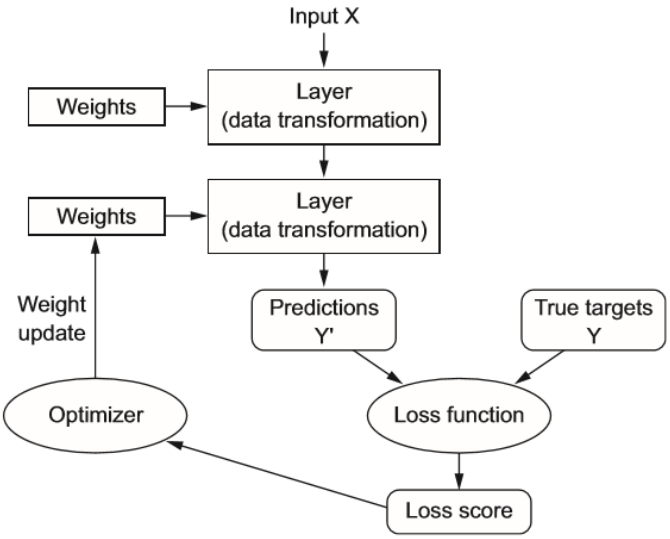


Figura 2.6. Relación entre la red, las capas, la pérdida y el optimizador.

Siguiendo la Figura 2.6. (Chollet, 2020), las distintas conexiones entre los elementos que forman una red neuronal son las siguientes:

- Los datos de entrada (*input x*) son asignados, a través de las distintas capas (*layers*) de la red, a las predicciones (*predictions*).
- Luego actúa la función de pérdida (*loss function*), cuya labor es comparar estas predicciones (*predictions*) con los objetivos (*true targets*) que queremos conseguir y da un valor de pérdida que nos indicará si lo obtenido por la red coincide con lo que esperábamos recibir.
- Por último, el optimizador (*optimizer*) usa este valor que nos proporciona la función de pérdida para poder ajustar la red (*weights update*) de la forma más adecuada para que finalmente, alcancemos la meta deseada.

2.2.1. Capas: bloques de construcción de deep learning

Las capas son el componente principal de las redes neuronales. Se encargan de procesar los datos a través de tensores: reciben como entrada uno o más tensores y devuelven como salida uno o más tensores también. Entramos, entonces, en un nuevo concepto que no habíamos visto hasta ahora y que es imprescindible entender para poder explicar el funcionamiento de las capas y su forma de transformar los datos que se les ingresa.

- Los **tensores** son la estructura de datos básica en los sistemas de *machine learning*. Simplemente son contenedores de datos, por lo general de tipo numérico. Dependiendo de la dimensión que tengan podemos diferenciar tensores 0D (escalares) de dimensión cero, tensores 1D (vectores) de 1 dimensión, tensores 2D (matrices) de 2 dimensiones, tensores 3D de 3 dimensiones y tensores con más dimensiones. El tipo de tensor que nos interesa para realizar este proyecto son los tensores 2D, las matrices Numpy de Python, y los tensores 4D, las imágenes.

Como hemos dicho, las capas trabajan con tensores, por tanto, cada tipo de capa es apropiada para un tipo de tensor en concreto y para un tipo de procesamiento. En este caso, los tensores 2D, que presentan una forma (muestras, características), son procesados por capas densas y los tensores 4D, datos de imágenes, son procesados por capas convoluciones, las grandes protagonistas de este trabajo.

En este punto hay que tener en cuenta la importancia de la “compatibilidad de capas”, es decir, cada capa solo acepta una estructura determinada de tensores y devolverá esa misma.

2.2.2. Modelos: redes de capas

¿A qué nos referimos con un modelo en *deep learning*? Simplemente a un montón de capas apiladas una encima de otras, a través del cual se le asigna una entrada a una salida. Existen distintas topologías de red y uno de los grandes problemas es encontrar la adecuada para un conjunto de datos determinado.

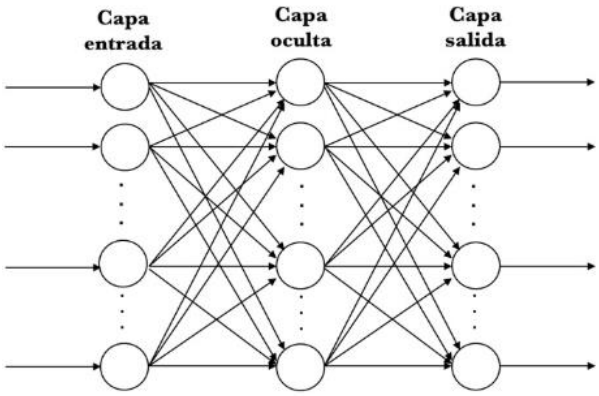


Figura 2.7. Ejemplo de red neuronal de 3 capas.

En esta imagen (Figura 2.7 - (Torres, 2018)), vemos una red neuronal formada por 3 capas en la que la capa de entrada, la cual recibe los datos de entrada, está interconectada con la capa oculta. Esta realiza la transformación de los datos que recibe de las neuronas de la capa anterior (capa de entrada) y los pasa a las de la capa posterior (capa de salida), de forma que devuelva la predicción realizada. Estas transformaciones y asignaciones entre capas son realizadas por los nodos que las forman (neuronas de la red neuronal), y cada uno de ellos tiene sus respectivos pesos (o parámetros) que luego irá ajustando la función de optimización.

Se considera entonces, que la elección de la topología de red adecuada es más un arte que una ciencia y solamente la práctica llevará a ser unos buenos arquitectos de redes neuronales.

2.2.3. Funciones de pérdida y optimizadores

Una vez que hemos elegido la arquitectura de red adecuada, las decisiones clave que la suceden son la elección de la función de pérdida (función objetivo) y del optimizador. Como se dijo anteriormente, la función de pérdida mide el éxito del modelo y el optimizador actualizará la red según el valor de pérdida obtenido.

La buena elección de estas dos cosas es muy importante ya que, de lo contrario, la red podría comportarse de forma errónea y coger el camino más corto para minimizar el valor de pérdida sin relacionarse por completo con el éxito del trabajo que queríamos realizar de forma adecuada. Existen distintas directrices para poder hacer esta elección de forma satisfactoria dependiendo del tipo de problema que se nos plantee.

El truco fundamental del aprendizaje de una red neuronal y del *deep learning* en general es, por tanto, utilizar el valor de pérdida proporcionado por la función objetivo, ver cómo nos alejamos de nuestra finalidad y usar esa información como “retroalimentación” para poder ajustar los pesos del modelo y acercarnos al objetivo deseado reduciendo la pérdida. (Figura 2.6 - (Chollet, 2020)).

Capítulo 3

Estado del Arte

3.1. Conceptos básicos

3.1.1. Conjuntos de entrenamiento, validación y prueba

Para poder empezar con el entrenamiento de un modelo en *deep learning*, hay que dividir el conjunto de datos en tres grupos o conjuntos: entrenamiento (*train*), validación (*validation*) y prueba (*test*). ¿Para qué sirve cada uno de ellos? Para entrenar, evaluar el modelo y para probarlo por última vez, respectivamente.

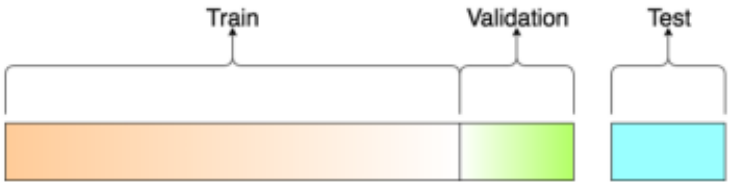


Figura 3.1. Ejemplo de división de conjunto de datos de entrenamiento, validación y test.

El conjunto de entrenamiento se utiliza, como su nombre indica, para entrenar el modelo creado. Por tanto, este conjunto de datos es utilizado para que los valores de los parámetros del modelo sean obtenidos por el algoritmo de aprendizaje usado. Habría que modificar el valor de algún hiperparámetro, en caso de que la red no se acondicionase a los datos de entrada, volver a entrenar el modelo con los datos de entrenamiento y evaluar con el conjunto de validación. Por último, usamos el modelo de prueba, un conjunto de datos nunca visto antes por el modelo, para probar el rendimiento de este.

La forma habitual de hacer la división de estos tres conjuntos es que el grupo de entrenamiento sea mayor que el de validación y prueba (Figura 3.1.).

3.1.2. Ingeniería de características

Es el proceso en el cual se usa nuestro conocimiento sobre la red neuronal y sobre los datos que usamos para que el algoritmo funcione de una forma mejor al poder emplear transformaciones no aprendidas a los datos que metemos como entrada en el modelo.

El desarrollo de la ingeniería de características fue muy importante hasta que surge el *deep learning* y deja atrás la necesidad de este concepto. ¿Por qué? Son ahora las famosas redes neuronales las que permiten sacar de forma automática las características de los datos, aunque siempre habrá que tener presente la ingeniería de características por dos simples razones: nos posibilita resolver problemas con menos datos y usando menos recursos.

3.1.3. Sobreajuste y Subajuste

Para poder entender qué es el sobreajuste y el subajuste debemos tener claro los siguientes términos y la relación entre ellos: optimización y generalización.

- Optimización: como se ha explicado anteriormente, es el proceso ajustar un modelo con los datos de entrenamiento para poder conseguir una mejor precisión.
- Generalización: es el rendimiento del modelo obtenido con los datos de prueba.

Uno de los principales problemas del *machine learning* es la tensión que se crea entre estos dos resultados y aprender a tratarlo es muy importante para poder conseguir un buen aprendizaje. De la relación entre estos dos conceptos nacen los siguientes problemas: el subajuste y el sobreajuste (Figura 3.2. - (EPICALSOFT INSTANCE BLOG, s.f.)):

- Subajuste: se produce cuando la red no ha modelado los datos de entrenamiento ni puede generalizar patrones a nuevos datos. Todavía queda progreso por hacer.

- Sobreajuste: tiene lugar cuando el modelo está empezando a aprender patrones que son característicos de los datos de entrenamiento, pero que son irrelevantes respecto a los nuevos datos. Esto tiene un impacto negativo en facultad de un modelo para generalizar.

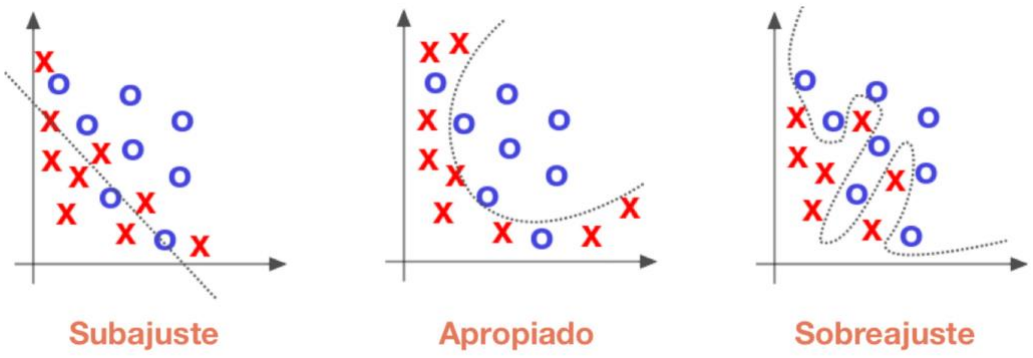


Figura 3.2. Subajuste y Sobreajuste en problemas de clasificación.

El proceso de combatir el sobreajuste es denominado “regularización”. Para evitar este problema, la mejor solución es coger más datos de entrenamiento, siempre que sea posible, y de esta forma el modelo se generalizará mejor. A continuación, explicaremos algunas de las principales técnicas de regularización.

3.1.4. *Dropout*

Es una de las técnicas de regularización más usadas en las redes neuronales y de las más útiles. Consiste en desactivar de forma aleatoria un número de características de salida de la capa durante el entrenamiento. ¿Cómo ayuda esto a acabar con el sobreajuste? Ayuda a reducirlo, ya que las neuronas (características) que se encuentran muy cercanas unas de otras suelen aprender relaciones y patrones muy específicos con los datos de entrenamiento. De esta forma, al aplicar *dropout* se reducen los patrones de casualidad que no son significativos y que la red aprendería en caso de no aplicar esta técnica.

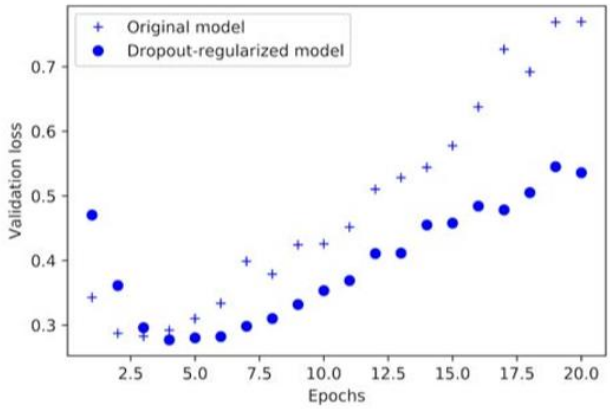


Figura 3.3. Efecto del *dropout* en la pérdida de validación.

En esta imagen (Figura 3.3. - (Chollet, 2020)) podemos ver cómo mejora la pérdida de validación en un modelo al que se le aplica *dropout* frente al original.

3.1.5. Aumento de datos (*Data Augmentation).*

Como se dijo anteriormente, un modelo es incapaz de generalizarse a datos nuevos si se tienen pocas muestras de las que aprender, produciéndose así, el ya mencionado sobreajuste. Una solución a este problema es coger las muestras de entrenamiento que ya tenemos y generar a partir de ellas más datos de entrenamiento. ¿Cómo? Transformando las imágenes originales aleatoriamente, de manera que se generen imágenes que sean totalmente creíbles.

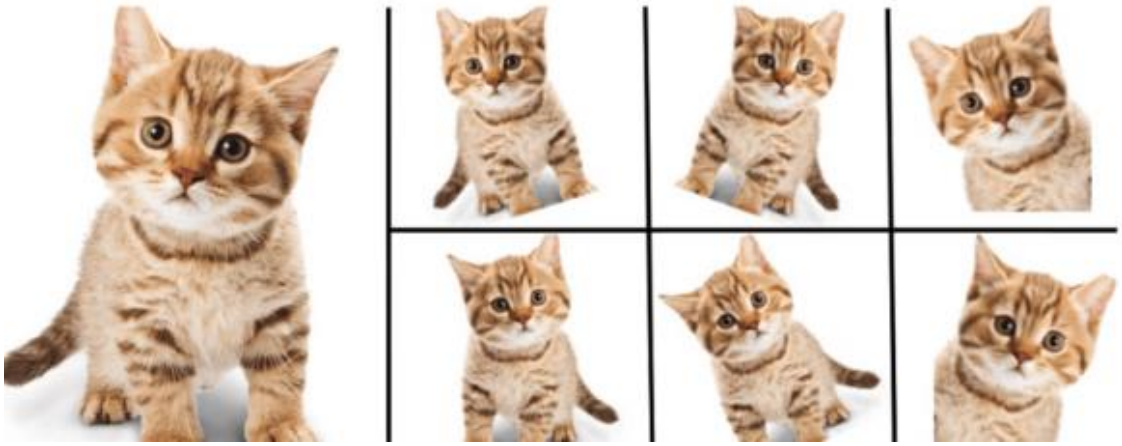


Figura 3.4. Ejemplo de *Data Augmentation*.

Como podemos ver en el ejemplo (Figura 3.4 - (franspg, 2020)), aplicamos a la imagen original distintos tipos de transformaciones como rotación, traslación o zoom, consiguiendo así, aumentar el tamaño y diversidad de nuestros datos. Puede que la aplicación de esta técnica sola no sea suficiente para acabar con el sobreajuste, ya que realmente no es información nueva sino una mezcla de la ya existente, pero si la combinamos con otras como la adición de *dropout*, podremos llegar a reducirlo considerablemente.

3.1.6. Tamaño de *batch* (*batchsize*)

El tamaño de *batch* o *batchsize* es un hiperparámetro aplicado en el entrenamiento del modelo que indica el número de lotes en los que vamos a particionar el conjunto de datos

de entrenamiento durante una iteración del mismo. Esto se aplica porque pasar la totalidad del conjunto de datos por la red neuronal es un proceso lento y caro computacionalmente.

3.1.7. Iteración y época

Son dos hiperparámetros que, en terminología de la red neuronal, se pueden definir de la siguiente forma:

- **Iteración**: se refiere a las veces que un lote de datos (establecido por el *batchsize*) pasa a través de la red neuronal. Este pase es hacia adelante y hacia atrás
- **Época** (o *epoch*): número de veces que la red neuronal ve el conjunto de datos de entrenamiento completo. Cada vez que el algoritmo ve todas las muestras, se completa una época. Una buena táctica de entrenamiento, que veremos más adelante, es aumentar el número de épocas hasta que la precisión de la validación empieza a decrecer (ahí detectamos sobreajuste).

Veamos un ejemplo, para poder aclarar estos dos conceptos y la relación entre ellos:

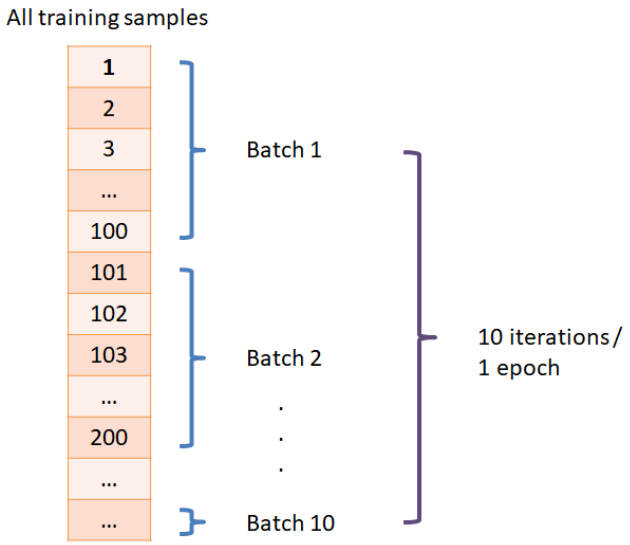


Figura 3.5. Tamaño de *batch*, iteración y época.

Tenemos un conjunto de 1000 muestras y un tamaño de lote de 100 muestras de entrenamiento, y establecemos que queremos que la red ejecute 1 época (N = 100, *batchsize* = 100, *epochs* = 1).

Por tanto, en cada época tenemos 10 lotes (1000/100 = 10). Cada uno de estos lotes, pasa a través de la red neuronal, así que se procesarán 10 iteraciones por Época y, como

solo hemos especificado 1 sola época, después de completar las 10 iteraciones, se habrán entrenado todas las muestras de entrenamiento.

3.1.8. Descenso de gradiente

Para poder explicar este concepto, tenemos que recordar primero qué es una derivada. Si tenemos una función de la forma f(x) = y, siendo x e y números reales, al realizar un pequeño cambio en x, se producirá un pequeño cambio en y. Si representamos este cambio como ε y aproximamos f como función lineal de pendiente a, nos queda lo siguiente:

$$f (x + \varepsilon x) = y + a * \varepsilon y$$

Por tanto, representamos la derivada de f en un punto p de la siguiente forma (Figura 3.6 - (Chollet, 2020)):

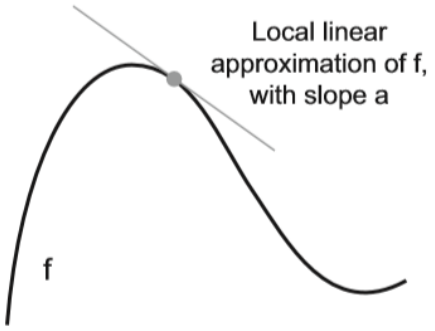


Figura 3.6. Derivada de f en p.

Se considera que el concepto de gradiente es la derivada de una operación con tensores, es decir, de una operación con entradas multidimensionales.

En concreto, hablaremos del descenso de gradiente estocástico. Esta nueva idea hace referencia a encontrar analíticamente el conjunto de valores de peso en una red neuronal que obtengan la función de pérdida más pequeña. Si regresamos al concepto de derivada de una función, simplemente estaríamos hablando de encontrar de entre todos los puntos donde la derivada de la función es 0 (mínimos de la función), el valor más bajo de la función en esos puntos.

Conseguir eso es fácil si seguimos un algoritmo que se conoce como “bucle de entrenamiento” y sigue los siguientes pasos:

1. Dibujar un lote de entrenamiento (x) y sus correspondientes objetivos (y).

2. Ejecutar el modelo sobre el conjunto de entrenamiento para conseguir unas predicciones (y_pred)
3. Calcular la pérdida de la red en este conjunto, la diferencia entre las predicciones (y_pred) obtenidas y los valores reales (y).
4. Computar el gradiente de la pérdida respecto a los pesos de la red (*backpropagation*).
5. Ajustar los pesos en dirección opuesta al gradiente, de forma que obtengamos menos pérdida.

Estos 5 pasos definen el descenso de gradiente estocástico (SGD). A continuación, vamos a ver un ejemplo con una entrada de un tensor 1D, una red con un solo parámetro y una muestra de entrenamiento:

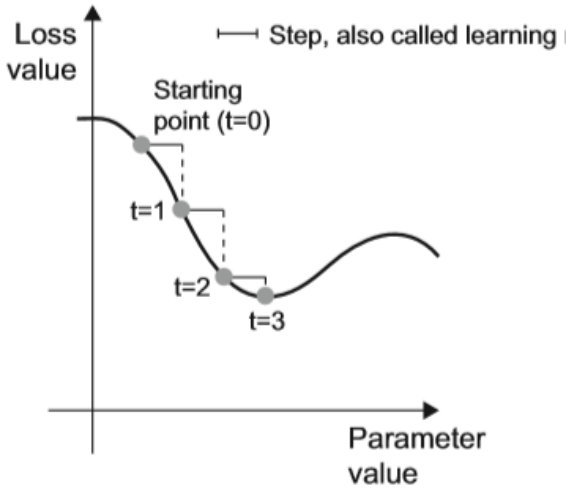


Figura 3.7. Descenso de gradiente estocástico en una curva de pérdida 1D.

Recalcamos que es realmente importante elegir un buen valor del hiperparámetro *learning rate* (del que se hablará un poco más adelante y representado en la Figura 3.7 ((Chollet, 2020)) como "*step*") ya que, si es pequeño, el descenso de gradiente necesitará demasiadas iteraciones y podría atascarse en un mínimo local, mientras que, si es grande, los ajustes podrían acabar en ubicaciones aleatorias de la curva de pérdida.

Existen múltiples variantes del descenso de gradiente estocástico (SGD) y son los conocidos optimizadores o métodos de optimización. La diferencia entre unas variantes u otras simplemente es la forma de ajustar el peso previo a computar el siguiente ajuste de peso. En muchas de ellas, tiene una gran importancia el concepto de "momento", que no es más que ajustar un parámetro a cada paso teniendo en cuenta no solo el gradiente actual, sino también el ajuste del peso anterior. Gracias al momento, se resuelven los dos problemas comentados anteriormente: los mínimos locales y la velocidad de convergencia.

3.1.9 Algoritmo de retropropagación

Hemos dicho que el paso número cuatro del “bucle de entrenamiento” es el algoritmo de retropropagación. Este proceso empieza en el último valor de pérdida y va hacia atrás desde las capas superiores hasta las inferiores, de forma que computa la aportación de cada parámetro al valor de pérdida.

En resumen, una red consta de una serie de capas que aplican una serie de operaciones con tensores a los datos de entrada. Estas operaciones necesitan los valores de los pesos que son los que permiten que la red tenga “conocimiento”. La función de pérdida se utiliza como señal de retroalimentación para aprender los pesos y que el entrenamiento intenta minimizar. El descenso de gradiente estocástico es el que intenta reducir este valor de pérdida y sus reglas de ajuste de pesos estarán definidas por el optimizador usado.

De esta forma la red “aprende” a encontrar una combinación de parámetros (pesos) que minimicen la función de pérdida para el conjunto de entrenamiento dado y sus respectivos objetivos.

3.1.10. Tasa de aprendizaje (*learning rate*)

Como se ha dicho anteriormente, existen múltiples variantes del descenso de gradiente estocástico. Estos algoritmos (optimizadores) multiplican el gradiente por un valor conocido como tasa de aprendizaje (*learning rate*, tamaño de paso o *step*) para precisar el siguiente paso. Por ejemplo, tenemos una gradiente de valor 3 y una tasa de aprendizaje de 0.001 (3 x 0.001 = 0.003), el descenso de gradiente cogerá el punto 0.003 más alejado del anterior.

La tasa de aprendizaje es un hiperparámetro de la red y como se puntualizó anteriormente, es muy difícil elegir un valor adecuado, pero es necesario hacerlo para que el modelo realice un buen aprendizaje, ya que se puede comportar de las siguientes formas:

- Si la tasa de aprendizaje es muy pequeña, el aprendizaje llevará mucho tiempo (Figura 3.8. - (developers.google.com, s.f.)).

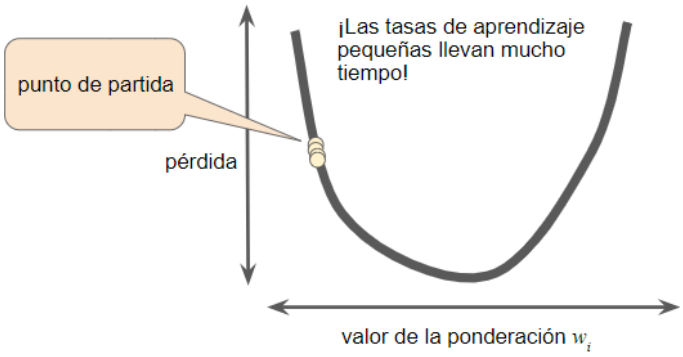


Figura 3.8. Tasa de aprendizaje excesivamente pequeña.

- Si la tasa de aprendizaje es muy grande, el siguiente paso será una ubicación aleatoria de la curva de pérdida (Figura 3.9. - (developers.google.com, s.f.)).

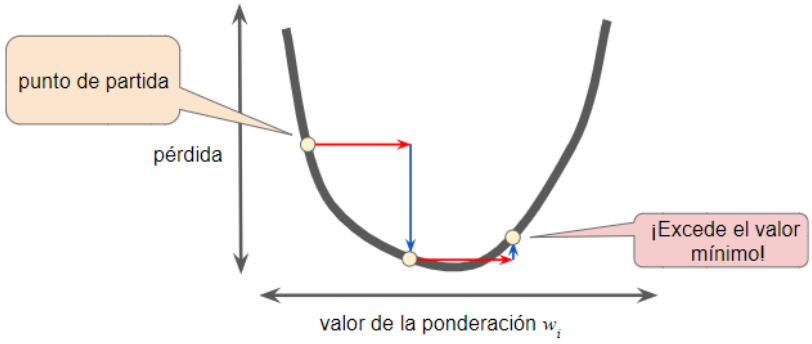


Figura 3.9. Tasa de aprendizaje excesivamente grande.

Elegir un valor de tasa de aprendizaje adecuado (Figura 3.10. - (developers.google.com, s.f.)) significará escoger para un gradiente de función de pérdida pequeño, una tasa de aprendizaje mayor, para compensar el gradiente pequeño (como sabemos, el gradiente se multiplica por el valor del *learning rate*).

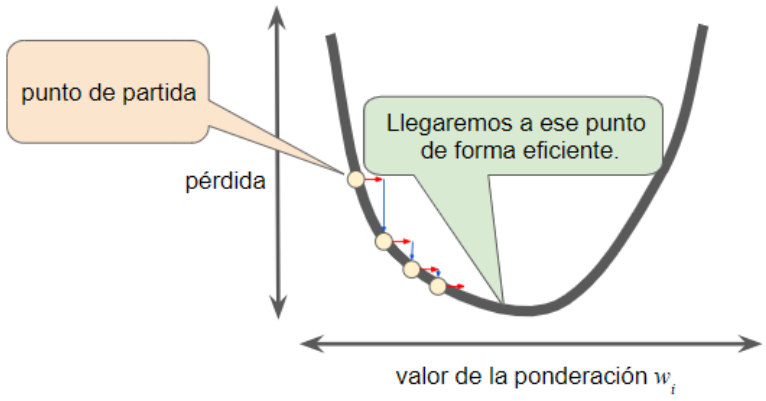


Figura 3.10. Tasa de aprendizaje correcta.

3.1.11. Inicialización de una red neuronal

Llegados a este punto, ya sabemos que las redes neuronales realizan asignaciones de datos de entrada (como imágenes) a objetivos (por ejemplo, la etiqueta “perro”) y que esto es posible gracias a poder observar una gran cantidad de entradas y objetivos y a las transformaciones que se llevan a cabo en las capas.

A continuación, veremos más en concreto cómo tiene lugar este aprendizaje y entraremos en un nuevo concepto que se ha mencionado, pero en el que no se ha profundizado: los pesos.

Los **pesos** son simplemente un montón de números que determinan lo que una capa provoca a los datos de entrada, es decir, la transformación de datos que lleva a cabo cada capa está parametrizada por sus pesos (por eso los pesos también son conocidos como “parámetros”).

Por tanto, para conseguir el objetivo del entrenamiento de la red neuronal, que es hacer correctamente la asignación entrada-objetivo, es necesario encontrar un conjunto de valores adecuados para los pesos de todas las capas que forman la red (Figura 3.11 - (Chollet, 2020)).

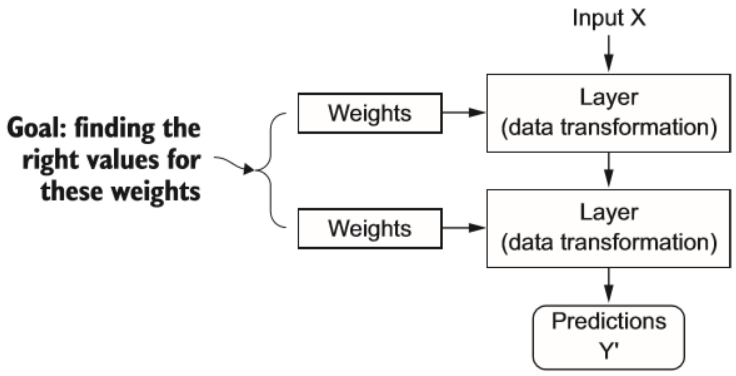


Figura 3.11. Red neuronal parametrizada por sus pesos.

Como una red puede tener millones de pesos, conseguir el valor correcto para todos parámetros es una tarea costosa, pero con la ayuda de la función de pérdida y el optimizador (Apartado 2.2.3), hacen que esto sea un mecanismo “simple” y permiten acercarnos a nuestro objetivo: entrenar la red neuronal.

3.1.12. Ajuste de hiperparámetros

Además de aplicar técnicas de regularización del modelo, como aumento de datos o adición de capas *dropout*, probar con distintos valores de hiperparámetros (*epochs*, *batchsize* y *learning rate*) hará que consigamos o nos acerquemos a una configuración óptima.

El poder ajustar los hiperparámetros de nuestro modelo nos permite redirigir el entrenamiento hacia un lado o hacia otro. Estas variables no están directamente relacionadas con los datos de entrenamiento, pero son variables de configuración, es decir, al terminar el entrenamiento del modelo, adaptamos la precisión global obtenida en busca de una combinación de parámetros inmejorable, dentro de lo posible.

El proceso recomendable para llevar a cabo este ajuste es seguir los siguientes pasos: construir un modelo, elegir el rango de valores para los hiperparámetros que decidimos que vamos a ajustar para obtener el mejor resultado de predicción posible y finalmente, decidir un método que evalúe y permita juzgar al modelo definido. Una vez que probamos todas las combinaciones posibles de hiperparámetros y observamos los resultados obtenidos, ya podremos saber qué valores son los que consiguen la mejor precisión del modelo.

3.2. Redes Neuronales Convolucionales.

Estamos a punto de adentrarnos en la teoría de las redes neuronales convoluciones y el hecho de por qué este mundo está en tan en auge actualmente.

Las Redes Neuronales Convolucionales (también conocidas como "*convnets*") son un tipo de red neuronal artificial que funcionan como el córtex visual del ojo humano, de forma que puede "ver" e identificar objetos a través del procesamiento de sus capas para identificar las diferentes características de entrada. Para poder comprender su funcionamiento hay que saber que existe una jerarquía y a medida que la *convnet* va profundizando en capas va "viendo mejor" o reconociendo formas más complejas, es decir, en las primeras capas detecta formas simples (líneas, curvas…) hasta que en las capas más profundas ya puede detectar formas como rostros o siluetas (Na8, 2018).

La arquitectura de una *convnet* consta de una estructura multicapa formada por capas convolucionales y de reducción alternadas que finalmente terminan en capas de conexión (Figura 3.12. - (Calvo, 2017)). Por tanto, los tres tipos de capas que forman una red *convnet* son:

- Capas convolucionales.
- Capas *Pooling* o de reducción.
- Capas densamente conectadas.

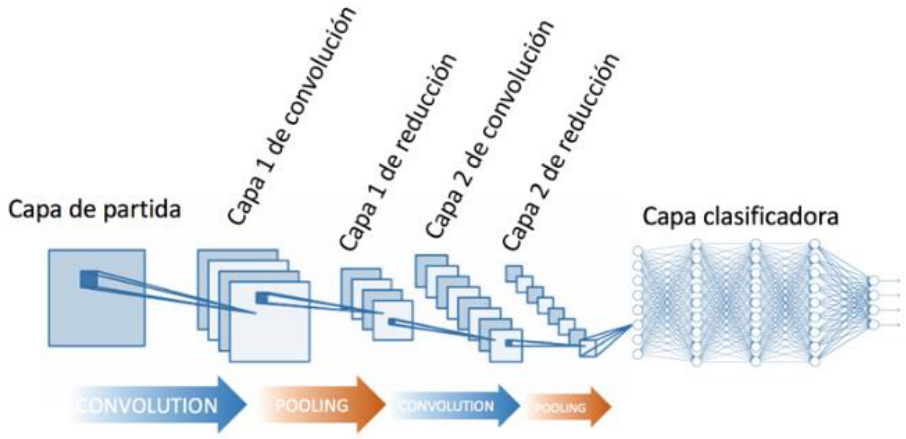


Figura 3.12. Arquitectura de una convnet.

A continuación, explicaremos cómo funcionan cada una de ellas.

3.2.1. Operación de convolución

Por una parte, las capas densas o densamente conectadas (Dense) aprenden patrones globales (implican todos los píxeles de la imagen a clasificar), mientras que las capas convolucionales aprenden patrones más específicos, patrones locales. (Figura 3.13 - (Chollet, 2020))

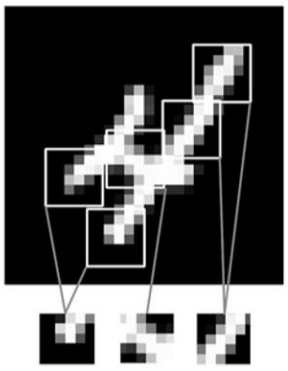


Figura 3.13. Imagen dividida en patrones locales (bordes, texturas, etc).

Esto proporciona a las *convnets* un aprendizaje más eficiente de conceptos que cada vez son más abstractos y complejos, ya que una primera capa de convolución aprende

patrones locales pequeños (como bordes), la segunda capa ya aprenderá patrones locales más grandes que ya contienen las características aprendidas por la capa anterior…y así sucesivamente (Figura 3.14. - (Chollet, 2020)).

De esta forma, un patrón determinado que haya aprendido una *convnet* en un sitio concreto de una imagen, esta lo puede reconocer en cualquier parte. A través de esta forma de aprender representaciones generalizadas, podremos realizar entrenamientos con menos muestras.

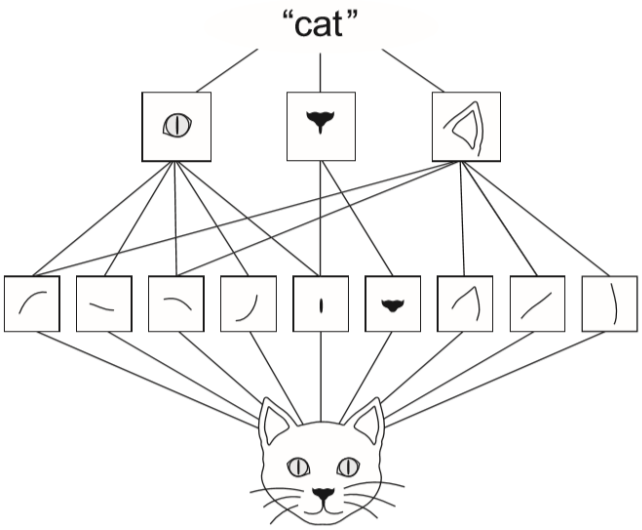


Figura 3.14. Bordes locales se combinan con objetos locales, como ojos u orejas, que se combinan para formar conceptos más complejos, como gato.

Las operaciones de convolución tienen la siguiente forma: (altura de la imagen, anchura de la imagen, canales), es decir, operan sobre tensores 3D. El número de canales representa el eje de profundidad y dependiendo de si se trata de una imagen en blanco y negro o color, el número de canales será 1 o 3 respectivamente. Estos tensores 3D son conocidos como “mapas de características”.

Las capas convolucionales extraen partes del mapa de características de entrada, aplicando la misma transformación a todas las porciones, creando un mapa de características de salida con forma de tensor 3D, llamado “mapa de respuestas”. En este caso, el eje de profundidad (canales) del tensor de salida, ya no indica el color en cuestión, sino que ya hablamos del concepto “filtro”. Los filtros computan aspectos específicos de los datos de entrada, por ejemplo, la presencia de un brazo en la imagen de entrada.

Por lo tanto, las convoluciones están definidas por dos parámetros clave:

- Tamaño de las porciones extraídas de la entada: (altura, anchura)

- Profundidad del mapa de características de salida: número de filtros codificados por la operación de convolución.

¿Cómo funcionan detalladamente las capas convolucionales? Para comprender la siguiente explicación hay que ir siguiendo el esquema que se muestra en la Figura 3.15.((Chollet, 2020)).

Las capas convoluciones (Conv2D) especifican los siguientes parámetros que pasan a la capa: Conv2D(profundidad_salida, (altura_ventana, anchura_ventana)).

La convolución funciona de forma que va deslizando las ventanas de tamaño especificado en los argumentos anteriores (altura, anchura) sobre el mapa de características de entrada (dimensión 3D), de forma que se va parando en cada situación posible y sacando la porción 3D de características posible (de forma (altura_ventana, anchura_ventana, profundidad_salida)). Cada una de estas porciones 3D se transforman en un vector 1D con forma (profundidad_salida) a través de un producto escalar con tensores llamado “kernel de convolución”. Finalmente, cada una de estas porciones transformadas se reagrupan formando el mapa de características de salida 3D con forma (altura, anchura, profundidad_salida). Hay que destacar que de esta forma cada ubicación espacial del mapa de características de entrada se corresponde con la misma ubicación en el mapa de características de salida.

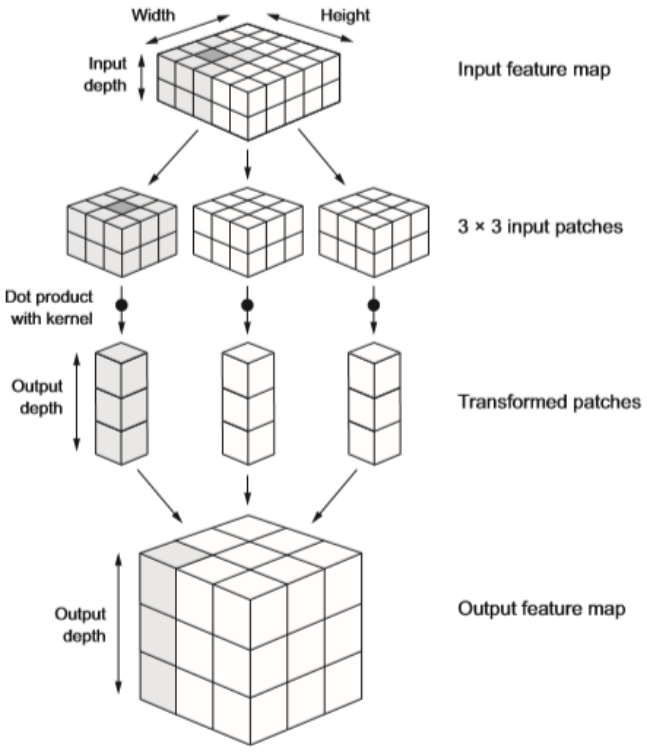


Figura 3.15. Funcionamiento de convolución.

3.2.2. Capas Max Pooling

Las operaciones de *Pooling* se caracterizan por quedarse con las características más comunes, reduciendo así la cantidad de coeficientes a procesar por el mapa de características de entrada.

Entre las posibles operaciones de *Pooling*, destacamos la operación MaxPooling. La capa MaxPooling2D se encarga de reducir a la mitad el tamaño del mapa de características. Funciona de manera que genera como salida de los mapas de características de entrada, la estimación máxima de cada canal. Es una operación similar a la de convolución, solo que en lugar de realizar un producto escalar con tensores llamado “kernel de convolución”, las porciones extraídas son transformadas por una operación con tensor max (Figura 3.16. - (Serrano, 2019)).

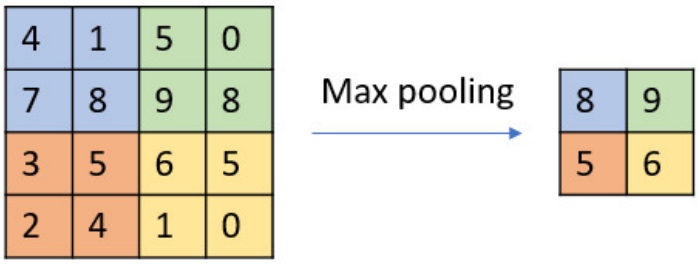


Figura 3.16. Operación MaxPooling.

Sin embargo, la razón principal de utilizar las operaciones de reducción no solo es ir dividiendo el mapa de características entre 2, sino hacer que las capas convolucionales miren ventanas de tamaño cada vez más grande (filtros espaciales más grandes).

3.2.3. Capas Flatten

La capa Flatten o capa aplanada se encarga de aplanar tensores, es decir, suprime todas las dimensiones menos una. Cambia la forma del tensor para que su forma sea igual al número de elementos que hay en el tensor. Por ejemplo, tenemos una red preentrenada VGG16 y la entrada a aplanar que nos da de forma (altura, anchura, filtros) es (4,4,512). Lo que hace la capa Flatten es multiplicar las tres dimensiones (4*4*512=8192) y las convierte

en una para que pueda entrar en la siguiente capa Dense cuya entrada es de tensores 1D, como podemos observar en la siguiente imagen (Figura 3.17):

```
>>> model.summary()
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Model)                (None, 4, 4, 512)         14714688
_________________________________________________________________
flatten_1 (Flatten)          (None, 8192)              0
_________________________________________________________________
dense_1 (Dense)              (None, 256)               2097408
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 257
=================================================================
```

Figura 3.17. Ejemplo de funcionamiento de la capa Flatten.

3.2.4. Capas densas.

Al final de esta estructura multicapa de operaciones convolucionales y de reducción, se suele utilizar capas densamente conectadas. En ellas, cada píxel es considerado una neurona separada de forma similar a un perceptrón multicapa. Veamos qué es esto y cómo funciona.

El perceptrón multicapa es usado habitualmente en clasificación multiclase, es decir, en clases exclusivas. Este concepto hace referencia a una estructura de *convnet* formada por una capa de entrada seguida de un conjunto de capas ocultas formadas por perceptrones y una capa final que será la capa de salida. Visualicemos esta organización (Figura 3.18 - (Torres, 2018)):

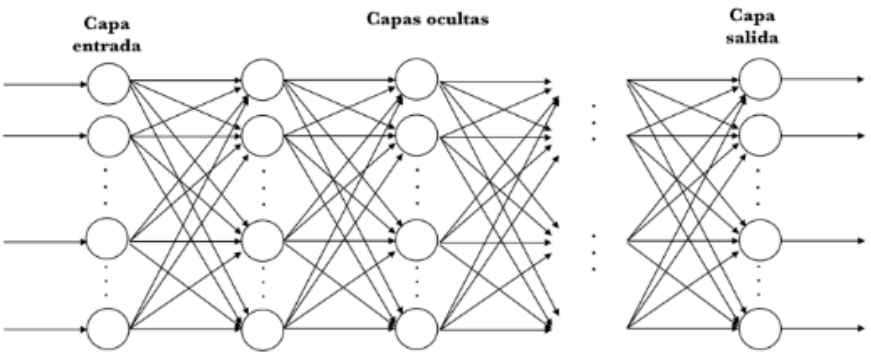


Figura 3.18. Perceptrón Multicapa.

La capa de salida es la capa de activación, cuya función de activación aplicada en este caso es softmax, de la cual hablaremos en el siguiente apartado.

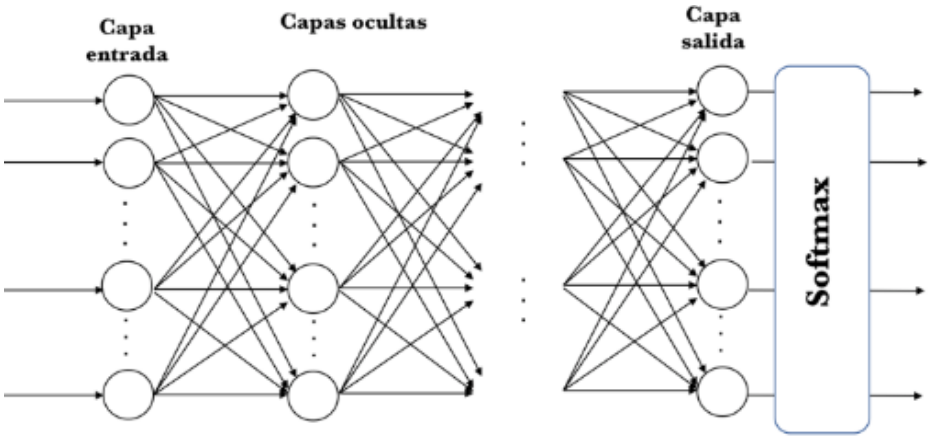


Figura 3.19. Perceptrón Multicapa.

Softmax actúa en la capa de salida (Figura 3.19 - (Torres, 2018)) de forma que a cada neurona le estima la probabilidad de predecir cada una de las clases a clasificar por la *convnet*.

3.2.5. Clasificación Softmax

La última capa de nuestra convnet es la capa clasificadora que tendrá tantas neuronas como clases a predecir. En ella aplicamos la función de activación softmax, como mencionamos en el apartado anterior. ¿Pero cómo funciona?

Esta función se basa en calcular la evidencia de que una imagen pueda pertenecer a una clase en concreto y luego la convierte en una probabilidad (Figura 3.20 - (Google Developers, s.f.)).

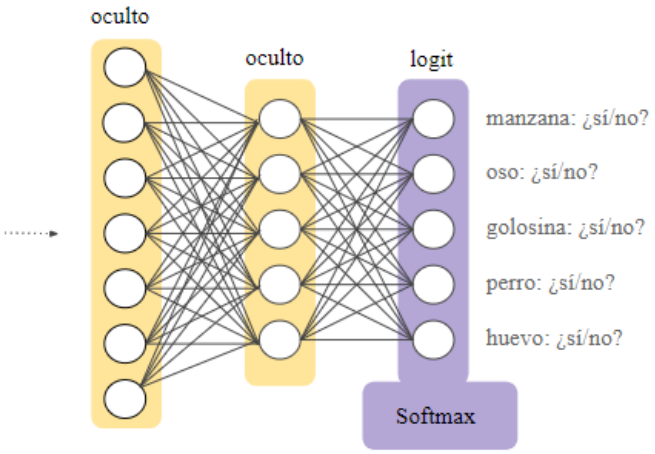


Figura 3.20. Capa softmax en una red neuronal.

Para medir la evidencia de una imagen pertenezca a una clase en particular, softmax realiza una suma ponderada de la evidencia de pertenencia de cada uno de sus píxeles a esa clase. (Torres, 2018).

Una vez que se realiza este cálculo, la evidencia de cada una de las diferentes clases, convierte las evidencias en probabilidades. Para poder realizar esta transformación, desarrolla la siguiente fórmula (Figura 3.21. - (Torres, 2018)).

$$Softmax_i = \frac{e^{evidencia_i}}{\sum_j e^{evidencia_j}}$$

Figura 3.21. Probabilidad de pertenencia a la clase i.

Para saber la probabilidad de pertenencia a una clase determinada (i), softmax emplea el valor exponencial de las evidencias calculadas y las normaliza (Figura 3.21. - (Torres, 2018)).

Capítulo 4

Estudio realizado - Presentación

4.1. Introducción

Una vez estudiado el estado del arte, se hará una descripción de las tres redes creadas que se han empleado en este trabajo de clasificación de imágenes y se explicará la diferencia entre unas y otras. Se entrenaron dos *convnets* desde cero: una más sencilla con pocas capas convolucionales y otra más compleja con más capas convolucionales y a la que se le aplicó la técnica de aumento de datos. Finalmente, también se utilizó una red preentrenada, VGG16, utilizando técnicas de *transfer learning* como extracción de características y ajuste fino. El objetivo de construir estos tres modelos es ver cómo va mejorando la precisión de clasificación según pasamos de uno a otro.

Además, se hablará un poco de la preparación y estudio que se llevó a cabo para poder hacer posible esta investigación, de las herramientas, recursos y el entorno empleados. También se explicará el reto kaggle que se intentó resolver y se hará una pequeña introducción al paquete de Python utilizado para dar solución a este problema, el paquete keras.

4.2. Casos de estudio

En este apartado, se hará un resumen de los tutoriales y cursos hechos para adentrarse en el mundo del *deep learning* y el entrenamiento de redes neuronales convolucionales. Los estudios más relevantes para poder desarrollar este trabajo fueron los siguientes:

- **Curso kaggle** (Becker, s.f.):

Se trata de un curso de *deep Learning* en Kaggle donde se nos va adentrando en el mundo del aprendizaje profundo de forma progresiva. En cada capítulo hay un vídeo explicativo y un ejercicio de programación en Python para comprobar que se ha entendido la lección explicada. En este curso se tratan los siguientes puntos:

- Una pequeña introducción al *deep Learning* para la visión por ordenador dando una descripción rápida de cómo funcionan los modelos que trabajan con imágenes.

- Entrada a la construcción de modelos a partir de convoluciones, desde bloques de construcción simples a modelos con capacidades más allá de las humanas.
- Programación con TensorFlow y keras.
- Técnica de *transfer learning* para construir modelos precisos incluso con datos limitados. Uso de redes preentrenadas.
- Técnica de aumento de datos: truco para aumentar la cantidad de datos disponibles para el entrenamiento del modelo.
- Cómo el descenso de gradiente estocástico y la propagación hacia atrás (*backpropagation*) entrenan un modelo de *deep learning*.
- Entrenar modelos desde cero: sin *transfer learning*. Especialmente para tipos de imágenes poco comunes.
- Ajustar el modelo para que sea más rápido y se reduzca el sobreajuste.

- **TensorFlow.Clasificación básica** (tensorflow.org, s.f.):

Esta es una guía para entrenar un modelo de red neuronal para clasificar imágenes de prendas de ropa como, por ejemplo, deportivos o camisetas. Simplemente es un repaso de un programa completo de Tensorflow explicando cada detalle a medida que se avanza en el *notebook*. En este se aprendió y repasó algunos de los siguientes conceptos aprendidos:

- Importación de datos MNIST.
- Exploración de datos.
- Preprocesado de datos.
- Construcción del modelo de clasificación multiclase.
- Entrenar el modelo.
- Evaluar la exactitud del modelo.

- **Tutorial Perros y gatos** (Chollet, 2020):

Aunque hay múltiples fuentes donde encontrar este tutorial, se siguió desde el libro de "Deep Learning con Pyhton" de François Chollet, ya que trata todos los puntos que se desarrollan en este trabajo, aunque utilizando conjuntos de datos pequeños para resaltar la relevancia del *deep learning* en problemas de pocos datos. Este tutorial nos demuestra que las *convnets* son la mejor herramienta para problemas de clasificación de imágenes en visión por ordenador, entrenar nuestra propia *convnet* desde cero para clasificar imágenes, entender cómo se usa el aumento de datos para combatir el sobreajuste creado en el modelo y finalmente, saber cómo utilizar una *convnet* preenetrenada aplicando las técnicas de extracción de datos y ajuste fino. También se destaca la importancia de entrenar modelos

desde cero con conjuntos de datos muy pequeños alcanzando resultados decentes, aunque estos produzcan sobreajuste.

En conclusión, nos proporciona un conjunto de herramientas que nos permitirán enfrentarnos al problema de clasificación de imágenes, pero con conjuntos de datos pequeños.

- **Tutorial Dígitos** (Chollet, 2020):

En este tutorial sacado del libro de François Chollet, como el anterior, nos muestra un primer vistazo a una red neuronal sencilla de clasificación multiclase de dígitos manuscritos, en concreto, de 10 clases (dígitos del 0 al 9). Se utiliza el conjunto de datos MNIST, recopilado por la el *National Institute of Standards and Tecnology*. En esta lección se familiarizó con los primeros conceptos relacionados con este mundo, especialmente, la función de pérdida, optimizador y métricas para medir la exactitud del modelo (*accuracy*). Nos enseña cómo crear y entrenar una red neuronal que clasifique dígitos manuscritos en menos de 20 líneas de código.

- **Tutorial *Deel Learning*** (Caparrini, 2018):

Este repaso de *Machine Learning,* creado por Fernando Sancho Caparrini de la Universidad de Sevilla, ha aportado notebooks realmente útiles para poder comprender todos estos nuevos conceptos a los que nos enfrentamos. Aunque hay documentos relacionados con temas que no son relevantes en este trabajo, se han utilizado las siguientes carpetas:

- "Introducción a keras": donde utiliza distintos notebooks para impartir una introducción a Tensowflow, keras, y un ejemplo de clasificación binaria y multiclase.
- "Regularización": para saber cómo optimizar la precisión de nuestro modelo a través de distintas técnicas para acabar con el sobreajuste que se crea.
- "Redes convolucionales": donde además de incluir el tutorial de "Perros y gatos" ya visto, incluye un notebook de cómo usar una red preentrenada y de cómo visualizar el aprendizaje de los modelos entrenados.

4.3. Reto Kaggle

4.3.1. Descripción de la competición y *dataset* proporcionado

El objetivo de este reto propuesto por kaggle es poder detectar distintos tipos de lesiones cutáneas en imágenes dermatoscópicas a través de una herramienta que nos permita realizar este diagnóstico automatizado.

En este caso, la tarea consistirá en clasificar un conjunto de imágenes dermatoscópicas en 7 tipos de lesiones cutáneas, es decir, el objetivo es realizar una predicción automatizada de imágenes de distintas clases.

El conjunto de datos proporcionado por el reto es el conocido conjunto HAM10000 (*Human Against Machine*) utilizado en múltiples competiciones. Está formado por 10015 imágenes en formato JPG, una serie de imágenes dermatoscópicas de diversas fuentes que será la entrada de la herramienta clasificadora que tendremos que desarrollar. Este conjunto de datos está disponible de forma pública.

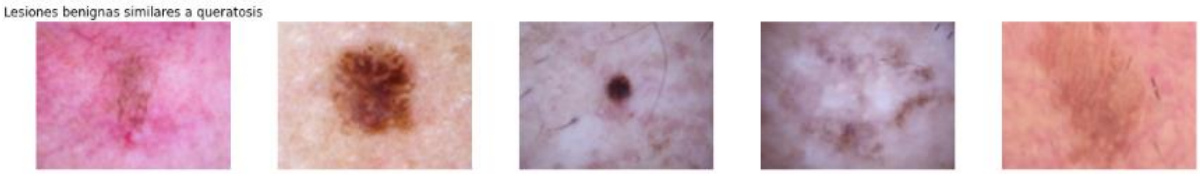
Para situarnos en el contexto del problema, el cáncer de piel es la neoplasia maligna de carácter humano más común. Se diagnostica principalmente de forma visual, comenzando con un examen clínico inicial, seguido potencialmente de un análisis dermatoscópico, una biopsia y un examen histopatológico. La clasificación automatizada de las lesiones cutáneas mediante imágenes es una tarea compleja debido a la variabilidad de grano fino en la apariencia de las lesiones cutáneas. Por eso, el objetivo de la creación de este clasificador es utilizar esta herramienta de *deep learning* como ayuda para alcanzar una mayor predicción a la hora de detectar cáncer de piel en comparación al ojo humano.

Veamos las 7 clases de lesiones cutáneas a predecir y 5 ejemplos de cada una de ellas (Figura 4.1):

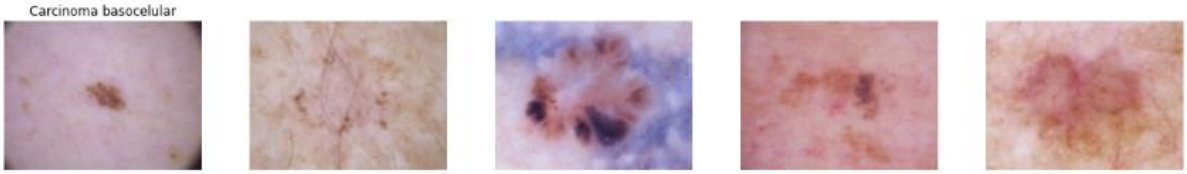
1. Nevos melanocíticos

2. Melanoma

3. Lesiones similares a la queratosis benigna

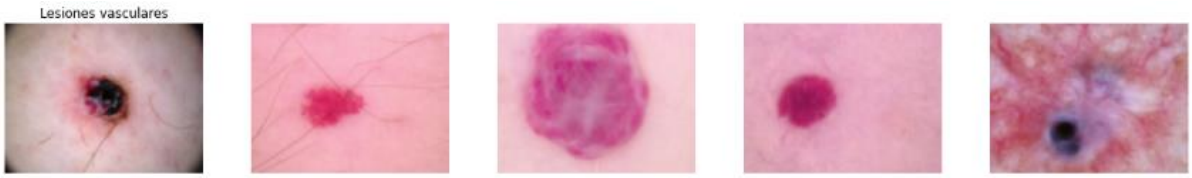


4. Carcinoma basocelular



5. Queratosis

6. Lesiones vasculares



7. Dermatofibroma

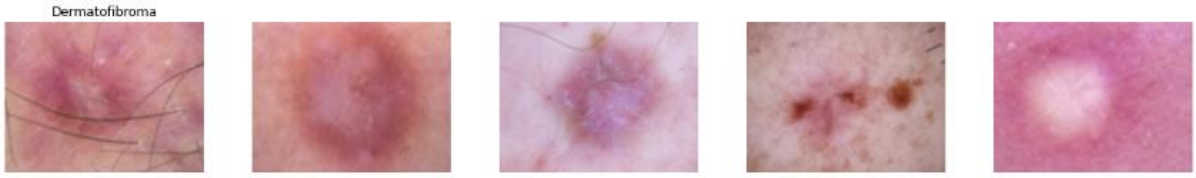


Figura 4.1: Ejemplos de imágenes de cada clase proporcionadas por el *dataset* del reto kaggle.

Además de las 10015 imágenes que utilizaremos para el entrenamiento de nuestros modelos, se nos proporciona un CSV llamado “HAM10000_metadata.csv” con las siguientes columnas de información:

- **lesión_id**: identificador de la lesión cutánea de la imagen a la que hace referencia.
- **image_id**: identificador de la imagen en concreto.
- **dx**: etiqueta que identifica el tipo de lesión.
- **dx_type**: tipo de validación técnica de cada imagen.
- **age**: edad del paciente.
- **sex**: sexo del paciente.
- **localization**: localización de la lesión en el paciente

	lesion_id	image_id	dx	dx_type	age	sex	localization
0	HAM_0000118	ISIC_0027419	bkl	histo	80.0	male	scalp
1	HAM_0000118	ISIC_0025030	bkl	histo	80.0	male	scalp

Figura 4.2. CSV de entrada.

A este CSV de entrada se la harán distintas modificaciones para procesar y organizar los datos que veremos más adelante.

Sin embargo, la respuesta del clasificador creado será un CSV con una fila codificada. En esta se representan clasificaciones binarias para cada imagen de entrada con el diagnóstico asignado a cada una de ellas.

4.3.2. Participantes en la competición

Para poder empezar con la solución al problema propuesto, después de toda la investigación realizada mediante tutoriales y cursos (Apartado 4.2.), se han estudiado las mejores soluciones aportadas por los participantes de este reto kaggle.

En primer lugar, como en este trabajo se han aplicado tanto redes entrenadas desde cero como redes preentrenadas, la visión de estudio fue mucho más amplia que si nos centrásemos en una sola solución en concreto. El objetivo del notebook creado es, como se mencionó antes, construir 3 posibles soluciones con distintas arquitecturas en las cuales se vea una mejora de precisión en la clasificación de manera progresiva. Además, hay que tener en cuenta que el paquete utilizado es Keras y que las mejores soluciones (mayor porcentaje de precisión) se alcanzaron con el paquete Pytorch (precisión>90%).

A continuación, se mostrará un pequeño resumen de los trabajos considerados más destacables y relevantes a la hora de abordar este proyecto desde la perspectiva deseada (Tabla 4.1. - (kaggle, s.f.)).

Participante	Participante	Red usada	Aumento de Datos	Resultado precisión
1	Manu Siddhartha	Red entrenada desde cero	Sí	77.0344%
2	Zuhdi	Red entrenada desde cero	Sí	75%
3	Alex Reyes	Red preentrenada (VGG16)	No	76%

Tabla 4.1. Participantes con mejores resultados del reto kaggle.

Realmente, estos *notebooks* proporcionados como solución a la competición, sirvieron de mucha ayuda, en especial a la de hora de tratar los datos, ya que en el estudio realizado mediante cursos y tutoriales de iniciación al entrenamiento de redes neuronales sí se recalca y se adentra en el tipo de clasificación binaria (por ejemplo: clasificación de perros y gatos, dos clases), pero apenas se profundiza en el tipo de clasificación multiclase. ¿Qué destacamos de cada *dataset*?

- A través del dataset 1 (Manu Siddhartha), podemos ver como se crea un diccionario de ruta de imagen que luego se usará para añadir nuevas columnas al *dataset* original y las cuales aportarán comodidad al tratamiento de datos futuro. En él se entrena una red desde cero, aplicando técnicas como *dropout*, aumento de datos y, además, algo nuevo hasta ahora pero que abordaremos más adelante, la función "ReduceLROnPlateau".

- El *dataset* 2 es muy parecido al anterior, mismo tratamiento y procesamiento de datos, mismas técnicas aplicadas que la solución anterior, pero distinta arquitectura de red, por lo que fue curioso ver cómo variaba una solución u otra en base a si su arquitectura de red era más sencilla o más compleja en cuanto a número de capas convolucionales y capas *dropout.*

- El *dataset* 3, usa la red preenetrenada elegida para este proyecto, VGG16, aplicando la técnica de *trasnfer learning* "Ajuste fino" al descongelar capas de la base preentrenada y entrenarlas con el modelo creado. Aunque algunas ideas que se plasman en esta solución sirven de ayuda, la información de más utilidad para llevar a cabo el empleo de una red preentrenada fue proporcionada por el libro de Francois Chollet. (Chollet, 2020)

No obstante, se han visualizado muchas otras aportaciones de participantes, pero se consideraron estas 3 más útiles porque el código desarrollado en ellas era más semejante al estudiado en el Apartado 4.2. Como se dijo antes, recalcamos la ayuda proporcionada por el *dataset* 1 en los siguientes pasos clave:

- Procesamiento y limpieza de datos.
- Redimensionamiento de imágenes.

4.4. Tecnologías utilizadas

4.4.1. Keras

Keras es una biblioteca de *deep learning* para Python que facilita una forma fácil y rápida de construir y entrenar cualquier tipo de modelo de *deep learning*. Entre sus características clave destacamos las siguientes:

- Posibilita ejecutar el código tanto en CPU o GPU sin interrupciones.
- Cuenta con una API fácil de usar, lo que hace que sea apropiado para crear de forma rápida casi cualquier tipo de modelo.
- Tiene un soporte integrado para redes convolucionales.

El flujo de trabajo de keras se organiza siguiendo estos pasos:

1. Definir un conjunto de datos de entrenamiento.
2. Definir un modelo (red de capas) en la que se asigne las entradas a los objetivos.
3. Elección de una función de pérdida y un optimizador para llevar a cabo el proceso de aprendizaje.
4. Usamos el método fit() para que el modelo itere con los datos de entrenamiento definidos.

Se puede definir un modelo de dos formas: a través de la clase Sequential() o usando la API funcional. La primera es la más común, utilizada solo para conjuntos lineales de capas.

A continuación, se expondrá un breve ejemplo de un modelo de 2 capas definido con la clase Sequential(). En él podremos observar con claridad los pasos del flujo de trabajo de keras antes explicados:

En esta imagen (Figura 4.3. - (Chollet, 2020)) podemos ver cómo se pasa la forma esperada de los datos de entrada a la primera capa (input_shape). El modelo está definido por dos capas densas (Capítulo 3.2.3).

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(32, activation='relu', input_shape=(784,)))
model.add(layers.Dense(10, activation='softmax'))
```

Figura 4.3. Ejemplo de modelo keras parte 1.

A continuación, vamos a compilar y así se podrá llevar a cabo el proceso de aprendizaje (Figura 4.4. - (Chollet, 2020)). Para ello, tenemos que definir un optimizador (RMSprop), una función de pérdida (mse) y la métrica que vamos a monitorizar durante el entrenamiento (*accuracy*). Además, también se puede especificar el hiperparámetro *learning rate* como argumento del optimizador seleccionado.

```
from keras import optimizers

model.compile(optimizer=optimizers.RMSprop(lr=0.001),
              loss='mse',
              metrics=['accuracy'])
```

Figura 4.4. Ejemplo de modelo keras parte 2.

Finalmente, para completar el proceso de aprendizaje, la función fit() (Figura 4.5. - (Chollet, 2020)) permite que los datos de entrada iteren con el modelo de la siguiente forma: se encarga de pasar matrices Numpy (tensores 2D) de datos de entrada al modelo. Además, especificamos los hiperparámetros de tamaño de lote (*batch size*) y el número de épocas (*epochs*).

```
model.fit(input_tensor, target_tensor, batch_size=128, epochs=10)
```

Figura 4.5. Ejemplo de modelo keras parte 3.

Llegados a este punto, ya tenemos una idea del flujo de trabajo que realiza keras. Más adelante, después de crear nuestros modelos con keras para darle solución al problema de clasificación de imágenes de lesiones cutáneas, veremos cómo ajustamos los hiperparámetros de los modelos para conseguir los resultados deseados.

4.4.2. Tensorflow

Keras proporciona bloques de construcción de modelos en Deep learning de alto nivel. Para ello, keras cuenta con un motor formado por una biblioteca especializada de tensores. Además, maneja su motor de forma modular (Figura 4.6. - (Chollet, 2020)), de forma que múltiples motores se pueden conectar sin interrupciones.

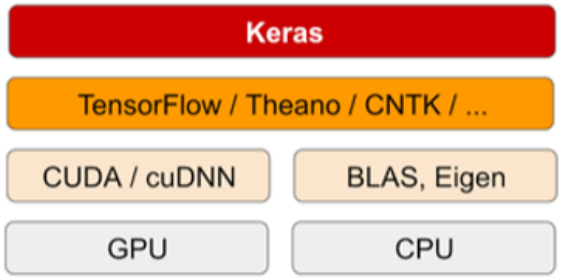


Figura 4.6. Pila de software y hardware de Deep Learning.

Como podemos ver en la imagen anterior (Figura 4.6. - (Chollet, 2020)), Tensorflow, Theano y CNTK son algunas de las plataformas que dan soporte a keras y que son herramientas principales para el *deep learning* hoy en día. Sin embargo, es Tensorflow la que se ha adoptado a nivel global, ya que es más ajustable y es la que más se adapta a las necesidades de *deep learning*. Tensorflow le da a keras el poder de ejecutarse sin ningún tipo de problema tanto en CPU como en GPU, diferenciando estas dos opciones en la biblioteca de operaciones proporcionada por tensorflow, una de bajo nivel y otra bien optimizada llamada NVIDIA CUDA Deep Neural (cuDNN) respectivamente.

Tensorflow le proporciona a keras una API de alto nivel, tf.keras, para entrenar y construir modelos en *deep learning*. Se caracteriza principalmente por ser:

- Amigable: simple y optimizada para casos de uso común.
- Configurable y modular: para construir los modelos se conectan bloques que son configurables entre sí.
- Gran extensión: permite expresar nuevas ideas de investigación, crear nuevas capas, métricas, funciones de pérdida, etc.

4.4.3. Google Colab: machine leraning en la nube

Uno de los principales problemas de entrenar redes neuronales es el gran coste computacional que esto requiere. Por esta razón, es recomendable ejecutar el código de *deep*

learning en GPU, ya que este proceso en CPU es lentísimo, especialmente el procesamiento de imágenes. Instalar una GPU en el ordenador podría multiplicar la velocidad del proceso por 5 o por 10, sin embargo, como no contamos con este recurso, una alternativa es usar GPU en la nube, solución proporcionada por Google Colab.

Google Colab es el entorno de desarrollo gratuito de Jupyter Notebook elegido para construir la herramienta de clasificación como solución al problema de diagnóstico automatizado de cáncer de piel y se ejecuta completamente en la nube.

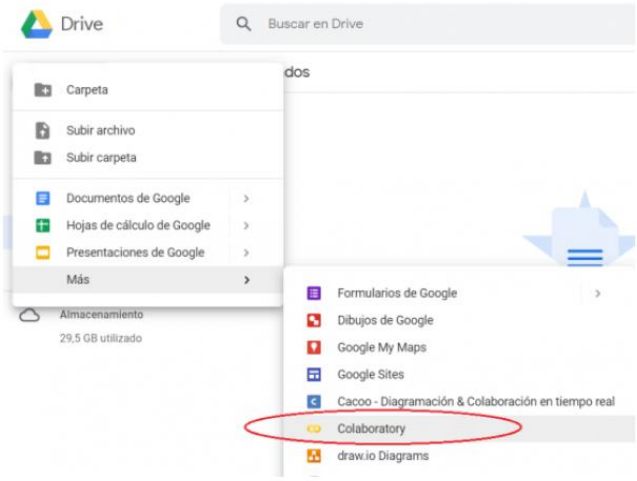


Figura 4.7. Entorno Google Colab.

Para acceder a esta herramienta, simplemente hay que entrar en Google Drive y seleccionar la pestaña “Colaboratory” (Figura 4.7. - (Sanz, 2019)) para que cree un nuevo notebook, en el cual ejecutaremos nuestro código en Python.

Una vez que ya tenemos nuestro cuaderno, podremos configurarlo de forma que se ejecute el código en GPU (Figura 4.8. - (Sanz, 2019)) de forma gratuita en lugar de CPU, dándonos una máquina con esos ajustes.

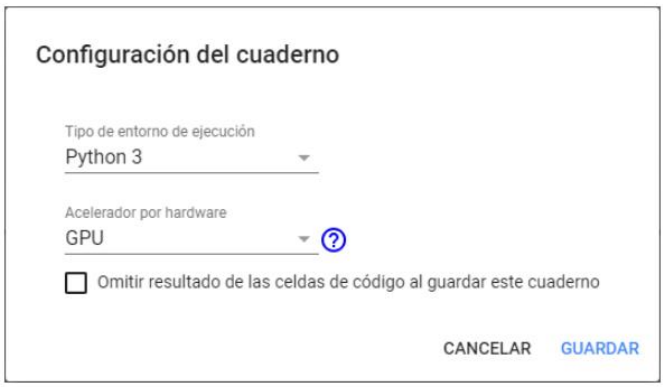


Figura 4.8. Configurar el entorno en GPU.

¿Pero qué problemas presenta Google Colab?

Limitaciones de recursos por parte de Google. Al tratarse de una GPU en la nube, en el momento que haya demasiados usuarios conectados al mismo tiempo, puede ser realmente difícil conectarse a una GPU o en caso de que se realice una operación de larga duración, puede que el entorno se desconecte.

Para saber si estamos conectados a una GPU, basta con compilar las siguientes líneas de código (Figura 4.9. - (Sanz, 2019)):

Figura 4.9. Verificación de la existencia de una GPU disponible.

Y la respuesta será afirmativa en caso de que se reproduzca lo siguiente: “Encontrada GPU: /device:GPU:0”.

Ante estos problemas, que se presentan de forma habitual al querer trabajar en este entorno de desarrollo, Google Colab ofrece la versión Google Colab Pro, versión que ya no es gratuita ($9.99), pero que resulta mucho más útil en el momento que el servicio se encuentra muy saturado. Los beneficios de esta alternativa de pago frente a la gratuita son los siguientes:

- GPU más rápidas: pasamos de usar K80 a una Nvidia Tesla T4 y P100.
- Acceso prioritario a GPU: menos tiempo esperando a conectarse a una máquina con GPU.
- Notebook de mayor duración: en caso de realizar operaciones de larga duración y de mayor coste, este notebook tendrá un mayor tiempo de inactividad antes de desconectarse.
- Más memoria RAM: lo que proporciona un mayor rendimiento.

La conclusión es que Google Colab es realmente útil como solución de machine learning en la nube, pese a sus problemas de limitación de recursos y, la alternativa Google Colab Pro, mejora un poquito más esta herramienta.

4.5. Tratamiento de datos

Antes de construir las herramientas clasificadoras es importante manipular los datos y organizarlos de forma que su tratamiento sea lo más sencillo posible. En este apartado nos centramos en los siguientes puntos del trabajo:

1. Procesamiento de datos.
2. Limpieza de datos.
3. Análisis de datos
4. Redimensionamiento de imágenes

1. Procesamiento de datos.

Una vez importadas las librerías y herramientas necesarias para trabajar con keras y tensorflow en Python, empezamos a procesar los datos. La carpeta original que se nos proporciona en la competición para realizar este estudio presenta 10015 imágenes dermatoscópicas en formato .jpg separadas en dos carpetas: HAM10000_iamges_part1 y HAM10000_images_part2. Para poder juntar todas las imágenes, fusionamos las dos carpetas, ¿cómo? Recopilando todos los elementos con extensión .jpg del directorio donde se encuentran estas dos carpetas (Figura 4.10).

```
directorio_base = os.path.join('/content/drive/My Drive/tfg/TFGGG', '')

# Fusionamos las imágenes en un solo direcotorio

imagenes = {os.path.splitext(os.path.basename(x))[0]: x
                    for x in glob(os.path.join(directorio_base, '*', '*.jpg'))}
```

Figura 4.10. Fragmento explicativo de código1. Recopilación de los datos de entrada.

Luego, se creará un diccionario de ruta de imagen en el que se le asigna a cada tipo de lesión una etiqueta (Figura 4.11).

```
# Diccionario

tipo_lesion = {
    'nv': 'Melanocíticos',
    'mel': 'Melanoma',
    'bkl': 'Lesiones benignas similares a queratosis',
    'bcc': 'Carcinoma basocelular',
    'akiec': 'Queratosis actínicas',
    'vasc': 'Lesiones vasculares',
    'df': 'Dermatofibroma'
}
```

Figura 4.11. Fragmento explicativo de código 2. Creación de un diccionario de ruta de imagen.

La creación de este diccionario para hacer referencia a cada una de las 7 clases de lesión cutánea y las tres nuevas columnas que vamos a añadir en el *dataset* original (Apartado 4.3.1) nos proporcionarán mucha más comodidad a la hora de trabajar luego con los datos. Las columnas que se van a añadir son "**path**" con la ruta de cada imagen del conjunto donde se fusionaron ambas carpetas, "**cell_type**" para indicar el nombre completo de la lesión a la que pertenece la etiqueta del *dataset* original y las cuales especificamos en el diccionario creado y "**cell_type_idx**", en la que categorizamos el tipo de lesión en del 0 al 6 en función de su posición en el diccionario (Figura 4.12).

	path	cell_type	cell_type_idx
	/content/drive/My Drive/tfg/TFGGG/HAM10000_ima...	Lesiones benignas similares a queratosis	2

Figura 4.12. Fragmento explicativo de código 3. Visualización de la adición de nuevas columnas.

2. Limpieza de datos.

Una vez que se consigue añadir esta información útil al *dataset* original, se nos presenta un nuevo reto: limpiar los valores faltantes o *missing values*, aquellos datos que no constan debido a cualquier acontecimiento. Para ello, miramos en qué columna del *dataset* hay valores nulos y los reemplazaremos por su media en esa columna. Esto es posible gracias a los siguientes comandos:

```
[ ] piel_df.isnull().sum()

lesion_id         0
image_id          0
dx                0
dx_type           0
age              57
sex               0
localization      0
path              0
cell_type         0
cell_type_idx     0
dtype: int64
```

Figura 4.13. Fragmento explicativo de código 4. Visualizar el número de *missing values* por cada columna del *dataset.*

Podemos ver que solo el campo "edad" presenta valores faltantes (Figura 4.13). Para limpiar estos valores nulos y sustituirlos por su media, usaremos la siguiente línea de código:

```
[ ]  piel_df['age'].fillna((piel_df['age'].mean()), inplace=True)
```

Figura 4.14. Fragmento explicativo de código 5. Sustituir los *missing values* por el valor medio de la columna en cuestión.

3. Análisis de datos.

Procedemos a explorar las características del conjunto de datos y las diferentes distribuciones de los campos que se especifican en el *datatset* proporcionado.

- ¿Cuántas imágenes encontramos de cada tipo de lesión cutánea?

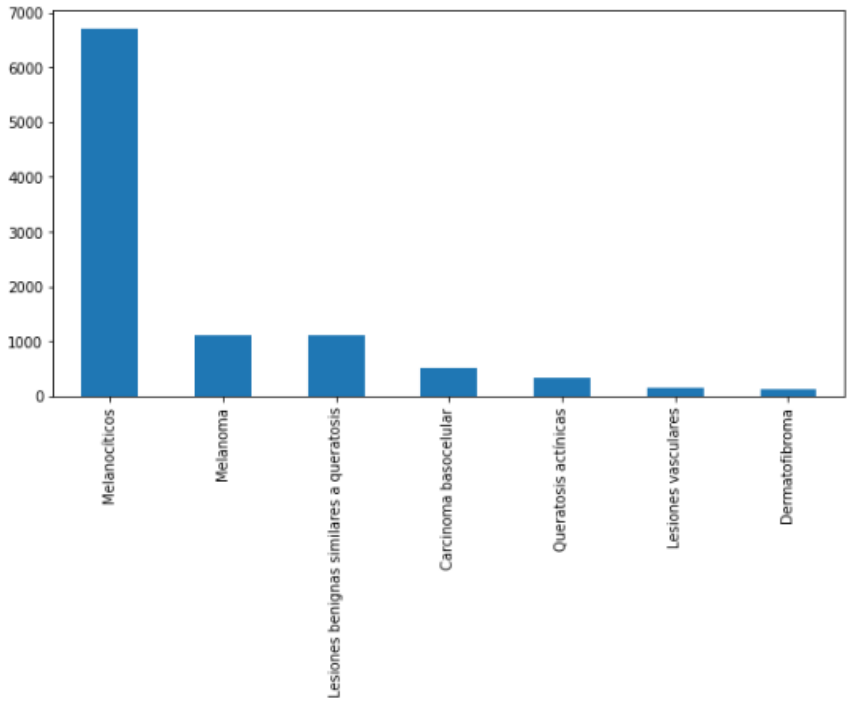


Figura 4.15. Cantidad de casos por tipo de lesión cutánea.

En esta gráfica (Figura 4.15), podemos ver que la cantidad de imágenes de cada tipo de lesión no es equitativa, ya que tenemos una gran cantidad de imágenes de tipo “Melanocíticos” (>6000) y, sin embargo, muy pocas de las otras 6 clases (=<1000).

Esto supone un problema en cuanto a la fiabilidad del clasificador construido, ya que para conseguir una precisión real de cada clase es necesario que la segmentación del conjunto de datos de entrenamiento y validación presente la misma proporción en las 7 clases a clasificar.

- ¿Cuá es la proporción de los tipos de diagnósticos relacionados con cada imagen?

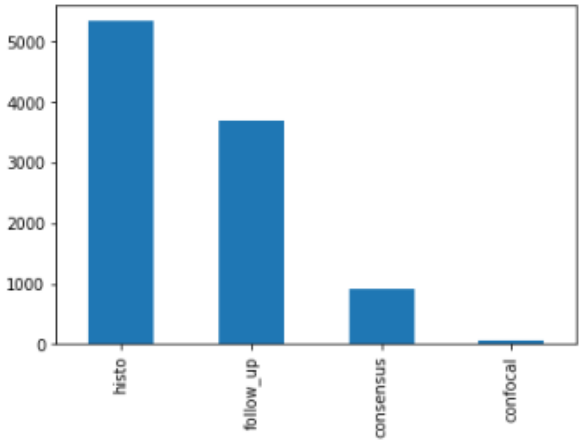


Figura 4.16. Número de imágenes por tipo de validación técnica.

Como se ha mencionado anteriormente, la columna “dx_type” nos indica el tipo de validación técnica de cada imagen y podemos observar 4 categorías en este campo:

  - “histo”: diagnóstico por histopatología.
  - “confocal”: diagnóstico por microscopía confocal.
  - “follow_up”: diagnóstico por seguimiento, solo los nevos monitoreados por dematoscopia digital que nos mostraron ningún cambio durante 3 visitas de seguimiento o en 1 año y medio.
  - “consensus”: diagnóstico por consenso de expertos.

Podemos ver (Figura 4.16) que la mayor parte de las imágenes fueron validadas por histologías.

- ¿En qué parte del cuerpo encontramos más lesiones cutáneas?

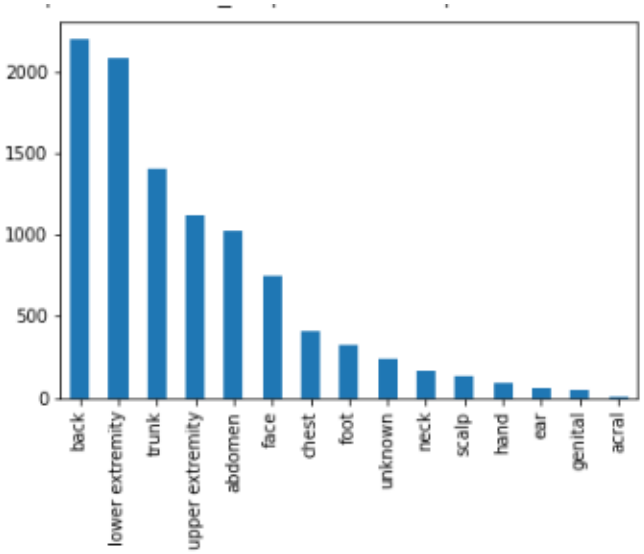


Figura 4.17. Número de lesiones cutáneas por zona del cuerpo afectada.

Las regiones más afectadas por el cáncer de piel son la espalda, las extremidades inferior y superior y el tronco (Figura4.17).

- ¿Es más común el cáncer de piel en hombres o en mujeres?

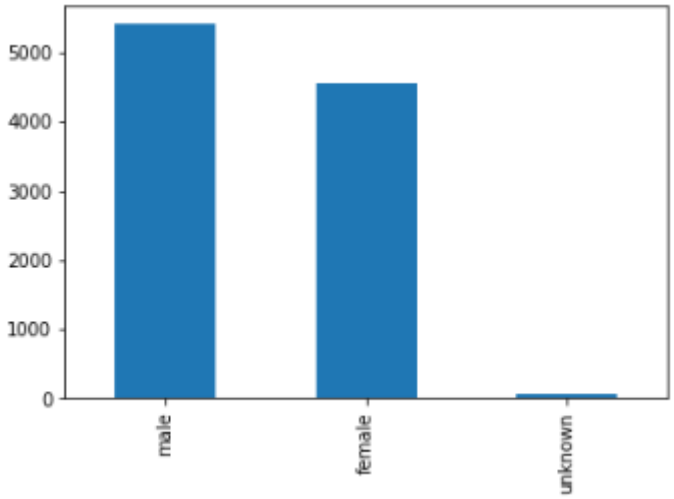


Figura 4.18. Casos de cáncer de piel por hombres y mujeres.

Las tasas de lesiones cutáneas son más altas en género masculino (Figura 4.18).

4.Redimensionamiento de imágenes.

Tensorflow no puede manejar imágenes de dimensiones superiores a 100x75 (ancho por alto), por lo que redimensionamos todas las imágenes de entrada, pasando de una dimensión original definida por 450x600x3 (altura, anchura, profundidad) a una dimensión de 100x75.

Esto será posible gracias a la función resize() (Figura 4.19):

```
#redimensionamos todas las imágenes
piel_df['image'] = piel_df['path'].map(lambda x: np.asarray(Image.open(x).resize((100,75))))
```

Figura 4.19. Fragmento explicativo de código 6. Redimensionamiento de imágenes.

4.6. Redes empleadas en el entrenamiento

En esta sección, veremos la arquitectura de las tres *convnets* empleadas en nuestro estudio. Empezaremos por una *convnet* entrenada desde cero, sencilla, con pocas capas convolucionales y sin ninguna técnica aplicada, estableciendo así un modelo de referencia para poder llegar a lo que queremos conseguir.

A continuación, se mostrará una *convnet* más compleja, una evolución de la anterior, con más capas convolucionales y en la que aplicaremos el aumento de datos, una técnica potente para reducir el sobreajuste, visualizando así una mejora de precisión con respecto a la *convnet* anterior.

Finalmente, se aplicarán dos técnicas esenciales de *transfer learning*, extracción de características y ajuste fino, a una red preentrenada (VGG16).

Estas tres soluciones serán nuestras herramientas para poder abordar el problema de clasificación de imágenes de lesiones cutáneas y poder elegir la mejor en base a los resultados obtenidos con cada una de ellas.

4.6.1. *Convnet* entrenada desde cero sin aumento de datos

La primera *convnet* creada presenta una arquitectura simple (Figura 4.20) organizada en una sucesión de dos capas convolucionales (Conv2D con activación relu, la cual se utiliza para no agregar linealidad a la red creada) y una capa MaxPooling. Aplicamos dos capas convolucionales seguidas para aumentar la capacidad de la red y luego aplicamos la capa MaxPooling para reducir el tamaño de los mapas de características y que estos no sean demasiado grandes cuando lleguen a la capa Flatten.

Finalmente, como se trata de un problema de clasificación multiclase, en concreto de 7 clases a predecir, acabaremos la red con 7 unidades (capa Dense de tamaño 7) y activación softmax.

- Instancias de la *convnet* pequeña.

```
input_shape = (75, 100, 3)
num_classes = 7

model1 = Sequential()
model1.add(Conv2D(32, kernel_size=(3, 3),
                  activation='relu',
                  input_shape=input_shape))
model1.add(Conv2D(64, (3, 3), activation='relu'))
model1.add(MaxPool2D(pool_size=(2, 2)))
model1.add(Dropout(0.25))
model1.add(Flatten())
model1.add(Dense(128, activation='relu'))
model1.add(Dropout(0.5))
model1.add(Dense(num_classes, activation='softmax'))
```

Figura 4.20. Fragmento explicativo de código 7. Modelo de una red sencilla entrenada desde cero.

- Seguimiento de cómo cambian las dimensiones de los mapas de características en la sucesión de las capas.

Empezamos con entradas de tamaño 75x100 (alto por ancho) y acabamos con mapas de características de tamaño 35x48 (justo antes de la capa Flatten).

Como podemos observar (Figura 4.21), la profundidad de los mapas de características aumenta de manera progresiva (de 32 a 64), pero en cambio, el tamaño se reduce (de 75x100 a 35x48). Este patrón lo encontraremos en casi todas las *convnets*.

```
model1.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 73, 98, 32)        896
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 71, 96, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 35, 48, 64)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 35, 48, 64)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 107520)            0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               13762688
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_2 (Dense)              (None, 7)                 903
=================================================================
```

Figura 4.21. Fragmento explicativo de código 8. Sucesión de características de las capas del modelo 1.

Como sabemos, el siguiente paso del flujo de trabajo de keras es la compilación del modelo a través de la función compile(), donde especificamos el optimizador aplicado (Adam), el valor del hiperparámetro *learning rate*, la función de pérdida (categorical_crossentropy) y la métrica de evaluación (*accuracy*). De cada uno de estos argumentos se profundizará en el siguiente capítulo, estas líneas de código (Figura 4.22) se van a repetir en las 3 *convnets*:

```
[ ]  model1.compile(optimizer= optimizers.adam(lr=0.0001),
                    loss='categorical_crossentropy',
                    metrics=['accuracy'])
```

Figura 4.22. Fragmento explicativo de código 9. Compilar modelo.

Finalmente, usamos el método fit() (Figura 4.23) para ajustar el modelo a los datos de entrenamiento y especificamos el tamaño de lote y el número de épocas en los que se va a desarrollar el entrenamiento. Empezamos a entrenar.

```
history = model1.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))
```

Figura 4.23. Fragmento explicativo de código 10. Función fit.

4.6.2. *Convnet* entrenada desde cero con aumento de datos

En el siguiente modelo creado, aumentamos la complejidad tanto de arquitectura como a la hora de entrenar la *convnet* construida. En este caso, la arquitectura estará formada por una mayor sucesión de múltiples capas convolucionales y capas MaxPooling alternas (Figura 4.24), lo que aumentará el tamaño de la red y su capacidad. Además, también se añaden más capas *dropout* para suavizar el sobreajuste causado.

- Instancias de la convnet más compleja.

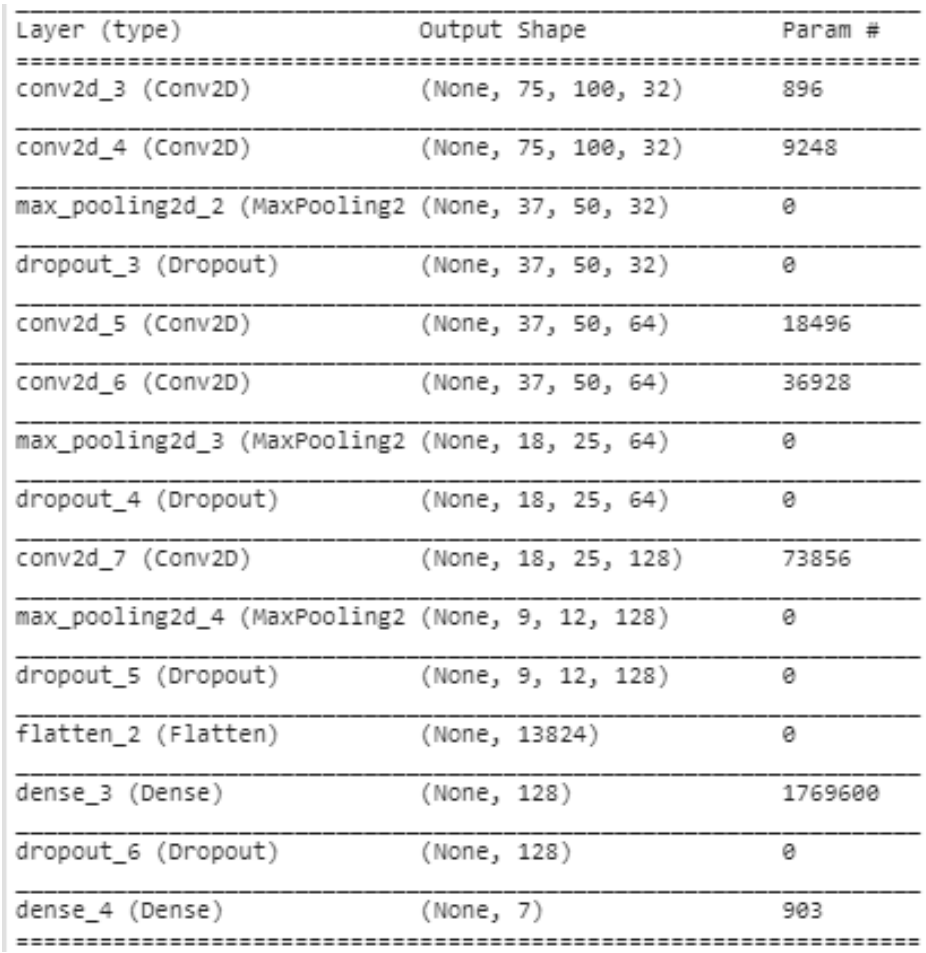
```
input_shape = (75, 100, 3)
num_classes = 7
model2 = Sequential()
model2.add(Conv2D(32, 3, padding='same', activation='relu', input_shape=input_shape))
model2.add(Conv2D(32, 3, padding='same', activation='relu'))
model2.add(MaxPool2D(pool_size = (2, 2)))
    #Capa Dropout
model2.add(Dropout(0.25))
model2.add(Conv2D(64, 3, padding='same', activation='relu', input_shape=input_shape))
model2.add(Conv2D(64, 3, padding='same', activation='relu'))
model2.add(MaxPool2D(pool_size=(2, 2)))
    #Capa Dropout
model2.add(Dropout(0.4))

model2.add(Conv2D(128, 3, padding='same', activation='relu'))
model2.add(MaxPool2D(pool_size=(2, 2)))
    #Capa Dropout
model2.add(Dropout(0.5))
model2.add(Flatten())
model2.add(Dense(128, activation='relu'))
    #Capa Dropout
model2.add(Dropout(0.55))
model2.add(Dense(7, activation='softmax'))
```

Figura 4.24. Fragmento explicativo de código 11. Arquitectura del modelo 2.

- Seguimiento de cómo cambian las dimensiones de los mapas de características en la sucesión de las capas del modelo 2.

En este caso, aumentamos la profundidad de los mapas de características de 32 a 128 y reducimos el tamaño de 75x100 (altura x anchura) a 9x12 (Figura 4.25).



Layer (type)	Output Shape	Param #
conv2d_3 (Conv2D)	(None, 75, 100, 32)	896
conv2d_4 (Conv2D)	(None, 75, 100, 32)	9248
max_pooling2d_2 (MaxPooling2	(None, 37, 50, 32)	0
dropout_3 (Dropout)	(None, 37, 50, 32)	0
conv2d_5 (Conv2D)	(None, 37, 50, 64)	18496
conv2d_6 (Conv2D)	(None, 37, 50, 64)	36928
max_pooling2d_3 (MaxPooling2	(None, 18, 25, 64)	0
dropout_4 (Dropout)	(None, 18, 25, 64)	0
conv2d_7 (Conv2D)	(None, 18, 25, 128)	73856
max_pooling2d_4 (MaxPooling2	(None, 9, 12, 128)	0
dropout_5 (Dropout)	(None, 9, 12, 128)	0
flatten_2 (Flatten)	(None, 13824)	0
dense_3 (Dense)	(None, 128)	1769600
dropout_6 (Dropout)	(None, 128)	0
dense_4 (Dense)	(None, 7)	903

Figura 4.25. Fragmento explicativo de código 12. Sucesión de características de las capas del modelo 2.

Como ya se ha dicho, en esta *convnet* vamos a aplicar la técnica de aumento de datos (*Data Augmentation*) para combatir el sobreajuste causado. Para ello determinamos la configuración de aumento de datos mediante un generador “ImageDataGenerator” el cual se utilizará para generar las transformaciones aleatorias a realizar en las imágenes leídas por este generador. Los cambios que se van a realizar sobre las imágenes originales para crear mayor cantidad de datos son las siguientes:

- Ratation_range: se rotan las imágenes tantos grados como se le indique (entre 0-180) de forma aleatoria. En estos modelos se aplica una rotación de 10 grados.
- Zoom_range: hace zoom dentro de las imágenes de forma aleatoria.
- Width_shift_range y height_shift_range: trasladan las imágenes horizontal o verticalmente de manera aleatoria.

```
datagen = ImageDataGenerator(
        featurewise_center=False,  # set input mean to 0 over the dataset
        samplewise_center=False,  # set each sample mean to 0
        featurewise_std_normalization=False,  # divide inputs by std of the dataset
        samplewise_std_normalization=False,  # divide each input by its std
        zca_whitening=False,  # apply ZCA whitening
        rotation_range= 10,  # randomly rotate images in the range (degrees, 0 to 180)
        zoom_range = 0.1, # Randomly zoom image
        width_shift_range= 0.1,  # randomly shift images horizontally (fraction of total width)
        height_shift_range= 0.1 ,  # randomly shift images vertically (fraction of total height)
        horizontal_flip=False,  # randomly flip images
        vertical_flip=False)  # randomly flip images

datagen.fit(x_train)
```

Figura 4.26. Fragmento de código explicativo 13. Configuración de la técnica *Data Augmentation*.

Aplicamos este generador sobre el conjunto de datos de entrenamiento para aumentar la cantidad de datos. A veces, esta técnica no es suficiente para acabar con el sobreajuste, ya que no generamos información nueva, sino que modificamos la información existente.

Otra función que se introduce nueva con respecto al entrenamiento del modelo 1 es la función “ReduceLROnPlateau()”, que se aplica en la función “fit()” como llamada (*callback*) (Figura 4.27). Su labor es reducir la tasa de aprendizaje cuando observa esta se estanca.

```
learning_rate_reduction = ReduceLROnPlateau(monitor='val_accuracy',
                                            patience=3,
                                            verbose=1,
                                            factor=0.5,
                                            min_lr=0.00001)
```

Figura 4.27. Fragmento de código explicativo 14. Reducción de tasa de aprendizaje.

La función compile() se mantiene igual que la anterior *convnet*, sin embargo, pasamos de usar el método “fit()” a introducir la función “fit_generator()” para poder aplicar el aumento de datos (generador) y la reducción de tasa de aprendizaje. El primer argumento es un generador que produce de forma indefinida lotes de entradas y objetivos. Como los datos se están generando infinitamente, el modelo necesita saber cuántas muestras sacar del generador antes de declarar que la repetición ha acabado. Esta es la labor del argumento “steps_per_epoch”, es decir, después de haberse ejecutado tantos “steps_per_epochs” establecidos, el proceso pasará a la siguiente repetición. Por último, en este caso, se debe especificar también el argumento “validation_steps” que indica al proceso cuántos lotes sacar del generador de validación para la evaluación (Figura 4.28).

```
epochs = 100
batch_size = 64
history1 = model2.fit_generator(
    datagen.flow(x_train,y_train, batch_size=batch_size),
    steps_per_epoch=x_train.shape[0] // batch_size,
    epochs=epochs,
    validation_data=(x_validate,y_validate),
    validation_steps=x_validate.shape[0] // batch_size
    ,callbacks=[learning_rate_reduction]
)
```

Figura 4.28. Fragmento de código explicativo 15. Fit_generator ()

4.6.3. Red preentrenada

Una red preentrenada es una red que ya ha sido entrenada con anterioridad y ha sido guardada, por lo que puede actuar de forma efectiva como modelo genérico en múltiples problemas. En este caso, vamos a utilizar una *convnet* grande entrenada con el conjunto de datos ImageNet. La red preentrenada elegida es la arquitectura VGG16, sencilla y muy utilizada. A continuación, no solo explicaremos su arquitectura sino la aplicación de dos técnicas de *transfer learning* sobre ella: la extracción de características y el ajuste fino.

4.6.3.1. Transferencia de aprendizaje

La transferencia de aprendizaje o "*transfer learning*" es la técnica aplicada al tomar una red preentrenada para resolver otro problema. Cogemos una red que ya tiene las capas entrenadas y la usamos como base de nuestro modelo. Las primeras capas de esta red ya entrenada reconocen formas simples como líneas y bordes, mientras que las últimas capas ya formas más complejas como patrones que se repiten, de esta forma la podemos usar para clasificar otras imágenes donde pueda reconocer estos patrones aprendidos.

4.6.3.2. VGG16

En primer lugar, debemos saber que las *convnets* que se utilizan para la clasificación de imágenes están formadas por dos partes: una serie de capas de convolución y *pooling* (base convolucional) y un clasificador densamente conectado. La técnica de *transfer learning*

conocida como **extracción de características** consiste en coger la base convolucional de una red preentrenada y entrenar un clasificador nuevo en su salida, de forma que se ejecuten datos nuevos a través de ellos.

En general se suele coger solo la base convolucional y no el clasificador densamente conectado, ya que es la primera la que contiene las representaciones aprendidas de carácter genérico, mientras que las del clasificador son conceptos más específicos y no contiene información de ubicación de objetos. Por tanto, solo nos interesará la base convolucional por ser reutilizable.

- Instancias de la base convolucional VGG16.

```
[ ]  vgg = VGG16(include_top=False, input_shape = (75,100,3), weights='imagenet')
```

Figura 4.29. Fragmento de código explicativo 16. Red preentrenada VGG16.

Los argumentos especificados nos indican lo siguiente:

  - "weights" nos dice el punto de control de los pesos.
  - "include_top" se refiere a si incluir o no el clasificador densamente conectado encima de la red (indicamos "False", ya que vamos a añadirlo y a entrenar el modelo con la base VGG16 y el clasificador adicional).
  - "input_shape" indica la forma de los tensores de entrada de red.

- Arquitectura de la base convolucional VGG16.

Como podemos ver en la Figura 4.30, la red VGG16 está formada por la sucesión de las siguientes capas convolucionales con activación relu, max pooling, densas y clasificador softmax en la última capa clasificadora. Su base convolucional a la cual añadiremos el clasificador acabaría en la última capa de color rojo. Esta imagen no coincide con nuestro modelo ya que las imágenes de entrada son de otra dimensión.

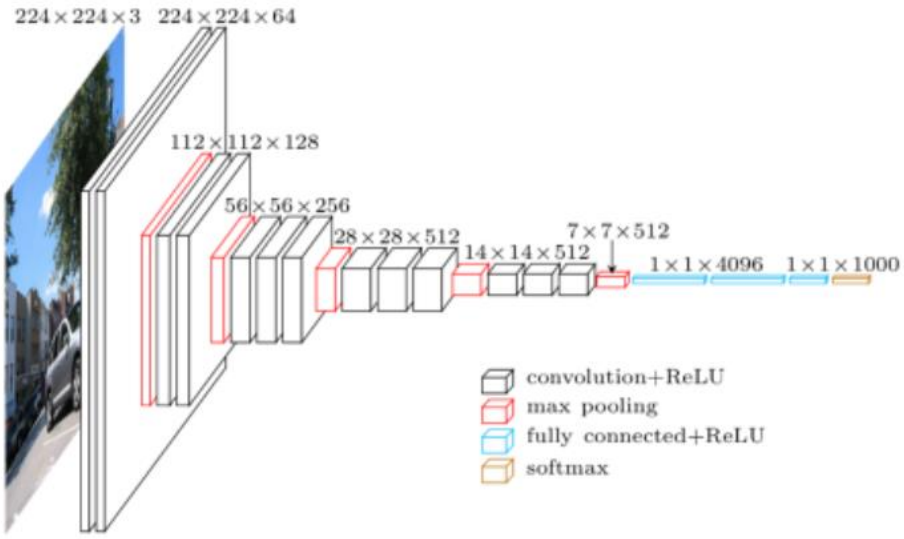


Figura 4.30. Fragmento de código explicativo 17. Arquitectura de la red preentrenada VGG16.

- Seguimiento de cómo cambian las dimensiones de los mapas de características en la sucesión de las capas de VGG16:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 75, 100, 3)        0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 75, 100, 64)       1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 75, 100, 64)       36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 37, 50, 64)        0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 37, 50, 128)       73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 37, 50, 128)       147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 18, 25, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 18, 25, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 18, 25, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 18, 25, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 9, 12, 256)        0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 9, 12, 512)        1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 9, 12, 512)        2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 9, 12, 512)        2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 4, 6, 512)         0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 4, 6, 512)         2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 4, 6, 512)         2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 4, 6, 512)         2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 2, 3, 512)         0
=================================================================
```

Figura 4.31. Fragmento de código explicativo 18. Características de VGG16.

Podemos observar (Figura 4.31) que el mapa de características de salida tiene la forma (2,3,512) Es encima de esta característica donde pondremos el clasificador densamente conectado. Habría varias formas de hacer esto, pero nos hemos decantado por añadir capas Dense encima y ejecutar todo de principio a fin con los datos de entrada.

Esta elección nos permitirá aplicar la técnica de aumento de datos (*Data Augmentation*), aunque sea un proceso computacionalmente costoso y más lento. Por tanto, se recomienda utilizar GPU.

Procedemos a añadir el clasificador densamente conectado encima de la base convolucional:

```
input_shape = (75, 100, 3)
num_classes = 7
model5= Sequential()

model5.add(vgg)

model5.add(Flatten())
model5.add(Dense(256,activation='relu'))
model5.add(Dropout(0.5))
model5.add(Dense(7, activation='softmax'))
```

Figura 4.32. Fragmento de código explicativo 19. Arquitectura del modelo 3.

Cuya sucesión de mapas de características será el siguiente:

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Model)                (None, 2, 3, 512)         14714688
_________________________________________________________________
flatten_3 (Flatten)          (None, 3072)              0
_________________________________________________________________
dense_5 (Dense)              (None, 256)               786688
_________________________________________________________________
dropout_7 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_6 (Dense)              (None, 7)                 1799
=================================================================
```

Figura 4.33. Fragmento de código explicativo 20. Características del modelo 3.

Antes de llevar a cabo el proceso de aprendizaje del modelo es importante congelar la base convolucional para evitar que sus pesos se actualicen durante el entrenamiento y que

no se modifiquen las representaciones aprendidas con anterioridad por esta red. Esto es posible con la siguiente línea de código:

```
[ ]  vgg.trainable = False
```

Figura 4.34. Fragmento de código explicativo 21. Congelación de la base convolucional de VGG16.

Así solamente se entrenarán los pesos de las capas Dense añadidas.

A continuación, compilamos el modelo, aplicamos la técnica aumento de datos (*Data Augmentation*), la reducción de tasa de aprendizaje (ReduceLROnPlateau) y ejecutamos el método fit() exactamente como se hizo en el modelo anterior.

Ahora pasaremos a ejecutar la siguiente técnica complementaria a la extracción de características: el **ajuste fino**. Este consiste en descongelar algunas capas de la base convolucional congelada en la extracción de características y entrenar el clasificador densamente conectado y estas capas descongeladas del modelo. Esto solo sería posible, o más bien recomendable, en las capas superiores de la capa convolucional, ya que resulta más útil hacer el ajuste fino de las capas más especializadas y con menos parámetros. Por tanto, se aplicará ajuste fino en las dos y tres capas superiores de la base convolucional, que como podemos ver en la arquitectura de la red VGG16 (Figura 4.31) será hasta el "block5_conv2" y "block5_conv1" respectivamente.

Por tanto, lo único que tendremos que cambiar con respecto a la extracción de características es que, si queremos descongelar las 3 capas superiores de la base convolucional ejecutaremos la siguiente línea de código, la cual descongela la base convolucional antes congelada para la extracción de características:

```
[ ]  vgg.trainable = True
```

Figura 4.35. Fragmento de código explicativo 22. Descongelación de la base convolucional VGG16.

Ahora indicaremos que queremos congelar todas las capas de la base hasta una en concreto, en este caso la capa convolucional 1 del bloque 5, y que partir de esta, las capas descongeladas pasarán a formar parte del entrenamiento del modelo.

```
[ ]  set_trainable = False
     for layer in vgg.layers:
       if layer.name == 'block5_conv1':
         set_trainable = True
       if set_trainable:
         layer.trainable = True
       else:
         layer.trainable = False
```

Figura 4.36. Fragmento explicativo de código 23. Descongelación de las 3 últimas capas de la base VGG16.

Y, por otra parte, si queremos descongelar las últimas dos capas, especificamos que queremos congelar toda la base hasta la capa convolucional 2 del bloque 5:

```
[ ]  set_trainable = False
     for layer in vgg.layers:
       if layer.name == 'block5_conv2':
         set_trainable = True
       if set_trainable:
         layer.trainable = True
       else:
         layer.trainable = False
```

Figura 4.37. Fragmento de código explicativo 24. Descongelación de las 2 últimas capas de la base VGG16.

Una vez que ya se ha explicado cómo se ha construido cada modelo y qué técnicas se han aplicado en cada entrenamiento, pasamos a analizar los resultados obtenidos.

Capítulo 5

Estudio realizado - Discusión

5.1. Introducción

En este capítulo se mostrarán los distintos experimentos realizados con los tres modelos creados. El objetivo de estas pruebas es conseguir unos valores de hiperparámetros y una distribución del *dataset* adecuados que consigan la mejor precisión para dar solución al problema propuesto por la competición de kaggle. Los numerosos entrenamientos fueron ejecutados en una máquina con GPU en la nube para acelerar la velocidad del proceso. (Apartado 4.4.3)

5.2. Marco de evaluación

5.2.1. *Dataset*

El *dataset* utilizado es el proporcionado por el reto kaggle, HAM10000 (Human Against Machine) que cuenta con 10015 imágenes dermatoscópicas en formato .jpg que se utilizarán para el entrenamiento de las *convnets* diseñadas. Como ya sabemos, estas imágenes han sido redimensionadas para que Tensorflow pudiese manipularlas, estableciendo así una dimensión de 75x100 (alto por ancho), tamaño de entrada para cada una de las redes que entrenamos.

Por una parte, se establecieron las divisiones de los conjuntos de entrenamiento (*train*), prueba (*test*) y validación (*validation*) (Capítulo 3.1.1). Una de las cuestiones que se planteó es qué proporción de imágenes en el conjunto de entrenamiento y validación es la mejor, es decir, con qué porcentajes se obtendría una mejor precisión clasificadora. Las diferentes opciones elegidas para las posibles particiones del conjunto proporcionado son las siguientes:

Porcentaje entrenamiento	Porcentaje validación
80% = 8012 imágenes	20% = 2003 imágenes
70% = 7010 imágenes	30% = 3.004 imágenes
55% = 5508 imágenes	45% = 4506 imágenes

Tabla 5.1. Distintas divisiones aplicadas en el entrenamiento de los modelos.

Aparte, los porcentajes establecidos para el conjunto de prueba (*test*) se mantuvieron en todas las pruebas con una partición del 20% del conjunto total.

De esta forma, al estudiar luego todas las posibles combinaciones de valores de hiperparámetros y particiones del dataset, podremos obtener los valores y porcentajes óptimos.

Además de realizar las distintas divisiones en el conjunto total de datos, se especifica la "normalización de los datos" durante el entrenamiento, en especial de los datos de entrenamiento y test, un paso clave en la etapa de procesamiento de datos, ya que se conseguirá mucho más rendimiento. Esta técnica, normalizar los datos de entrada, se usa para que los algoritmos de *machine learning* funcionen mejor, comprimiéndolos en un rango definido, esto es, para que cada píxel sea un número real que esté en un rango 0-1. El tipo de normalización aplicada es el llamado "escalado estándar" que consiste en restar a cada dato la media de la variable y se divide por su desviación típica (Figura 5.1. - (Cendrero, 2018)).

$$X_{normalized} = \frac{X - X_{mean}}{X_{stddev}}$$

Figura 5.1. Fórmula de escalado estándar.

En este caso, normalizamos los valores de las variables x del conjunto de entrenamiento y prueba (Figura 5.2).

```
x_train = np.asarray(x_train_o['image'].tolist())
x_test = np.asarray(x_test_o['image'].tolist())

x_train_mean = np.mean(x_train)
x_train_std = np.std(x_train)

x_test_mean = np.mean(x_test)
x_test_std = np.std(x_test)

x_train = (x_train - x_train_mean)/x_train_std
x_test = (x_test - x_test_mean)/x_test_std
```

Figura 5.2. Fragmento de código explicativo 25. Normalización de los datos de entrada.

Hay que tener cuidado con este tipo de técnicas, ya que una mala aplicación podría arruinar el análisis hecho.

Por otra parte, al tratarse de una clasificación multiclase (7 clases a predecir), debemos codificar estas clases con un vector one-hot. Este vector permite la transformación de las etiquetas de números enteros a un vector binario, mucho más útil a la hora de entrenar una red. Esto es posible gracias a la función “to_categorical()” de keras, cuya función es convertir un vector de clase (enteros) a una matriz de clase binaria (Figura 5.3).

En nuestro caso, aplicamos esta función al vector “y” de los conjuntos *train* y *test* que será convertido en una matriz en la que cada elemento es un vector formado por 7 enteros (número de clases a predecir) cuyos valores están a 0 menos uno de ellos que está a 1. Por ejemplo, la etiqueta con el valor 2 es “bkl” que hace referencia a las lesiones similares a la queratosis benigna y quedaría como 0010000, de esta forma, se asigna una posición concreta del número 1 para cada etiqueta.

```
y_train = to_categorical(y_train_o, num_classes = 7)
y_test = to_categorical(y_test_o, num_classes = 7)
```

Figura 5.3. Fragmento de código explicativo 26. Vectorización one-hot.

Después de realizar estos pasos, separamos el conjunto de validación del conjunto total de entrenamiento para poder ver si el modelo funciona correctamente durante el proceso de entrenamiento y no se produce sobreentrenamiento, lo que ocurre cuando un modelo aprende de memoria los datos de entrenamiento y no es capaz de generalizar nuevos datos que no estén en este conjunto.

5.2.2. Métricas

Los siguientes tres puntos en los que vamos a profundizar a continuación (optimizador, función de pérdida y precisión), se pasan como argumentos de la función compile() que aplicamos a nuestros modelos:

```
[ ]  model1.compile(optimizer= optimizers.adam(lr=0.0001),
                    loss='categorical_crossentropy',
                    metrics=['accuracy'])
```

Figura 5.4. Fragmento de código explicativo 27. Función compile().

Optimizador

El algoritmo de descenso de gradiente utilizado como optimizador en el aprendizaje de nuestros modelos es el algoritmo ADAM (*Algorithm Development And Mining*). Este funciona de forma que adapta el hiperparámetro *learning rate* en función de la distribución de los parámetros (pesos del modelo), de manera que aumentará la tasa de aprendizaje en caso de que los pesos estén muy dispersos.

Es una variación del procedimiento clásico de descenso de gradiente estocástico (SGD). La diferencia es que el descenso de gradiente estocástico mantiene una tasa de aprendizaje única para todas las actualizaciones de peso y esta tasa no cambia durante el proceso de entrenamiento.

Este método mantiene una tasa de aprendizaje para cada peso de la red (parámetro) y se adapta por separado según se va desarrollando el aprendizaje. Calcula las tasas de aprendizaje adaptativo individual para los diferentes parámetros, por eso se dice que es una combinación de otras dos variaciones del descenso de gradiente estocástico: AdaGrad (*Adaptive Gradient Algorithm*) y RMSProp (*Root Mean Square Propagation*).

Precisión o “Accuracy”

La medida de precisión del modelo será, junto a la función de pérdida, necesaria para poder evaluar los distintos modelos. Esta precisión o *accuracy* será la métrica de evaluación para la clasificación y no es más que el porcentaje total de elementos clasificados correctamente.

Una de las desventajas de esta métrica es que puede ocasionar problemas de precisión en caso de que las muestras de cada clase no estén balanceadas, como es el caso, ya que la proporción de imágenes entre unas clases y otras es totalmente desproporcionada.

Función de pérdida

La función de pérdida (o loss) aplicada es la entropía cruzada (*categorical crossentropy*), como ya sabemos, utilizada para mover los valores de los parámetros de la red (pesos) hacia los valores óptimos, es decir, qué tan cerca está la distribución predicha de la distribución real. Un breve ejemplo:

- Distribución real de pertenecer a la clase A es 1.0 y a la clase B es 0.
- Distribución predicha por el algoritmo de pertenecer a la clase A es 0.228 y de pertenecer a la clase B es 0.623.

La labor de la función de pérdida de entropía cruzada es predecir cómo de errónea es la predicción de la herramienta de clasificación en función de las distribuciones reales.

5.3. Ajuste de hiperparámetros

Para poder encontrar la combinación óptima de hiperparámetros en cada una de las redes creadas, se probarán los siguientes valores de épocas (*epochs*), tamaño de lote (*batch size*) y tasa de aprendizaje (*learning rate*):

Batch_size	12, 32, 64
Learning rate	0.01, 0.001, 0.0001
Epochs	50, 100

Tabla 5.2. Distintos valores de hiperparámetros probados en los modelos diseñados.

En cuanto a los valores del tamaño de lote, se utilizarán valores en un rango de 12 a 64**.** Como no hay un valor adecuado del número de épocas, se probará con 50 y 100 para ver dónde converge la red y se demostró, a través de las gráficas de precisión y pérdida obtenidas, que generalmente es a partir de, aproximadamente, las 30-40 épocas cuando sucede esto, ya que es a partir de este número cuando las curvas de precisión y pérdida de los modelos se convierten en una recta constante se estancan. Finalmente, en cuanto a los valores de la tasa de aprendizaje se probará con el valor estándar 0.001, con un valor más pequeño y con otro más grande para ver el efecto de la convergencia de la red.

Además, en la *convnet* compleja entrenada desde cero y en la preentrenada, en las cuales utilizamos la técnica de aumento de datos (*Data Augmentation*), se aplicará sobre las

imágenes originales transformaciones que implicarán rotación, *zoom*, traslaciones verticales y horizontales:

Rotation_range	10
Zoom_range	0.1
Width_shift_range	0.1
Height_shift_range	0.1

Tabla 5.3. Valores de las transformaciones aplicadas en el aumento de datos en los modelos 2 y 3.

Una vez obtenidas las soluciones óptimas de cada red, es decir, la combinación de hiperparámetros y división de *dataset* con el que se alcanza la mejor precisión en el conjunto de validación, probaremos cómo afectan las distintas aplicaciones del aumento de datos y cuál es la mejor.

Finalmente, para cada modelo óptimo visualizaremos el número de predicciones realizadas correctamente, a través de una matriz de confusión, y el número de predicciones realizadas de forma incorrecta, a través de una gráfica.

5.4. Resultados sobre las redes

5.4.1. Resultados sobre la *convnet* entrenada desde cero sin aumento de datos

Para poder observar el comportamiento de la *convnet* 1, se realizaron las siguientes pruebas, con todas las posibles combinaciones de los porcentajes del *dataset* y valores de los hiperparámetros a ajustar. Estos fueron los resultados de *accuracy* obtenidos para los conjuntos de entrenamiento y validación para cada posible opción:

EXP	PROPORCIÓN TRAIN / VAL	LEARNING_RATE	EPOCHS	BATCH_SIZE	ACCURACY TRAIN	ACCURACY VAL
1	80 / 20	0.01	50	12	0.659011	0.673947
2	80 / 20	0.01	50	32	0.659011	0.676856
3	80 / 20	0.01	50	64	0.659011	0.676856
4	80 / 20	0.01	100	12	0.659011	0.676856
5	80 / 20	0.01	100	32	0.659011	0.676856
6	80 / 20	0.01	100	64	0.659011	0.676856
7	80 / 20	0.001	50	12	0.709935	0.716781
8	80 / 20	0.001	50	32	0.714428	0.716781
9	80 / 20	0.001	50	64	0.703944	0.713662
10	80 / 20	0.001	100	12	0.704443	0.705552

11	80 / 20	0.001	100	32	0.708937	0.714910
12	80 / 20	0.001	100	64	0.703944	0.706800
13	**80 / 20**	**0.0001**	**50**	**12**	0.721917	**0.747349**
14	80 / 20	0.0001	50	32	0.727908	0.734248
15	80 / 20	0.0001	50	64	0.732901	0.739239
16	80 / 20	0.0001	100	12	0.729905	0.735496
17	80 / 20	0.0001	100	32	0.717923	0.723019
18	80 / 20	0.0001	100	64	0.725412	0.740487
19	70 / 30	0.01	50	12	0.659011	0.673045
20	70 / 30	0.01	50	32	0.659011	0.673045
21	70 / 30	0.01	50	64	0.659011	0.673045
22	70 / 30	0.01	100	12	0.659011	0.673045
23	70 / 30	0.01	100	32	0.659011	0.673045
24	70 / 30	0.01	100	64	0.659011	0.673045
25	70 / 30	0.001	50	12	0.701448	0.710899
26	70 / 30	0.001	50	32	0.702446	0.709235
27	70 / 30	0.001	50	64	0.703445	0.707571
28	70 / 30	0.001	100	12	0.687469	0.690100
29	70 / 30	0.001	100	32	0.679481	0.689684
30	70 / 30	0.001	100	64	0.695457	0.690932
31	70 / 30	0.0001	50	12	0.725412	0.735441
32	70 / 30	0.0001	50	32	0.720419	0.734609
33	70 / 30	0.0001	50	64	0.732901	0.743760
34	70 / 30	0.0001	100	12	0.724913	0.733777
35	70 / 30	0.0001	100	32	0.724413	0.733777
36	70 / 30	0.0001	100	64	0.721418	0.737521
37	55 / 45	0.01	50	12	0.659011	0.672490
38	55 / 45	0.01	50	32	0.659011	0.672490
39	55 / 45	0.01	50	64	0.659011	0.672490
40	55 / 45	0.01	100	12	0.659011	0.672490
41	55 / 45	0.01	100	32	0.659011	0.672490
42	55 /45	0.01	100	64	0.659011	0.672490
43	55 / 45	0.001	50	12	0.693460	0.697171
44	55 / 45	0.001	50	32	0.691463	0.701331
45	55 / 45	0.001	50	64	0.693460	0.702163
46	55 / 45	0.001	100	12	0.672991	0.680532
47	55 / 45	0.001	100	32	0.675986	0.689129
48	55 / 45	0.001	100	64	0.686470	0.694676
49	55 / 45	0.0001	50	12	0.711932	0.719911
50	55 / 45	0.0001	50	32	0.717424	0.727953
51	55 / 45	0.0001	50	64	0.721418	0.728785
52	55 / 45	0.0001	100	12	0.710934	0.725180
53	55 / 45	0.0001	100	32	0.719920	0.731836
54	55 / 45	0.0001	100	64	0.712431	0.728231

Tabla 5.4. Resultados de precisión obtenidos para cada experimento en el modelo 1.

Nos fijamos en el *accuracy* obtenido para el conjunto validación, ya que como se ha dicho, es el conjunto que nos permite evaluarla. La solución óptima es la prueba realizada número 13, con una división del *dataset* en conjunto de entrenamiento y validación del 80%

y 20% respectivamente, un valor de *learning rate* superior al estándar (0.0001), 50 épocas y un tamaño de lote pequeño (12).

Visualizamos los resultados obtenidos en la siguiente gráfica:

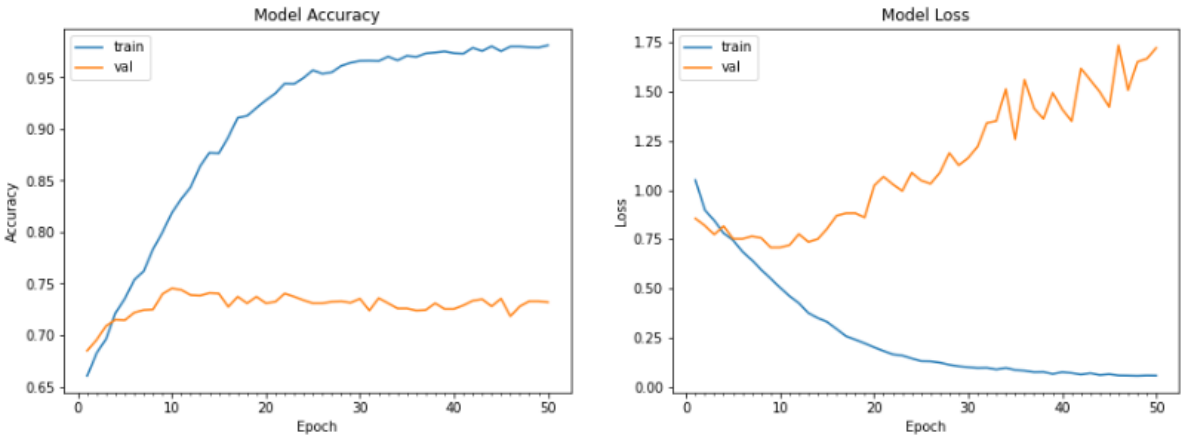


Figura 5.5. Precisión y pérdida de los conjuntos de entrenamiento y validación del modelo 1.

Como podemos ver (Figura 5.5), la exactitud de entrenamiento aumenta de forma lineal con el tiempo, hasta que alcanza casi el 100%, mientras que la exactitud de validación se estanca en el 74-75%. La pérdida de entrenamiento se va reduciendo de forma lineal hasta que alcanza casi el 0, mientras que la de validación alcanza el mínimo en las 10 épocas y luego va aumentando. Estos gráficos son característicos del sobreajuste.

A continuación, visualizaremos el número de predicciones correctas respecto al total, realizadas por el modelo en cuestión, a través de una matriz de confusión:

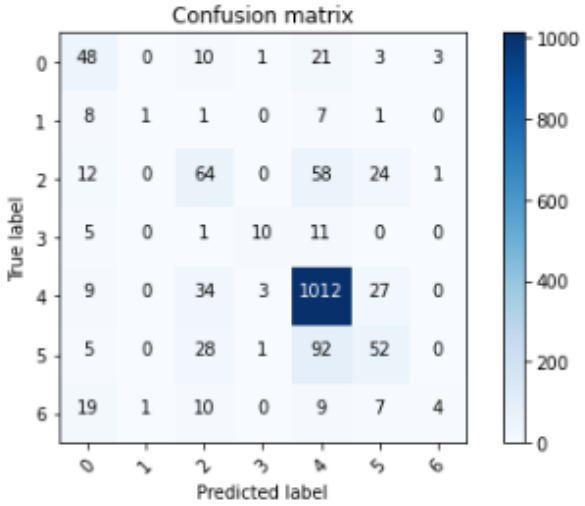


Figura 5.6. Matriz de confusión del modelo1.

Como podemos observar (Figura 5.6), la clase con mejores predicciones es la clase identificada con el número 4, la queratosis actínica.

Y, para terminar, se mostrarán el número de predicciones realizadas incorrectamente por el modelo en cada clase:

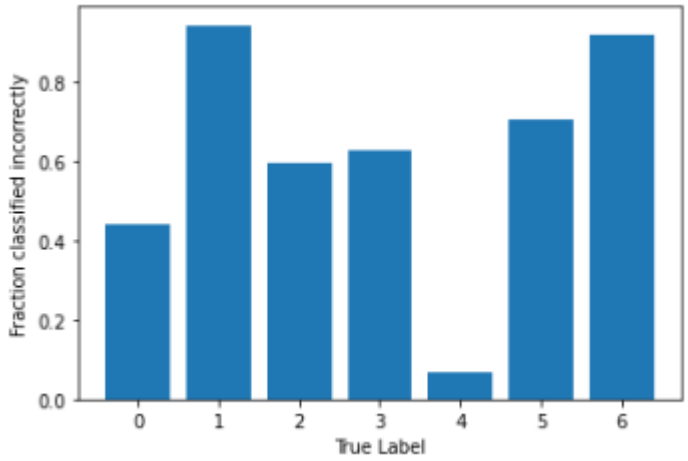


Figura 5.7. Predicciones incorrectas realizadas por el modelo 1 en cada clase.

Podemos ver que, con respecto a las demás clases, es la lesión cutánea representada por el número 1 (Melanoma) la que presenta un mayor número de predicciones incorrectas (Figura 5.7).

5.4.2. Resultados sobre la *convnet* entrenada desde cero con aumento de datos.

En este apartado evaluaremos todas las posibles combinaciones en el modelo número 2, una versión mejorada del anterior, en el que además aplicamos la técnica aumento de datos para combatir el posible sobreajuste creado. Los resultados obtenidos son los siguientes:

EXP	PROPORCIÓN TRAIN / VAL	LEARNING_RATE	EPOCHS	BATCH_SIZE	ACCURACY TRAIN	ACCURACY VAL
1	80 / 20	0.01	50	12	0.659011	0.673947
2	80 / 20	0.01	50	32	0.659011	0.676856
3	80 / 20	0.01	50	64	0.659011	0.676856
4	80 / 20	0.01	100	12	0.659011	0.676856
5	80 / 20	0.01	100	32	0.659011	0.673947
6	80 / 20	0.01	100	64	0.659011	0.673947
7	80 / 20	0.001	50	12	0.739391	0.747973

8	80 / 20	0.001	50	32	0.741887	0.749220
9	80 / 20	0.001	50	64	0.741887	0.745477
10	80 / 20	0.001	100	12	0.744883	0.751716
11	80 / 20	0.001	100	32	0.745881	0.747349
12	80 / 20	0.001	100	64	0.745881	0.744853
13	80 / 20	0.0001	50	12	0.721917	0.740487
14	80 / 20	0.0001	50	32	0.728907	0.742358
15	80 / 20	0.0001	50	64	0.708437	0.716157
16	80 / 20	0.0001	100	12	0.728907	0.754211
17	80 / 20	0.0001	100	32	0.725911	0.740487
18	80 / 20	0.0001	100	64	0.731403	0.751716
19	70 / 30	0.01	50	12	0.659011	0.673045
20	70 / 30	0.01	50	32	0.659011	0.673045
21	70 / 30	0.01	50	64	0.659011	0.673045
22	70 / 30	0.01	100	12	0.659011	0.673045
23	70 / 30	0.01	100	32	0.659011	0.673045
24	70 / 30	0.01	100	64	0.659011	0.673045
25	70 / 30	0.001	50	12	0.659011	0.673045
26	70 / 30	0.001	50	32	0.659011	0.673045
27	70 / 30	0.001	50	64	0.659011	0.673045
28	70 / 30	0.001	100	12	0.659011	0.673045
29	70 / 30	0.001	100	32	0.659011	0.673045
30	70 / 30	0.001	100	64	0.659011	0.673045
31	70 / 30	0.0001	50	12	0.713929	0.736689
32	70 / 30	0.0001	50	32	0.722416	0.750000
33	70 / 30	0.0001	50	64	0.716925	0.739185
34	70 / 30	0.0001	100	12	0.700449	0.713394
35	70 / 30	0.0001	100	32	0.679980	0.700915
36	70 / 30	0.0001	100	64	0.664004	0.681364
37	55 / 45	0.01	50	12	0.659011	0.672490
38	55 / 45	0.01	50	32	0.659011	0.672490
39	55 / 45	0.01	50	64	0.659011	0.672490
40	55 / 45	0.01	100	12	0.659011	0.672490
41	55 / 45	0.01	100	32	0.659011	0.672490
42	55 / 45	0.01	100	64	0.659011	0.672490
43	55 / 45	0.001	50	12	0.752371	0.751525
44	55 / 45	0.001	50	32	0.756365	0.755962
45	55 / 45	0.001	50	64	0.756865	0.757626
46	55 / 45	0.001	100	12	0.751373	0.757626
47	55 / 45	0.001	100	32	0.753869	0.759567
48	55 / 45	0.001	100	64	0.754868	**0.760677**
49	55 / 45	0.0001	50	12	0.730904	0.746811
50	55 / 45	0.0001	50	32	0.726410	0.744870
51	55 / 45	0.0001	50	64	0.722416	0.739046
52	55 / 45	0.0001	100	12	0.718922	0.731836
53	55 / 45	0.0001	100	32	0.730404	0.745979
54	55 / 45	0.0001	100	64	0.728407	0.744870

Tabla 5.5. Resultados de precisión obtenidos para cada experimento en el modelo 2.

La mejor opción se alcanza con una partición del *dataset* del 55% conjunto de entrenamiento y 45% conjunto de validación, un valor de *learning rate* estándar (0.001), 100 épocas y un tamaño de lote grande (64).

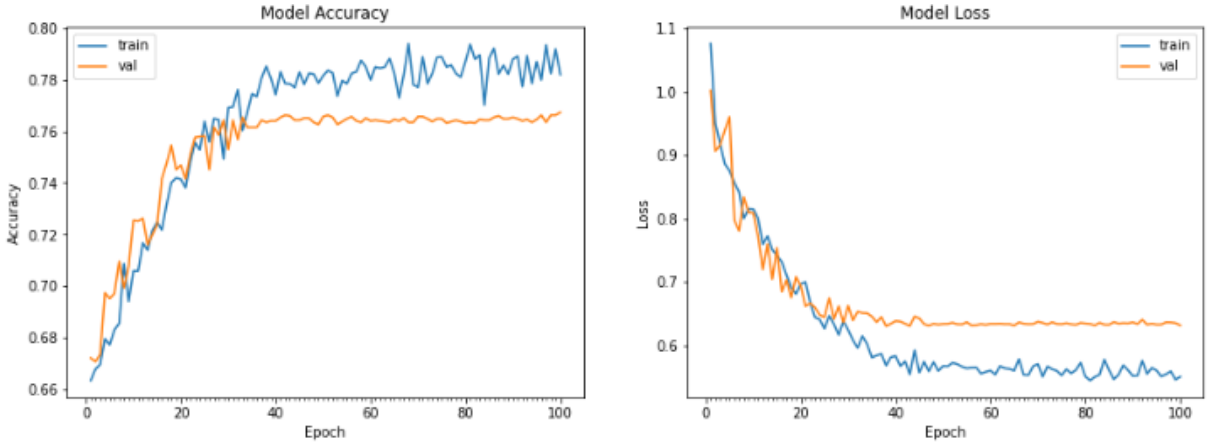


Figura 5.8. Precisión y pérdida del conjunto de entrenamiento y validación del modelo 2.

En este caso (Figura 5.8), obtenemos una precisión del 76-77 % y se ha combatido el sobreajuste hasta aproximadamente las 40-50 épocas ya que, a partir de ahí, el *accuracy* de validación y entrenamiento se estancan, comienza a sobreajustar el modelo. Lo mismo ocurre con las pérdidas que se reducen hasta pasadas 40 épocas y luego se estancan.

A continuación, vamos a ver cómo afectan las distintas transformaciones y sus combinaciones aplicadas en el aumento da datos en esta versión óptima del modelo, obtenida al realizar las distintas pruebas.

	Rotation	**Zoom**	**Width**	**height**	**Accuracy Train**	**Acuuracy Val**
1	NO	NO	NO	NO	0.743884	0.747643
2	SI	NO	NO	NO	0.741388	0.751525
3	SI	SI	NO	NO	0.741388	0.749307
4	SI	SI	SI	NO	0.741388	0.752357
5	SI	SI	SI	SI	0.754868	**0.760677**

Tabla 5.6. Resultados de precisión obtenidos en la mejor versión del modelo 2 aplicando distintas transformaciones de aumento de datos.

Veamos las gráficas de precisión y pérdida obtenidas para cada uno de los 5 casos:

1

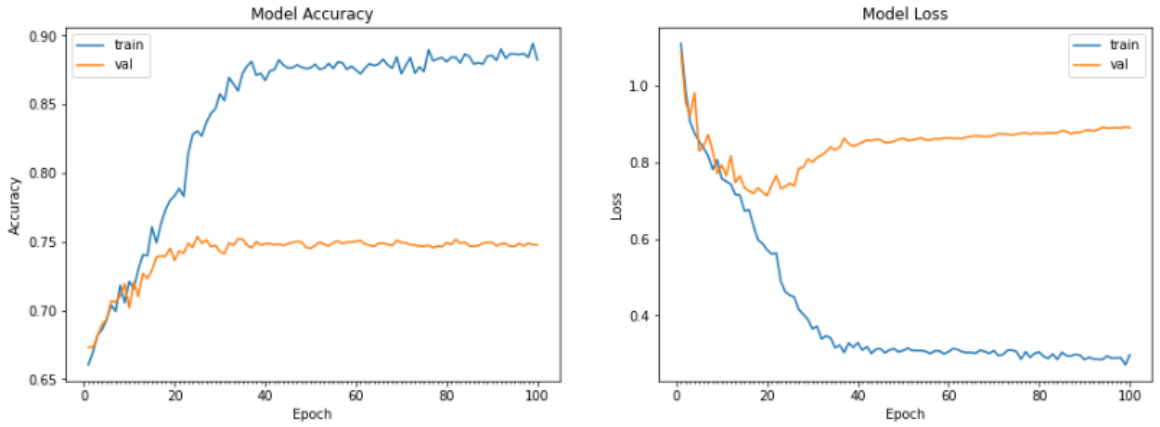


Figura 5.9. Gráfica de precisión y pérdida de entrenamiento y validación del modelo 2 sin aplicar ninguna transformación de *Data Augmentation*.

2

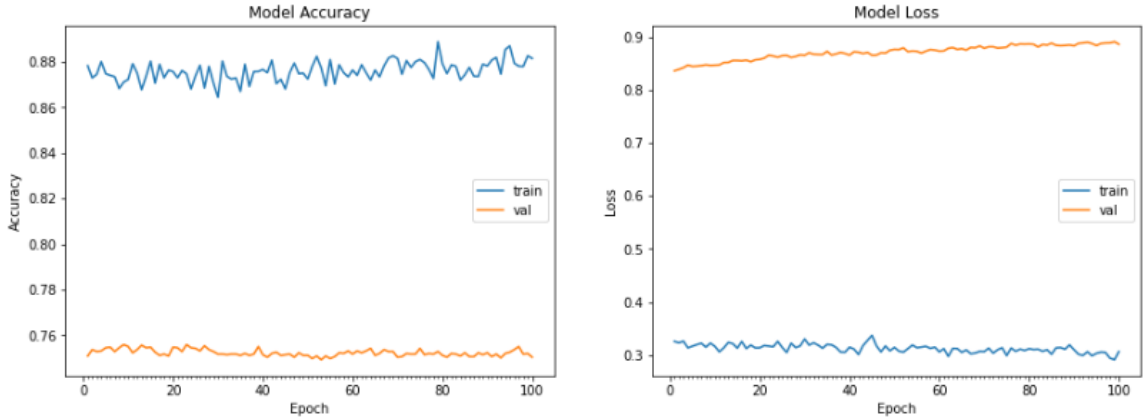


Figura 5.10. Gráfica de precisión y pérdida de entrenamiento y validación del modelo 2 aplicando la transformación de rotación establecida por la técnica de *Data Augmentation*.

3

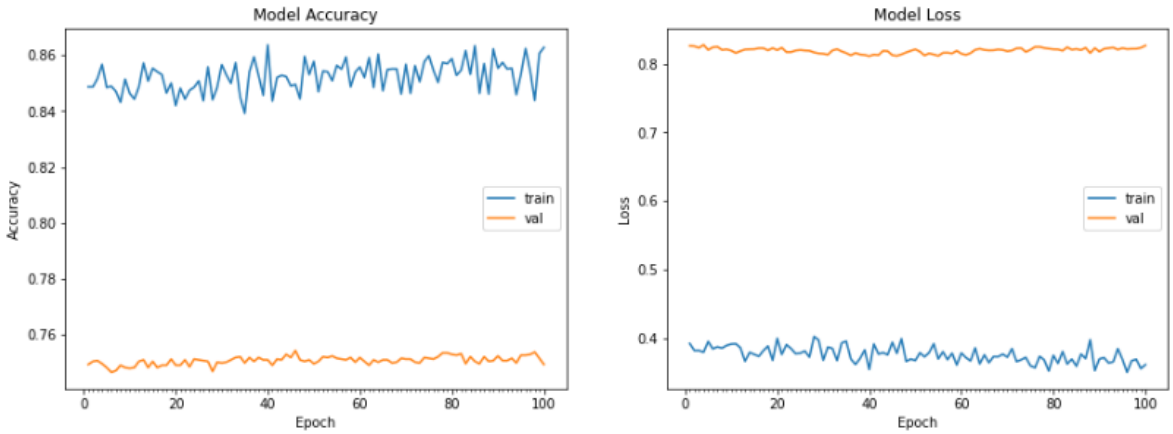


Figura 5.11. Gráfica de precisión y pérdida de entrenamiento y validación del modelo 2 aplicando las transformaciones de rotación y zoom establecidas por la técnica de *Data Augmentation.*

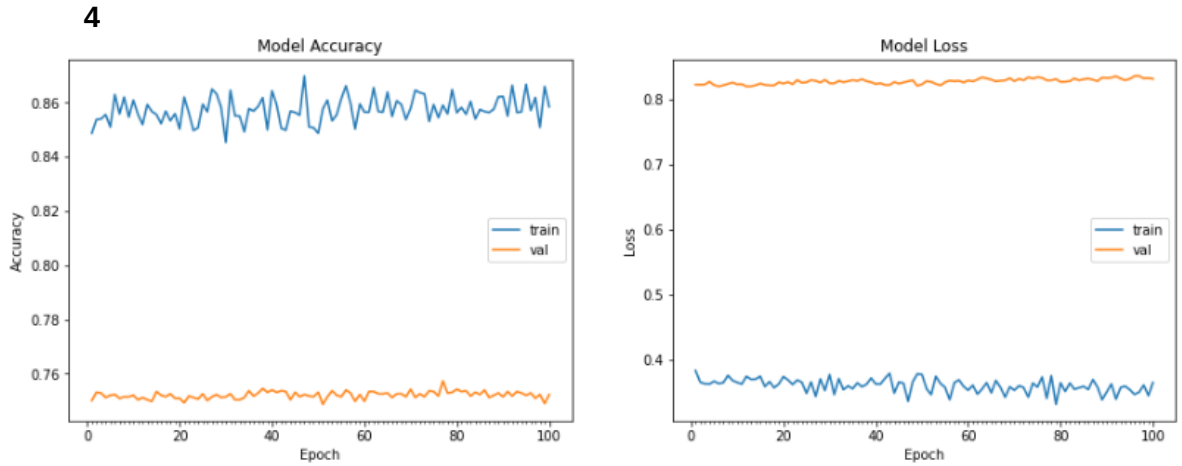


Figura 5.12. Gráfica de precisión y pérdida de entrenamiento y validación del modelo 2 aplicando las transformaciones de rotación, zoom y traslación en anchura establecidas por la técnica de *Data Augmentation*

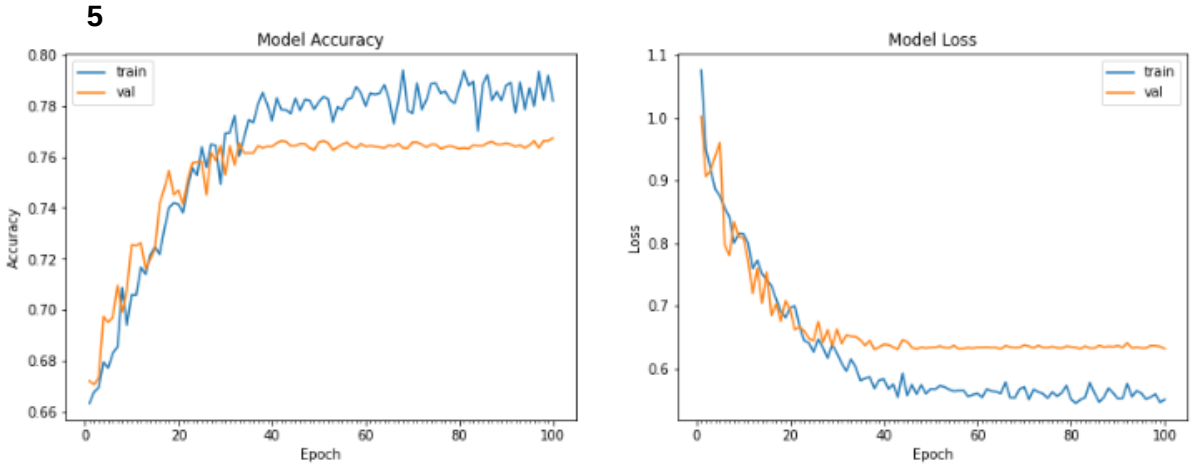


Figura 5.13. Gráfica de precisión y pérdida de entrenamiento y validación del modelo 2 aplicando las cuatro transformaciones establecidas por la técnica de *Data Augmentation*

Las gráficas obtenidas para las 5 posibilidades nos muestran que, el modelo óptimo se consigue aplicando las cuatro transformaciones que hemos establecido en el aumento de datos (Figura 5.13).

A continuación, vamos a ver el número de predicciones realizadas correctamente por este modelo a través de su matriz de confusión:

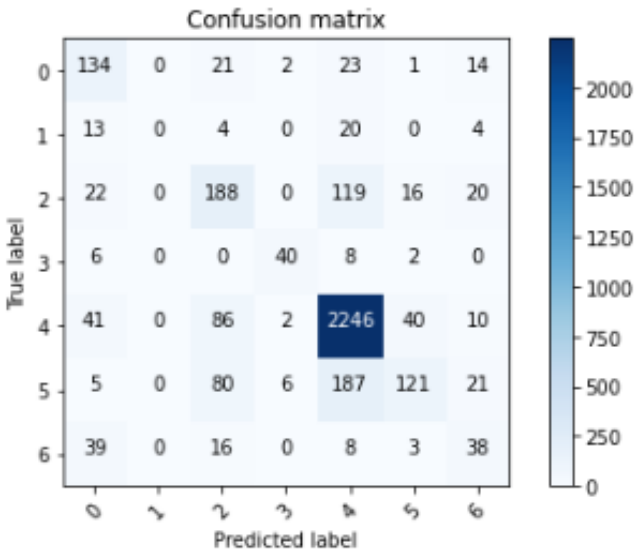


Figura 5.14. Matriz de confusión del modelo 2.

Podemos observar que, como en el modelo anterior, la lesión cutánea de queratosis actínica es la que presentar un mayor número de predicciones correctas (Figura 5.14).

Veamos la proporción de predicciones incorrectas:

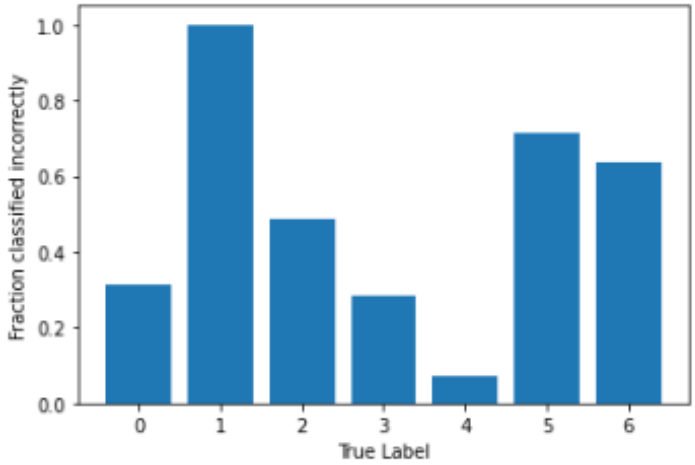


Figura 5.15. Predicciones incorrectas del modelo 2.

Claramente, como el anterior modelo, la clase con mayor número de predicciones incorrectas es la de melanoma (Figura 5.15). Esto puede ser porque es la clase con mayor número de muestras

5.4.3. Resultados sobre VGG16

A continuación, vamos a utilizar la red preentrenada VGG16 como base convolucional con un clasificador densamente conectado. Estos fueron los resultados obtenidos para la técnica de **extracción de características**:

EXP	PROPORCIÓN TRAIN / VAL	LEARNING_RATE	EPOCHS	BATCH_SIZE	ACCURACY TRAIN	ACCURACY VAL
1	80 / 20	0.01	50	12	0.659011	0.673947
2	80 / 20	0.01	50	32	0.748877	0.747973
3	80 / 20	0.01	50	64	0.752371	0.767311
4	80 / 20	0.01	100	12	0.750374	0.763568
5	80 / 20	0.01	100	32	0.751373	0.764816
6	80 / 20	0.01	100	64	0.752871	0.764816
7	80 / 20	0.001	50	12	0.754868	0.766064
8	80 / 20	0.001	50	32	0.754868	0.763568
9	80 / 20	0.001	50	64	0.754868	0.765440
10	80 / 20	0.001	100	12	0.755367	0.763568
11	80 / 20	0.001	100	32	0.755367	0.762321
12	80 / 20	0.001	100	64	0.756865	0.762321
13	80 / 20	0.0001	50	12	0.755866	0.761073
14	80 / 20	0.0001	50	32	0.761358	0.761697
15	80 / 20	0.0001	50	64	0.755367	0.761073
16	80 / 20	0.0001	100	12	0.759361	0.763568
17	80 / 20	0.0001	100	32	0.759361	0.761073
18	80 / 20	0.0001	100	64	0.758862	0.762944
19	70 / 30	0.01	50	12	0.658013	0.675125
20	70 / 30	0.01	50	32	0.658013	0.674709
21	70 / 30	0.01	50	64	0.658013	0.674709
22	70 / 30	0.01	100	12	0.667998	0.673461
23	70 / 30	0.01	100	32	0.679980	0.681364
24	70 / 30	0.01	100	64	0.687469	0.688020
25	70 / 30	0.001	50	12	0.771343	0.774126
26	70 / 30	0.001	50	32	0.773839	0.776622
27	70 / 30	0.001	50	64	0.772841	0.777870
28	70 / 30	0.001	100	12	0.771343	0.781614
29	70 / 30	0.001	100	32	0.775337	0.781614
30	70 / 30	0.001	100	64	0.773839	0.784110
31	70 / 30	0.0001	50	12	0.771343	0.777870
32	70 / 30	0.0001	50	32	0.772341	0.777454
33	70 / 30	0.0001	50	64	0.773340	0.777870
34	70 / 30	0.0001	100	12	0.777833	0.783694
35	70 / 30	0.0001	100	32	0.779331	0.784942
36	70 / 30	0.0001	100	64	0.776836	**0.786190**
37	55 / 45	0.01	50	12	0.728907	0.734886
38	55 / 45	0.01	50	32	0.732401	0.736273
39	55 / 45	0.01	50	64	0.731902	0.736273

40	55 / 45	0.01	100	12	0.658013	0.673600
41	55 / 45	0.01	100	32	0.658014	0.673322
42	55 / 45	0.01	100	64	0.659110	0.673600
43	55 / 45	0.001	50	12	0.744383	0.748475
44	55 / 45	0.001	50	32	0.742886	0.748752
45	55 / 45	0.001	50	64	0.744883	0.748475
46	55 / 45	0.001	100	12	0.724913	0.735996
47	55 / 45	0.001	100	32	0.725412	0.736273
48	55 / 45	0.001	100	64	0.724413	0.737382
49	55 / 45	0.0001	50	12	0.745382	0.748475
50	55 / 45	0.0001	50	32	0.744383	0.750416
51	55 / 45	0.0001	50	64	0.745382	0.751248
52	55 / 45	0.0001	100	12	0.723415	0.737105
53	55 / 45	0.0001	100	32	0.761358	0.764836
54	55 / 45	0.0001	100	64	0.761857	0.767055

Tabla 5.7. Resultados de precisión obtenidos en los distintos experimentos realizados en el modelo 3.

Las distintas pruebas se realizaron relativamente rápido, ya que solo tenemos que tratar dos capas Dense. El mejor resultado de precisión en el conjunto de validación fue la prueba número 36, caracterizada por una división del *dataset* del 70% de entrenamiento y 30% validación, un valor de *learning rate* mayor al estándar, 100 épocas y un tamaño de lote grande. Visualizamos el resultado a través de la siguiente gráfica:

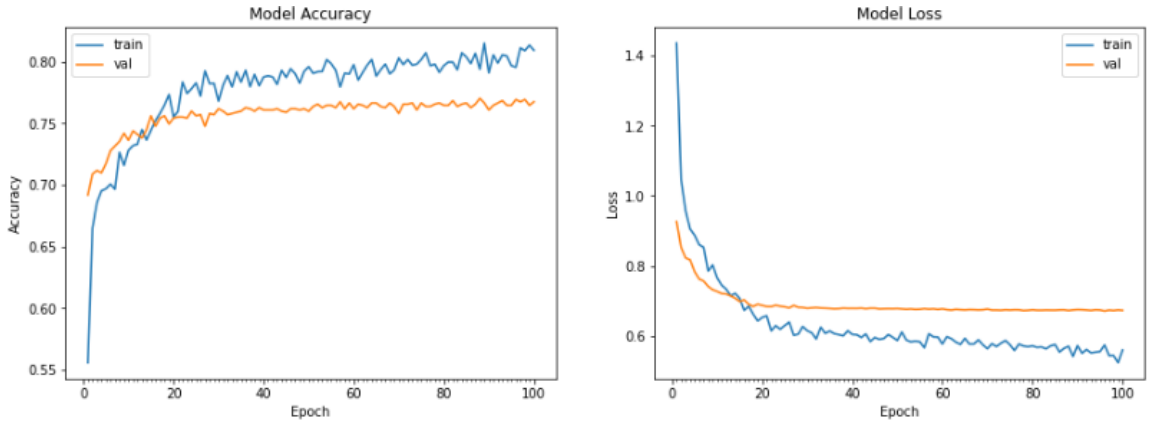


Figura 5.16. Gráfica de precisión y pérdida de entrenamiento y validación del modelo 3 aplicando extracción de características.

La gráfica obtenida (Figura 5.16) nos indica que se consigue una precisión de, aproximadamente, el 77-78% y que es a partir de las 30 épocas más o menos cuando la precisión se estanca y el modelo comienza a sobreajustar. Lo mismo ocurre con la pérdida, donde observamos una pérdida continua hasta que pasadas 30 épocas se estanca.

Probamos el efecto de las distintas transformaciones de la técnica de aumento de datos sobre la versión óptima obtenida con la extracción de características:

	Rotation	Zoom	Width	height	Accuracy Train	Acuuracy Val
1	NO	NO	NO	NO	0.757863	0.775790
2	SI	NO	NO	NO	0.755367	0.779118
3	SI	SI	NO	NO	0.764853	0.782030
4	SI	SI	SI	NO	0.765352	0.782030
5	SI	SI	SI	SI	0.776836	**0.786190**

Tabla 5.8. Resultados de precisión obtenidos al aplicar distintas transformaciones de aumento de datos sobre la versión óptima del modelo 3.

Veamos los efectos de cada posible combinación establecidas na Tabla 5.8:

1.

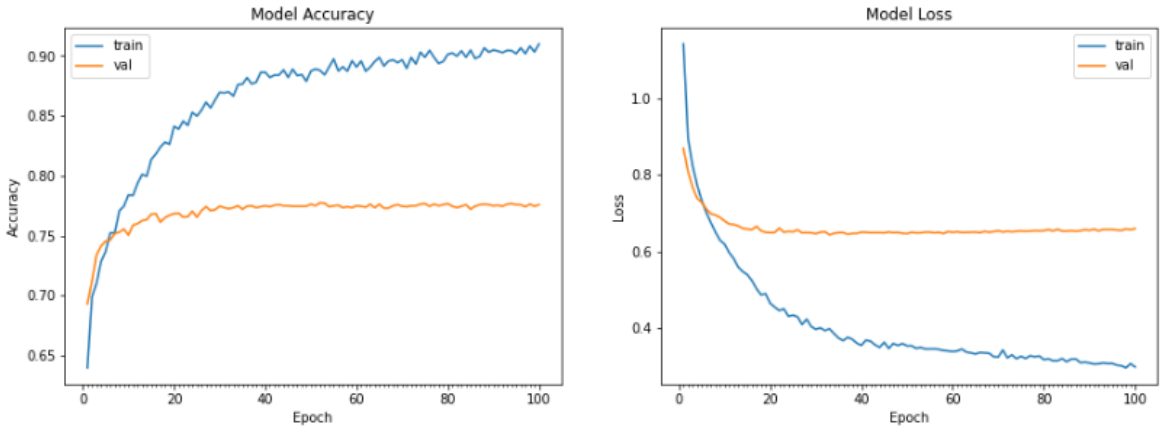


Figura 5.17. Gráfica de precisión y pérdida de entrenamiento y validación del modelo 3 sin aplicar ninguna transformación de *Data Augmentation*

2.

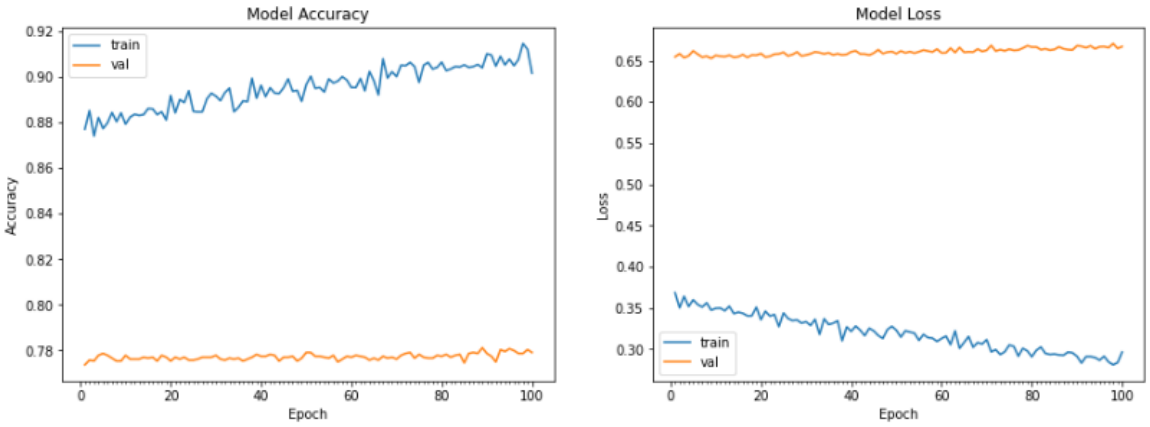


Figura 5.18. Gráfica de precisión y pérdida de entrenamiento y validación del modelo 3 aplicando la transformación de rotación establecida por la técnica de *Data Augmentation*

3.

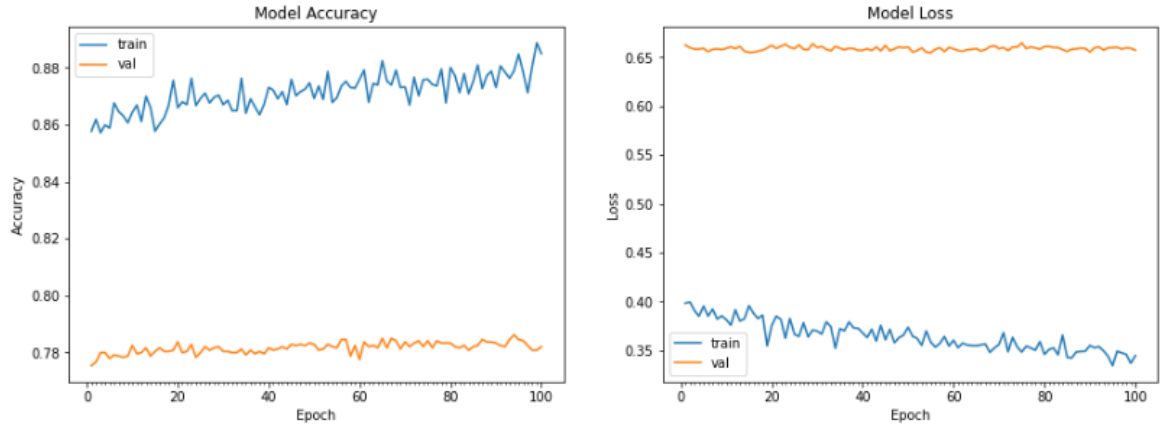


Figura 5.19. Gráfica de precisión y pérdida de entrenamiento y validación del modelo 3 aplicando las transformaciones de rotación y zoom establecidas por la técnica de *Data Augmentation.*

4.

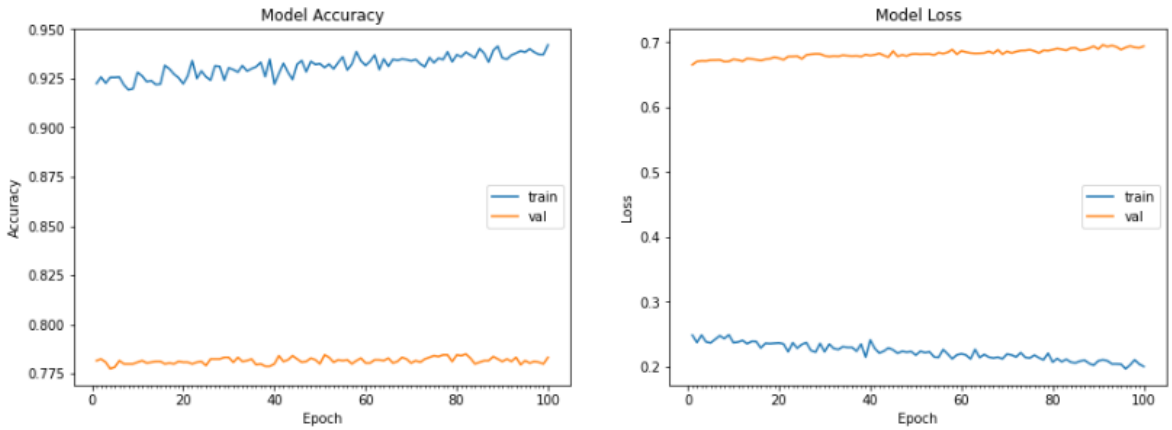


Figura 5.20. Gráfica de precisión y pérdida de entrenamiento y validación del modelo 3 aplicando las transformaciones de rotación, zoom y traslación en anchura establecidas por la técnica de *Data Augmentation*

5.

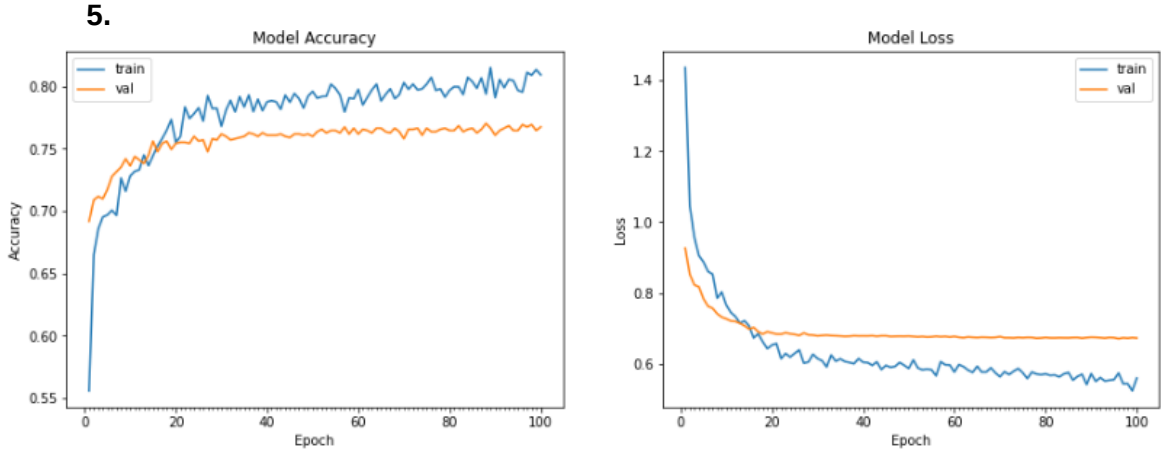


Figura 5.21. Gráfica de precisión y pérdida de entrenamiento y validación del modelo 3 aplicando las cuatro transformaciones establecidas por la técnica de *Data Augmentation*.

Vemos que realmente, la mejor versión del modelo se consigue aplicando los cuatro filtros seleccionados en la técnica de aumento de datos (Figura 5.21).

Veamos el número de predicciones correctas realizas por el modelo 3 usando extracción de características a través de una matriz de confusión:

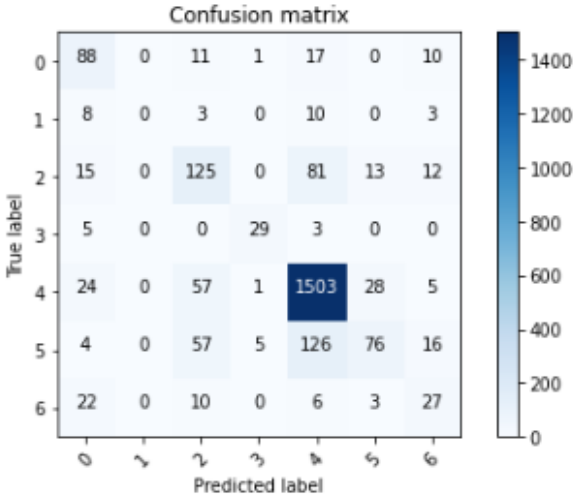


Figura 5.22. Matriz de confusión del modelo 3 con extracción de características.

Como en los entrenamientos anteriores, se obtiene el mayor número de predicciones correctas en la lesión cutánea representada por el número 4 (queratosis actínica).

Veamos las predicciones erróneas de esta red:

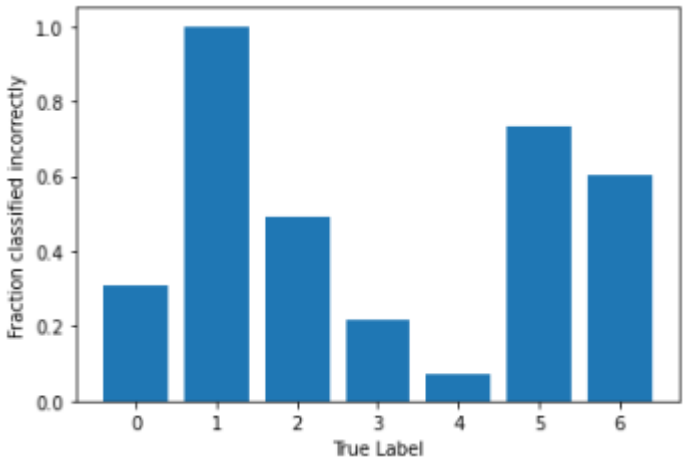


Figura 5.23. Número de predicciones erróneas por clase en el entrenamiento del modelo 3.

Seguidamente, vamos a comentar los resultados obtenidos para la técnica de **ajuste fino**. Este método solamente fue aplicado sobre la versión óptima conseguida con la técnica anterior de extracción de características.

Primero, se hizo la prueba descongelando las últimas tres capas de la base convolucional y se produjo lo siguiente:

Accuracy train	0.814060
Accuracy validation	0.792811

Tabla 5.9. Precisión obtenida al aplicar ajuste fino en las tres últimas capas de la red VGG16.

Visualicemos los resultados obtenidos en una gráfica:

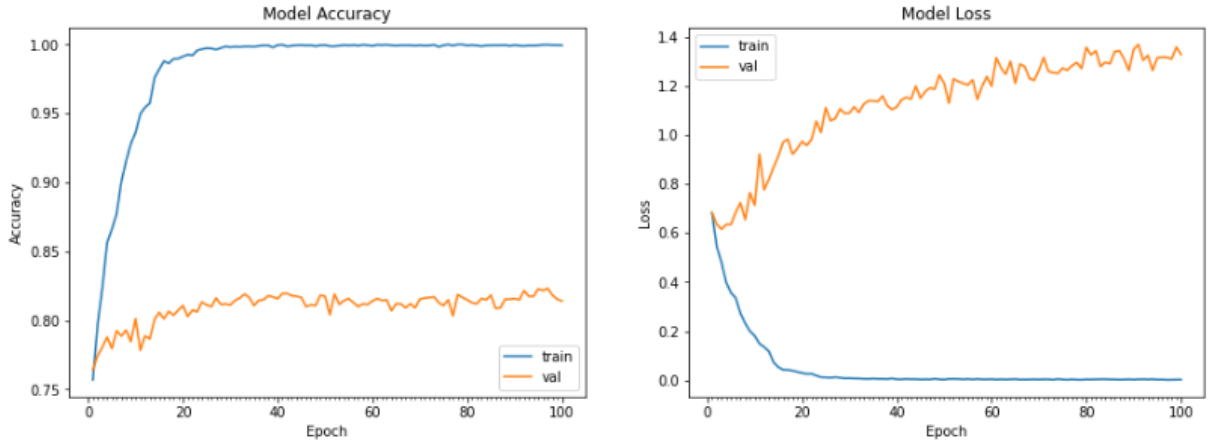


Figura 5.24. Gráfica de precisión y pérdida de entrenamiento y validación del modelo 3 aplicando ajuste fino en las últimas 3 capas de la red preentrenada VGG16.

La curva de exactitud de validación se ve mucho más limpia (Figura 5.24), viendo una mejora absoluta de la exactitud que alcanza el 79-80%. Sin embargo, la curva de pérdida no muestra ninguna mejora real. La respuesta es sencilla: lo que se muestra es una media de valores de pérdida punta a punto y lo que importa para la exactitud es la distribución de los valores de pérdida, no su media

El número de predicciones correctas obtenido con esta técnica fue el siguiente:

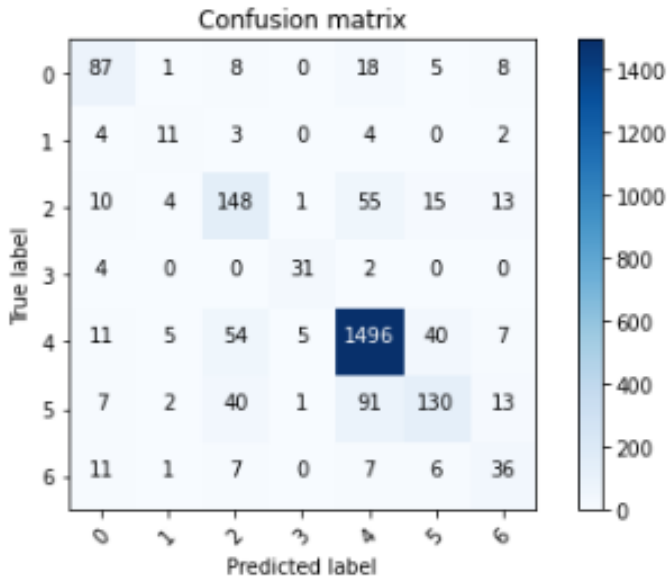


Figura 5.25. Matriz de confusión obtenida con el entrenamiento del modelo 3 aplicando ajuste fino en las 3 últimas capas de VGG16.

La matriz obtenida es similar a las anteriores, donde el mayor número de predicciones correctas es obtenido en la clase 4, queratosis actínica.

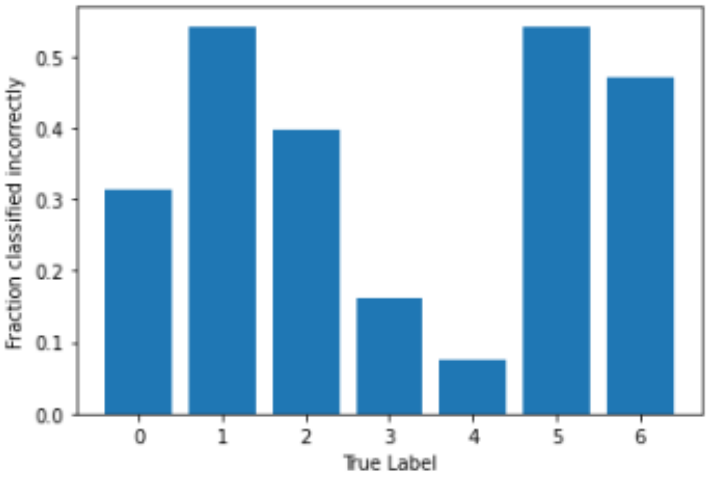


Figura 5.26. Gráfica con el número de predicciones incorrectas realizadas por el modelo 3 con ajuste fino en las 3 últimas capas de la base VGG16.

Se obtiene el mayor número de predicciones incorrectas para los tipos de lesión cutánea “Melanoma” y “Lesiones vasculares”.

En segundo lugar, descongelamos las últimas dos capas y obtuvimos los siguientes resultados:

Accuracy train	0.786321
Accuracy validation	0.815724

Tabla 5.10. Precisión obtenida al aplicar ajuste fino en las dos últimas capas de VGG16.

Visualicemos los resultados conseguidos en las siguientes gráficas:

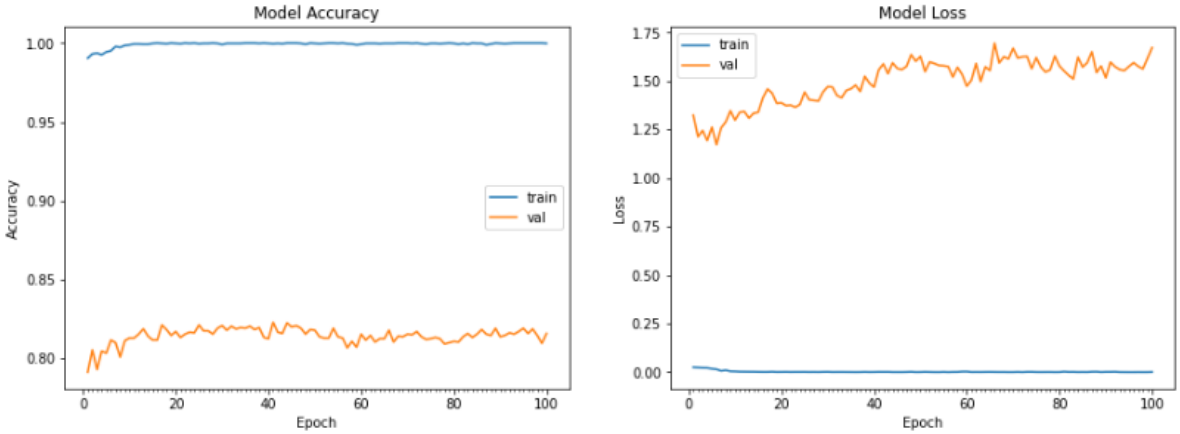


Figura 5.27. Gráfica de precisión y pérdida de entrenamiento y validación obtenida en el modelo 3 al aplicar ajuste fino en las 2 últimas capas de VGG16.

Conseguimos una precisión similar al anterior ajuste fino, un poco mejorada, aproximadamente de un 81-82%. En cuanto a la curva de pérdida no muestra ninguna mejora real como en el caso anterior.

Finalmente, observamos el número de predicciones correctas realizadas por este modelo:

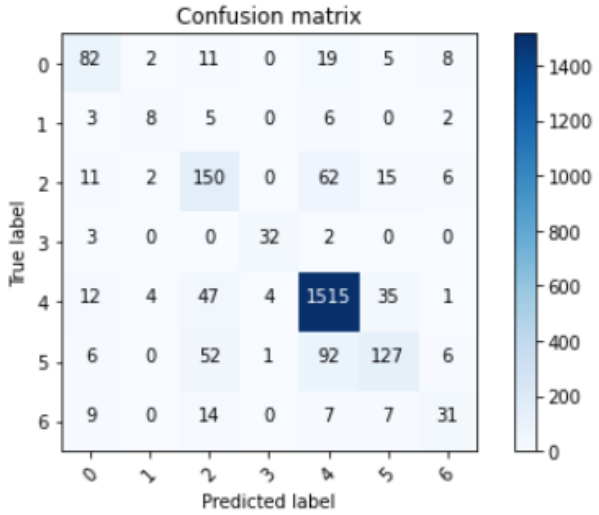


Figura 5.28: Matriz de confusión del modelo 3 con ajuste fino en las 2 últimas capas de VGG16.

El efecto conseguido es similar a los anteriores modelos.

Finalmente, veamos cuántas predicciones erróneas ejecutó este último modelo:

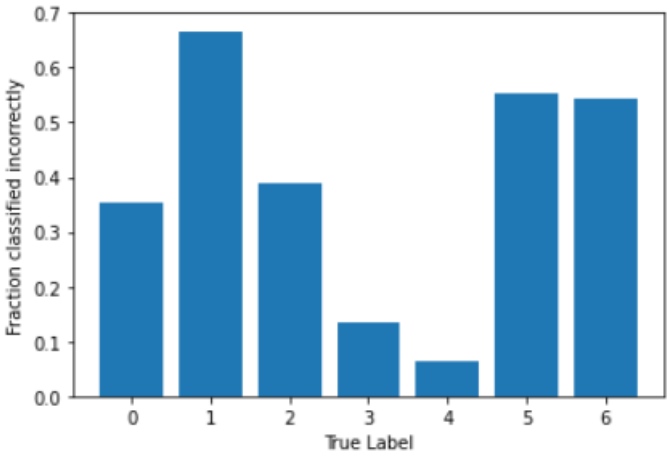


Figura 5.29. Número de predicciones erróneas realizadas por el modelo 3 con ajuste fino en las 2 últimas capas de la base VGG16.

Una vez más, el mayor número de predicciones erróneas se obtienen en la lesión cutánea “Melanoma”.

5.5. Comparativa de resultados

Comparando los resultados obtenidos de los experimentos realizados sobre los 3 modelos, podemos concluir lo siguiente:

- En las *convnets* entrenadas desde cero, la división del *dataset* con la que se consigue la mejor precisión es 55% conjunto de entrenamiento y 45% validación. Esto puede ser porque se produce sobreajuste pasadas las 40-50 épocas, es decir, los datos de entrenamiento son aprendidos de memoria y el modelo obtiene peores resultados al aplicar el conjunto de validación porque son datos distintos a los aprendidos.
- El funcionamiento óptimo de la técnica de aumento de datos se consigue aplicando las 4 transformaciones escogidas.
- Los mejores resultados globales, como era de esperar, se consiguieron aplicando la técnica de ajuste fino al descongelar únicamente las dos últimas capas de la base convolucional de la red preentrenada.
- Todos los modelos presentan el mayor número de predicciones realizadas correctamente para el tipo de lesión cutánea Queratosis Actínica.

- Todos los modelos obtienen el mayor número de predicciones incorrectas para la lesión cutánea Melanoma. Esto se puede deber a que esta clase presenta muchas más muestras en comparación a las demás.

Capítulo 6

Conclusiones y trabajo futuro

6.1. Conclusiones

A lo largo de este proyecto presentado para solventar el problema propuesto por el reto kaggle de crear una herramienta que clasifique imágenes dermatoscópicas en 7 tipos de lesiones cutáneas, se han llevado a cabo diversas labores y se ha intentado acabar con todas las adversidades que han ido apareciendo a lo largo del proceso de desarrollo.

Se ha de destacar la ayuda proporcionada por los cursos, tutoriales y notebooks completados para adentrarnos en este mundo del *deep learning*, así como del estudio realizado en los apartados 2 y 3 del trabajo que hacen referencia a la Introducción a la temática y al Estado del arte. Además, se destaca la ayuda facilitada por los notebooks de los participantes de la competición, considerada como ayuda clave para la evolución de este proyecto.

Uno de los principales problemas encontrados, y que sin duda ha llevado más tiempo resolver, ha sido encontrar una herramienta que nos permitiese ejecutar los distintos entrenamientos en GPU, ya que en CPU tardaban horas, incluso días. Como no se disponía de GPU física, como se ha dicho, se efectuó en el entorno GoogleColab con GPU en la nube. Esta herramienta, comparada con las otras posibles soluciones probadas, fue la que permitió realizar los entrenamientos de las redes neuronales un poco más rápido, aunque resaltamos el hecho de que se desconecta en caso de saturación de la nube o por inactividad.

Otro de los inconvenientes que no nos permitió alcanzar la precisión óptima en los modelos creados era la distribución de imágenes de cada clase, proporción nada equitativa entre las siete posibles clases, ya que el tipo de lesión “Melanoma” presentaba una mayor parte de los datos proporcionados, mientras que de las demás clases tenían muy pocos datos relacionados.

En este trabajo se ha conseguido crear y entrenar una *convnet* desde cero aplicando la técnica de aumento de datos con una precisión del 76% y otro modelo en el que se emplea una red preentrenada a la que se le aplica la técnica de extracción de características y la técnica ajuste fino y en el cual se obtiene una precisión del 82%. Los resultados obtenidos son aproximados a los conseguidos por otros participantes del reto que trabajaron con la

herramienta keras, aunque los mejores los alcanzasen los participantes que utilizaron la librería Pytorch.

Los resultados obtenidos son considerados buenos en comparación al resto de participantes que presentaron una posible solución al problema propuesto. Además, como ayuda al personal sanitario para determinar un diagnóstico automatizado de cáncer de piel, un porcentaje de precisión de aproximadamente un 80% está bien, aunque al tratarse de un problema grave, sería mejor obtener una solución óptima con la mejor precisión posible.

Aunque no se consiguiese la totalidad del porcentaje de precisión, o una aproximación, el principal objetivo de este proyecto era aprender a crear y entrenar un modelo y ver cómo a través de añadir capas, aplicar distintas técnicas y ajustar parámetros se conseguía una solución aceptable para el problema planteado. La demostración de que este objetivo fue conseguido con creces está en el proceso de desarrollo del trabajo, donde partimos de un modelo sencillo y vamos aumentando la complejidad y precisión según vamos avanzando en el transcurso.

6.2. Trabajo futuro

En función de los resultados obtenidos en este trabajo y de los distintos problemas que fueron surgiendo a lo largo del proceso, considero que algunas de las posibles mejoras para conseguir modelos con mayor precisión y calidad son las siguientes:

- En cuanto a la división del *dataset*, hay que procurar mantener la misma proporción de datos en los conjuntos de validación y prueba, siempre dejando el mayor porcentaje del *dataset* al conjunto de entrenamiento.

- Otro punto relacionado con el *dataset* es intentar conseguir la misma proporción de datos para cada una de las 7 clases a clasificar, acabando con la falta de equidad entre clases y dándole así una precisión real a la herramienta.

- Jugar con más parámetros de la técnica aumento de datos, ya que solamente se aplicaron 4 de las 10 posibles transformaciones de datos.

- Evaluar la métrica *accuracy* de cada clase por separado, de forma que se pueda ver cual fracasa y cual funciona perfectamente.

- Utilizar otras redes preentrenadas más complejas como, por ejemplo, Inception.

- En el caso de la red preentrenada VGG16, probar a introducir más capas al final de la red, antes del clasificador softmax, para reducir el número de parámetros.

- Probar a descongelar solo la última capa de la red preentrenada VGG16 al aplicar la técnica de ajuste fino.

Bibliografía

Becker, D. (s.f.). *Kaggle*. Obtenido de https://www.kaggle.com/learn/deep-learning

Burgal, J. U. (1 de Agosto de 2018). *En mi local funciona*. Obtenido de https://enmilocalfunciona.io/deep-learning-basico-con-keras-parte-3-vgg/

Calvo, D. (20 de Julio de 2017). *Diego Calvo*. Obtenido de http://www.diegocalvo.es/red-neuronal-convolucional/

Caparrini, F. S. (3 de Diciembre de 2018). *GitHub*. Obtenido de https://github.com/fsancho/DL/blob/master/4.%20Redes%20Convolucionales/4.2%20CNN%20con%20datasets%20peque%C3%B1os.ipynb

Cendrero, S. M. (1 de Noviembre de 2018). *Think Big / Empresas*. Obtenido de https://empresas.blogthinkbig.com/precauciones-la-hora-de-normalizar/

Chollet, F. (2020). *Deep Learning con Python.* ANAYA.

Durán, J. (4 de Septiembre de 2019). *Medium*. Obtenido de https://medium.com/metadatos/todo-lo-que-necesitas-saber-sobre-el-descenso-del-gradiente-aplicado-a-redes-neuronales-19bdbb706a78

EPICALSOFT INSTANCE BLOG. (s.f.). Obtenido de http://epicalsoft.blogspot.com/2019/02/azure-machine-learning-sobreajuste-y.html

franspg. (27 de Enero de 2020). *franspg*. Obtenido de https://franspg.wordpress.com/author/franspg/

Google Developers. (s.f.). Obtenido de https://developers.google.com/machine-learning/crash-course/multi-class-neural-networks/softmax?hl=es-419

kaggle. (s.f.). Obtenido de https://www.kaggle.com/kmader/skin-cancer-mnist-ham10000/kernels

MC.AI. (22 de Junio de 2018). Obtenido de https://mc.ai/convolutional-neural-networks-cnn-a-dummy-overview/

Na8. (29 de Noviembre de 2018). *Aprende Machine Learning*. Obtenido de https://www.aprendemachinelearning.com/como-funcionan-las-convolutional-neural-networks-vision-por-ordenador/

Rodríguez, V. (09 de Noviembre de 2018). *VicentBlog*. Obtenido de https://vincentblog.xyz/posts/dropout-y-batch-normalization

Sanz, O. M. (4 de Junio de 2019). *Adictos al trabajo*. Obtenido de https://www.adictosaltrabajo.com/2019/06/04/google-colab-python-y-machine-learning-en-la-nube/

Serrano, A. G. (21 de Octubre de 2019). *El laberinto de Falken*. Obtenido de https://www.ellaberintodefalken.com/2019/10/vision-artificial-redes-convolucionales-CNN.html

tensorflow.org. (s.f.). Obtenido de https://www.tensorflow.org/tutorials/keras/classification

Torres, J. (2018). *Deep Learning Introducción Práctica con keras (primera parte).* WATCH THIS SPACE.

Zambrano, J. (31 de Marzo de 2018). *medium.com*. Obtenido de https://medium.com/@juanzambrano/aprendizaje-supervisado-o-no-supervisado-39ccf1fd6e7b