ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA

INFORMÁTICA

Trabajo fin de Grado

Grado en Ingeniería de la Salud. Mención Informática clínica

DETECCIÓN DE NEUMONÍA EN IMÁGENES DE PULMÓM MEDIANTE APRENDIZAJE PROFUNDO

Realizado por:
Marta Núñez Cascales

Dirigido por:
Juan Antonio Nepomuceno Chamorro

Departamento:
Lenguaje y Sistemas Informáticos

Sevilla, 10 de junio del 2020 (v.1.0.0)

Agradecimientos

En primer lugar, quería agradecer a mi profesor D. Juan Antonio Nepomuceno Chamorro, no solo por la ayuda, supervisión y apoyo en el trabajo realizado, sino también por su afable docencia a lo largo de mi formación universitaria, la cual me ha permitido conocer y crecer de la forma más bonita en el campo del lenguaje informáticos.

Agradecer a la Universidad de Sevilla y a el colectivo docente y administrativo, por hacer posible mi formación universitaria por medio de su docencia y gestión.

En segundo lugar, quería agradecer a mi familia. A mis padres, porque sin ellos no sería quien soy ni habría llegado hasta el punto donde he llegado, ellos han permitido por medio de su esfuerzo que existiera el camino que yo quería ir tomando en todo momento.

A mi hermana mayor Carlota, por haber sido mi punto de apoyo y la que siempre ha creído en mí. A mis hermanas pequeñas y a mi sobrino por haberme transmitido siempre fuerza y alegría.

Finalmente, agradecer a mis amigos, porque sin ellos esta etapa no la podría recordar con tantos buenos momentos y tantas anécdotas, a pesar de las muchas horas echadas en la biblioteca. Gracias a ellos he tenido un apoyo continuo para lograr mis objetivos.

Especialmente, agradecer a mi amiga Ana Caamaño Cundíns, porque además de demostrar su amistad en estos años de carrera, ha sido mi codo a codo a lo largo de todo el trabajo realizado.

Resumen

El presente estudio designado “Detección de neumonía en imágenes de pulmón mediante Aprendizaje Profundo”, parte del propósito de cooperar a optimizar el problema temporal que presentan los facultativos neumólogos en la detección de patologías como la neumonía en imágenes de pulmón. Para ello hacemos uso del campo de la Inteligencia artificial, mediante la aplicación del modelo computacional perteneciente al Aprendizaje profundo, las redes neuronales convolucionales.

Dado dicho problema, nuestro objetivo es conocer en profundidad el campo del Aprendizaje profundo y desarrollar un modelo computacional de red neuronal convolucional que realice la clasificación de las imágenes de pulmón, en función de su patología, si tiene neumonía o no.

Este documento presenta el estudio realizado a satisfacer los objetivos manifestados. El análisis comienza con un estudio documental exhaustivo del campo, en el que se trabaja, la inteligencia artificial, Machine learning y Deep learning, seguido de una comprensión de las Redes Neuronales y su funcionamiento. Posteriormente, nos centramos en la formación práctica desarrollada en Python, de las Redes Neuronales mediante el uso de la librería Keras en diferentes notebooks, permitiéndonos conocer la implementación del desarrollo de las redes neuronales convolucionales y las diferentes técnicas y herramientas aplicables para obtener un funcionamiento óptimo de ellas. Finalmente, tras ahondar en la temática y tener suficientes conocimientos, se realiza la implementación de un modelo de red neuronal convolucional que postule los objetivos.

Abstract

The current study named as "Detection of pneumonia in lung images by means of Deep Learning" is based on the purpose of cooperating to optimize the temporal problem presented by pulmonologists in the detection of pathologies such as pneumonia in lung images. In order to do so it is most common the use of artificial intelligence, by applying the computational model belonging to deep learning, convolutional neural networks.

Given this problem, the main objective is to study in depth the field of “deep learning” and so, develop a computational model of convolutional neural network that performs the classification of lung images and, depending on their pathology, determine whether they suffer from pneumonia or not.

This document presents the study carried out to satisfy the stated objectives. The analysis begins with an exhaustive documentary study of the field in which we work: artificial intelligence, machine learning and deep learning; followed by an understanding of Neural Networks and their operation. Subsequently the study is focused on the practical training developed in Python of Neural Networks, which we were able to develop through the use of the Keras library in different notebooks, that allowed us to learn about the implementation of the development of convolutional neural networks and the different techniques and tools applicable to obtain an optimal performance. Finally, after undergoing this exhaustive analysis of the subject and having obtained sufficient knowledge, the implementation of a convolutional neural network model, that postulates the objectives, is carried out.

Material de apoyo

En el siguiente enlace podemos encontrar un repositorio GitHub con el conjunto de notebook utilizados y desarrollados para dicho trabajo.

- **Enlace GitHub:**

https://github.com/marnuncas1/Deteccion-de-Neumonia-en-imagenes-de-pulmon-mediante-Aprendizaje-Profundo

- **Código QR:**

Índice general

Índice de cuadros

Índice de figuras

Índice de código

1. Introducción

En el presente documento vamos a desarrollar el estudio en profundidad de un campo dentro del Aprendizaje profundo, las redes neuronales convolucionales. Esta investigación ha sido llevada a cabo para desarrollar una solución al problema de capacitar a una maquina a realizar una clasificación de imágenes de pulmón.

1.1 Guía del documento

El estudio presentado en el documento está basado en un estudio documental y experimental a gran nivel de profundidad y finalmente nuestra aportación para alcanzar los objetivos.

Este estudio previo a nivel documental se encuentra desarrollado en el documento en los capítulos 2 y 3, llamados introducción a la temática y marco teórico, respectivamente, donde hablamos en profundidad del campo en el que se encuentra la temática, su arquitectura y funcionamiento.

Posteriormente continuamos el estudio a nivel experimental, donde encontramos las diferentes implementaciones desarrolladas para el estudio de la temática, este se encuentra desarrollado en el capítulo 4, llamado estudio previo.

El desarrollo de una observación con nuestra aportación a nivel experimental con propósito de solventar el problema y cumplir nuestros objetivos. Este apartado se encuentra en los capítulos 5 llamado Reto Kaggle. En el capítulo 6 llamado Discusiones, podemos encontrar las diferentes dificultades encontradas a la hora de desarrollar el proyecto con posibles caminos a tomar y finalmente el camino escogido.

Finalmente, en el capítulo 7 se encuentran las conclusiones donde recogemos lo sacado en el estudio realizado.

Este estudio experimental ha sido llevado a cabo mediante el uso del lenguaje de programación Python bajo el uso del framework de alto nivel Keras.

1.2 Objetivos.

Como principal objetivo, el estudio en profundidad del campo de las redes neuronales convolucionales, la comprensión de su funcionamiento y el conocimiento necesario para la implementación de un modelo de red neuronal convolucional.

Por otra parte, atendiendo al problema propuesto. El problema por considerar consta de un conjunto de imágenes médicas de pulmón con diferentes patologías. Algunas imágenes presentan la patología de neumonía y otras no.

Queremos facilitar la tarea de detección de patologías, en este caso la enfermedad de neumonía al personal sanitario, dando una segunda opinión mediante el uso del aprendizaje profundo con redes neuronales convolucionales, optimizando así el tiempo de estos facultativos y agilizando el sistema.

Dado este problema, nuestro segundo objetivo es desarrollar un modelo de red neuronal convolucional que realice de forma óptima la correcta clasificación de imágenes de pulmón. Esta clasificación consta en pulmones con la patología neumonía y pulmones sanos, es decir, pulmones que no presentan la enfermedad neumonía.

1.3 Plan de proyecto

Para llevar a cabo el desarrollo del trabajo, necesitábamos una previa organización en la que se detalle los objetivos y el alcance del proyecto, la iteración de tareas a desarrollar para cumplir los objetivos, valoración de la dificultad de cada tarea, una planificación temporal …etc.

Esto es llevado a cabo bajo un enfoque Agile. Agile es una metodología de desarrollo de software basada en procesos iterativos organizadas en pequeñas iteraciones de tiempo. Esta realiza una organización del proyecto de forma flexible, que permite pequeños cambios en función de la evolución del desarrollo del proyecto.

1.3.1 Tareas por realizar

Para comenzar el desarrollo de un proyecto debemos conocemos el problema y definir los objetivos, los cuales acabamos de detallar.

Posteriormente enumeramos y listamos el conjunto de tareas a llevar a cabo para alcanzar nuestros objetivos. Este listado es realizado a alto nivel de profundidad, especificando de forma en que se va a basar el estudio documental y en que cursos y notebook en principio nos vamos a apoyar para el estudio experimental y nuestra posterior aportación.

Debemos conocer el nivel de dificultad que conlleva cada tarea descrita para conocer el esfuerzo y tamaño relativo de cada una y poder establecer así un desarrollo temporal del proyecto ajustado por cada tarea, pudiendo así cumplir nuestros objetivos en un plazo determinado.

Para llevar a cabo la evaluación del nivel de dificultad de cada tarea que forma nuestro trabajo, hicimos uso de una técnica de la metodología Scrum, una metodología Ágil. Esta técnica es basada en la técnica Scrum Poker. Esta técnica consiste en que cada miembro del equipo del proyecto evalúe el nivel de dificultad por medio de una valoración numérica, obteniendo así un consenso de su nivel de dificultad. En nuestro caso la valoración se la voy a dar únicamente yo y el rango de valores va a ser entre 1 y 5, siendo 1 el nivel más bajo de dificultad y 5 el nivel más alto. Esta técnica nos va a permitir conocer de forma relativa el nivel de dificultad de una tarea y la duración temporal que tendrá en nuestro trabajo.

El listado de tareas y la valoración de la dificultad fueron las siguientes.

Tarea	Subtarea	Evaluación de dificultad
Estudio documental	Estudio del campo (IA)	1
	Deep Learning	4
	Estudio de herramientas	2
Estudio experimental	Estudio python	3
	Estudio keras	2
	Estudio herramientas GPU	4
	Curso Keras	4
	Tutorial MNIST	4
	Tutorial Perros y gatos	4
	Tutorial MNIST-Fashion	4
	Redes preentrenadas	4
	Trasnferencia de aprendizaje	4
	Técnicas de sobreajuste	5
Aportación - Reto Kaggle	Estudio del dataset	3
	Preprocesamiento de datos	3
	Configuración modelo CNN	4
	Monitorización del aprendizaje	4
	Ajuste de hiperparámetros	5
	Evaluación	5
	Técnicas de sobreajuste	5
	Métricas	3
Memoria		2

Cuadro 1. Listado y evaluación de tareas del proyecto

1.3.2 Estructura de descomposición del trabajo

Conocer la valoración de dificultad del conjunto de tareas nos permite realizar una organización temporal más exhausta del proyecto. Un elemento visual de la organización de nuestro proyecto nos permite conocer de forma general el tiempo aproximado para cada labor, además de proporcionarnos una métrica a la hora de evaluar el progreso que llevamos a lo largo del proyecto

Esta organización temporal la llevamos a cabo haciendo uso de la herramienta gráfica diagrama de Gantt. Un diagrama de Gantt es una herramienta visual, cuyo objetivo es la organización temporal, en la que se muestra el tiempo de dedicación a las diferentes tareas a lo largo del proyecto.

Nuestro proyecto ha tenido una duración de 4 meses en total y 300 horas. El diagrama de Gantt desarrollado y en el que nos apoyamos fue el siguiente.

Tarea	Subtarea	Año 2020 Febrero	Marzo	Abril	Mayo	Junio
Estudio documental	Estudio del campo (IA)					
	Deep Learning					
	Estudio de herramientas					
Estudio experimental	Estudio python					
	Estudio keras					
	Estudio herramientas GPU					
	Curso Keras					
	Tutorial MNIST					
	Tutorial Perros y gatos					
	Tutorial MNIST-Fashion					
	Redes preentrenadas					
	Trasnferencia de aprendizaje					
	Técnicas de sobreajuste					
Aportación - Reto Kaggle	Estudio del dataset					
	Preprocesamiento de datos					
	Configuración modelo CNN					
	Monitorización del aprendizaje					
	Ajuste de hiperparámetros					
	Evaluación					
	Técnicas de sobreajuste					
	Métricas					
Memoria						

Cuadro 2. Diagrama de Gantt

Finalmente, tras conocer el conjunto de tareas a desarrollar en el proyecto software, determinar el nivel de dificultad de cada tarea y establecer una organización temporal del proyecto. Vamos a hacer uso de una última metodología Ágil mediante el sistema Kanban para la organización de las tareas durante el desarrollo.

Kanban es un método que forma parte de la metodología Ágil desarrollado para la gestión del trabajo. Esta gestión de trabajo desarrollada en Kanban se realiza mediante el uso de una pizarra y unos post-it. En los post-it indicamos cada una de las tareas que debemos desarrollar a lo largo del proyecto y en la pizarra pintamos 3 columnas, donde cada representa un estado de proceso que tiene una tarea. Estos estados son: pendiente, en curso y completado.

Para llevar a cabo esta metodología de trabajo de proyecto, vamos a hacer uso de la herramienta de trello. Trello es una herramienta gratuita online que nos permite organizar los proyectos a través de tableros los cuales hacen de pizarra y etiquetas en función de post-it.

Esta herramienta tan visual nos permite junto con el Diagrama de Gantt, observar cuantas de las tareas definida para el tiempo establecido se han completado, cuantas están en curso y cuantas aún no se han comenzado. Esto nos permite una mayor organización a nivel proyecto.

Así se encontraba a lo largo del desarrollo del proyecto

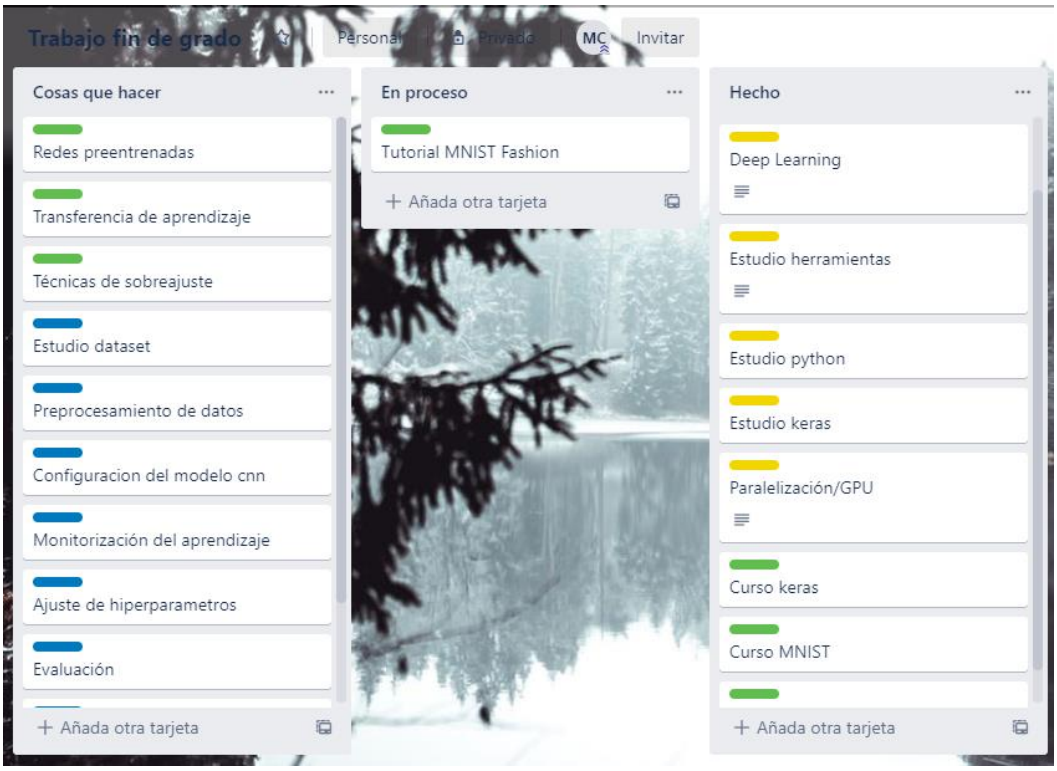


Figura 1. Herramienta Trello. Organización a lo largo del desarrollo del proyecto.

1.3.3 Costes estimados

El coste del proyecto llevado a cabo esta basado en un coste económico mediante el cual conoceremos el valor que alcanza dicho trabajo y por otra parte veremos el coste temporal que conlleva dicho desarrollo. Ambos costes como es natural se encuentran relacionados.

El estudio y desarrollo de dicho proyecto tiene una duración de 4 meses aproximadamente, es decir 17 semanas, comenzando el 10 de febrero y finalizando el 10 de junio.

El tiempo estimado dedicado semanalmente al proyecto ha sido de unas 35 horas semanales, es decir 7 horas diarias de lunes a viernes, siendo en total 595 horas.

El sueldo mensual establecidos por la dirección general de trabajo para un perfil de trabajo en el colectivo de educación universitaria e investigación es de 1692.64 de 40 horas semanales.

Esto determinaría un coste total de nuestro proyecto de 6.294,505 euros

Por otra parte, dicho trabajo posee un coste temporal muy alto. Este coste temporal es casi única y exclusivamente de la subtarea de entrenamiento y/o ajuste de hiperparámetros, sin contar todo el estudio previo a nivel documental y experimental.

Llevar a cabo el desarrollo de un modelo de red convolucional cuyo funcionamiento sea óptimo, requiere un continuo ajuste de hiperparámetros mediante un continuo prueba y error. Esto nos lleva a realizar de forma iterativa nuevas modificaciones de los parámetros de nuestro modelo y su posterior entrenamiento.

El desarrollo de un modelo de red convolucional requiere el uso de una GPU para disminuir este coste temporal. Las condiciones en nuestro campo de trabajo era trabajar con una GPU en la nube de forma gratuita cuya disponibilidad era intermitente o hacer uso de nuestros ordenadores los cuales no disponen de GPU. Es por ello el coste tan alto que ha tenido el proyecto.

Para que el lector se haga una idea del coste temporal que únicamente supone el entrenamiento del modelo. La media total de los distintos entrenamientos llevados a cabo en este estudio ha sido de 5 horas mínimo. Y hemos podido realizar más de 30 entrenamiento.

Por lo tanto, podemos decir que el trabajo desarrollado ha tenido un coste temporal sumamente elevado. Siendo un 75% de este coste temporal en entrenamiento del modelo de red convolucional y el 25% restante, el estudio previo, la experimentación y el desarrollo de la memoria.

1.4 Plan de contingencia

En el caso de la imposibilidad de desarrollar un modelo de red convolucional debido a la falta de herramienta de alto nivel de computación, desarrollamos un conjunto de procedimientos alternativos que nos permitieran el desarrollo del trabajo en dicho campo y con el problema y conjunto de datos que disponíamos.

El plan de contingencia definido estuvo basado en el análisis de imágenes médicas mediante el uso de Python.

Este no tuvimos que llevarlo a cabo ya que gracias a la correcta planificación del proyecto que realizamos y la evaluación de la dificultad que presenta cada tarea, pudimos adelantarnos a dicha dificultad y realizarlos con suficiente tiempo de antelación.

2. Introducción a la temática

En el siguiente apartado comenzaremos hablando de la temática en la que está enfocado nuestro trabajo, la Inteligencia Artificial y los campos que engloba, los cuales emplearemos para abordar nuestro objetivo.

La **Inteligencia Artificial** es la habilidad que se le otorga a una máquina mediante algoritmos para adquirir un aprendizaje, realizar tareas y tomar decisiones, imitando al comportamiento del ser humano. Estas dos características: Aprendizaje y toma de decisión, otorga gran estimación a la IA y es por ello la gran aplicabilidad que tiene en diversos campos, esto es debido a sus benévolos resultados. A diferencia de un ser humano, la IA está ausente del cansancio y saturación que puede llegar a sentir un ser humano al realizar labores de largas horas de trabajo y tratamiento de datos, optimizando dichas tareas. (Rouhiainen, 2018).

Esta disciplina contempla diversos enfoques, nosotros nos centraremos en el cual le capacita al sistema a pensar como un ser humano. Para cualificar con esta mención, la máquina o sistema debe poseer diferentes capacidades, como: Procesamiento del lenguaje natural, visión computacional, robótica, razonamiento automático, representación del conocimiento y aprendizaje automático. Esta última característica nos bifurca a un nuevo campo llamado Machine Learning o Aprendizaje automático. (López, 2017)

Aprendizaje Automático, también conocido como **Machine Learning**, es uno de los subcampos que engloba la Inteligencia Artificial, esta se distingue por una capacidad significante adquirida mediante algoritmos de aprendizaje de patrones, es capaz de aprender algo sin estar determinadamente programado para ello. (López, 2017). Existen tres tipos de Aprendizaje Automático, los cuales discrepan en función del tipo de instrucción que se le otorgue a la máquina. Para explicar los diferentes tipos haremos uso de un ejemplo. Supongamos que queremos clasificar los tipos de zapatos que existe.

El **Aprendizaje Supervisado**, es similar al aprendizaje de un niño pequeño. Los algoritmos aprenden utilizando datos catalogados previamente con etiquetas y una retroalimentación, indicando las categorías en las que se clasificará posteriormente la nueva información. Aplicando nuestro ejemplo, mostraremos al algoritmo imágenes con los diferentes tipos de zapatos y su etiqueta correspondiente, la cual indica a qué clase pertenece. Como hemos dicho con anterioridad, el Aprendizaje supervisado posee algoritmos de aprendizaje de patrones, por lo tanto, posteriormente podrá identificar en una imagen nueva los patrones correspondientes al grupo al que pertenece, clasificando correctamente la nueva información en su grupo correspondiente.

El **Aprendizaje no supervisado**, es un tipo de aprendizaje a ciegas. El algoritmo conoce los datos, pero no sus etiquetas ni las posibles categorías en las que se clasificará posteriormente la nueva información. Dicho algoritmo hará uso de los patrones que ha observado en los datos para sacar unas conclusiones u organización.

Aplicando nuestro ejemplo, se le pasaría el conjunto de imágenes de todos los tipos de zapatos sin indicar a qué clase corresponde y el algoritmo mediante la observación de los patrones que poseen dichos datos devolverá un resultado.

El **Aprendizaje por refuerzo**, es el menos utilizado en la actualidad. Este algoritmo recibe información de su entorno y aprende a través de refuerzo positivo al llevar a cabo una acción acertada. (Rouhiainen, 2018)

Machine Learning posee diferentes tipos de algoritmos, entre ellos, el más utilizado y conocido llamado **Aprendizaje profundo** o **Deep Learning**. Este es una rama dentro del Aprendizaje automático. El aprendizaje profundo discrepa del resto de algoritmos de Aprendizaje Automático tradicionales, este es empleado para el tratamiento de grandes volúmenes de datos, permitiéndoles así un aprendizaje de características y patrones en profundidad, aprobando un aprendizaje y experiencia enormemente profunda en el campo que trabaje. Estos modelos están formados por un conjunto de capas, las cuales les permite una captación de dichos patrones tan específicos.

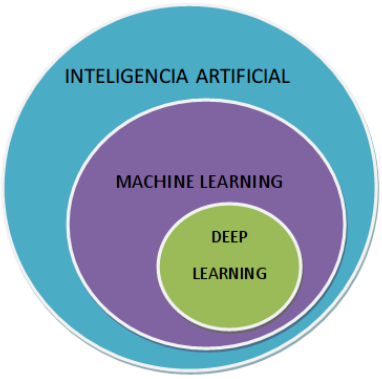


Figura 2. Campo Inteligencia artificial

Para alcanzar dicho aprendizaje, hace uso de un modelo computacional artificial llamado **Redes Neuronales**. Este modelo nació de la observación de nuestra propia naturaleza, el comportamiento y aprendizaje biológico. La estructura de las Redes neuronales artificiales, fueron inspiradas en la transmisión de la información del sistema nervioso, el funcionamiento de las neuronas y la sinapsis, siendo estas las estructuras básicas del Aprendizaje profundo. Por otra parte, el Aprendizaje profundo emula la actividad de las capas neuronales del neocórtex, estas realizan su aprendizaje a través de múltiples capas de representación, transformando un nivel de representación inferior a uno más elocuente.

Dentro del aprendizaje profundo y las Redes neuronales, nos centraremos en las **Redes Neuronales Convolucionales**, estas al igual que los modelos anteriores fueron inspirados en funcionamiento de las neuronas en el córtex visual del cerebro. Son un tipo de Redes Neuronales Artificiales multicapa con aprendizaje supervisado. Tiene gran aplicación en el procesamiento de imágenes (Hernández, 2018). En el siguiente capítulo hablaremos de su estructura y funcionamiento en profundidad.

Como hemos visto el campo de la inteligencia artificial es bastante amplio y abarca entre muchos otros el Aprendizaje Automático y este a su vez el Aprendizaje Profundo. Este está en continuo crecimiento y es por ello que a pesar de que la Inteligencia Artificial naciera hace más de 70 años gracias a Alan Turing, hoy en día

se considera una de las ramas de las ciencias de computación cuyo crecimiento está siendo de forma exponencial. Uno de los principales motivos por lo que su crecimiento se está observando ahora, es debido a la capacidad de cómputo de los ordenadores actuales, los cuales son satisfactoriamente mejores que los de hace una década atrás. Este es un factor indispensable en nuestro campo y es debido a ello su crecimiento, los avances alcanzados y la aplicación en gran número de sectores.

Aprendizaje y toma de decisión son las dos características fundamentales de este campo, las cuales nos resultan atractivas y al mismo tiempo nos puedes dar cierta incertidumbre o pánico. Operaciones que considerábamos propias del ser humano debido a que son inteligentes pueden ser hoy en día automatizadas, prescindiendo del ser humano. Es por ello que debemos tener cierta ética de cara al futuro en este campo con respecto al alcance que puede llegar a tener. (De Paz, 2019).

Reconocimiento de imágenes, estrategias de algoritmos comerciales, procesamiento de datos, mantenimiento predictivo, detección y clasificación, distribución de contenido en las redes sociales y por último ciberseguridad, son algunas de las aplicaciones que más están creciendo en la actualidad

La IA se encuentra actualmente conviviendo con nosotros, abarcando campos propios del ser humano, como puede ser la toma de decisión en una compra online, produciendo así un cambio en la publicidad tradicional de las empresas, las cuales debido a los satisfactorios resultados económicos cada día son más empresas las que implementan este campo en su sector. Predicción y recomendaciones mediante una aplicación móvil sobre nuestra salud y bienestar, hardware robotizados sustituyendo labores fáciles pero tediosas y que suponen un cansancio físico, como puede ser el transportar cajas en un almacén, mejorando con este campo debido a la falta de cansancio y descanso que tiene un robot. (Rouhiainen, 2018).

La IA ha crecido en estos últimos años abriéndose paso en todos los campos de nuestra sociedad y nuestra vida. A nivel de investigación el campo del Aprendizaje profundo tiene una fuerte relación con el sector sanitario debido a la disponibilidad de grandes cantidades de datos de gran riqueza, permitiéndoles de este modo obtener predicciones de sucesos médicos a partir de los datos facilitados, abriéndose paso en la investigación y el desarrollo en la medicina. Por otra parte, las Redes Neuronales Convolucionales, modelo computacional perteneciente al Aprendizaje Profundo, son excelentes en la clasificación de imágenes, siendo así una gran influencia en este sector sanitario, mediante la detección de patologías médicas que pueden presentar un paciente dado un conjunto de imágenes médicas. Esto es debido como hemos comentado con anterioridad al nivel de profundidad de aprendizaje de dichas patologías que puede adquirir este modelo, permitiendo un apoyo al médico a la hora de determinar el diagnóstico de un paciente.

Como hemos visto, la IA está actualmente presente en nuestras vidas. Aún queda mucho que explotar de ella y muchos sectores que explotar con ella, como es el sector sanitario, en el cual nos centraremos nosotros.

3. Estado del arte

Tras una breve introducción a la temática en la que está enfocada nuestro trabajo y tras habernos puesto en contexto. Vamos a profundizar en el campo de Deep Learning y las Redes Neuronales, conocer su estructura y comprender cómo funcionan los algoritmos matemáticos que hay tras el aprendizaje profundo.

Posteriormente, hablaremos de las diferentes tecnologías que hacen posible el desarrollo de estos modelos computacionales, las cuales hemos utilizado para desarrollar nuestro trabajo.

3.1 Deep Learning.

Como mencionamos en el capítulo anterior, Deep Learning es un campo perteneciente al Aprendizaje Automático. Conjunto de algoritmos que discrepan del resto de algoritmos de Aprendizaje Automático debido al tratamiento con grandes volúmenes de datos, adquiriendo un aprendizaje de características y patrones en profundidad de los datos estudiados. Este aprendizaje es caracterizado debido a que es un aprendizaje de capas sucesivas de representación cada vez más significativas Este nivel de profundidad apreciado en sus estudios es debido al modelo computacional utilizado, Redes Neuronales.

Para comprender cómo es su arquitectura y funcionamiento, vamos a verlas más de cerca.

3.2 Redes neuronales tradicionales.

Las Redes Neuronales, es un modelo computacional aplicado en los algoritmos de Aprendizaje profundo, basado en el sistema nervioso animal, es decir la transmisión de la información a través de las neuronas y el proceso de sinapsis.

La unidad básica funcional de una Red Neuronal es el Perceptón o Neurona simple. Dichas Neuronas simples procesa la información de entrada dada, generando una salida. (Julián, 2014). El Perceptón o Neurona Simple está formado por:

- Entradas: Datos de entrada a la neurona
- Pesos: Valores relativos asociado a cada entrada de forma aleatoria. Los cual iremos ajustando a lo largo del entrenamiento hasta alcanzar la salida deseada. Veremos este proceso en profundidad más adelante.
- Sesgo: Valor constante, se le suma a la entrada de la función de activación del perceptrón
- Función de activación: Esta función se aplica sobre el sumatorio de las entradas por su peso correspondiente, atribuyendo una salida no lineal a la neurona. Hay muchos tipos de funciones de activación que explicaremos más adelante.
- Salida: Será la salida de la función de activación. (López, 2017)

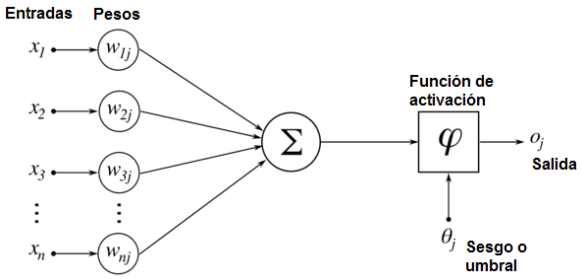


Figura 3. Arquitectura perceptrón

La estructura general de una red neuronal está formada por un conjunto de neuronas simples o perceptrones, conectados entre sí, con una estructura jerárquica organizadas en capas. El procesamiento de la información que realiza las Neuronas de cada capa aumenta en función del nivel jerárquico en el que se encuentre dicha capa. La similitud con el sistema nervioso de los animales, donde la unidad funcional son las neuronas, está en que una neurona recibe una información, la procesa y da una salida, esta salida es la entrada a la neurona del siguiente nivel a la cual está conectada, donde el nivel de procesamiento de información será mayor, aumentando la complejidad de forma jerárquica por niveles, así sucesivamente hasta llegar a la salida final de la red neuronal, dando un resultado complejo esperado

La estructura, se divide en las siguientes capas:

- Capas de entrada.
- Capas Intermedias u ocultas.
- Capas de salida

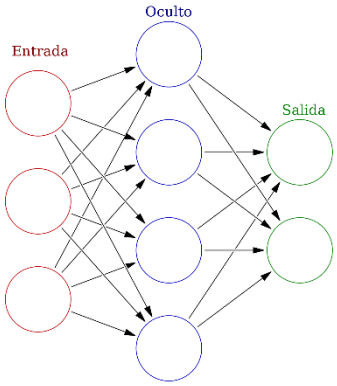


Figura 4. Arquitectura perceptrón multicapas.

3.3 Funcionamiento de las redes neuronales

Ahora que conocemos porque están formadas y cómo es su arquitectura, vamos a hablar de cómo funcionan. Para hacer uso de una Red Neuronal, hay que realizar una serie de pasos previos.

- En primer lugar, **el entrenamiento**, este es un proceso de aprendizaje, mediante el cual la red neuronal adquiere el aprendizaje de patrones en profundidad correspondientes a una clase u otra, llegando a un equilibrio este la generalización y la optimización de la red. Para ello hay que entrenarla.
- En segundo lugar, **la configuración del proceso de aprendizaje**, en el determinamos que tipos de algoritmos vamos a utilizar para aplicar en el proceso de aprendizaje.

- Posteriormente, realizamos la **evaluación de la red neuronal** con un conjunto de datos que no haya visto y evaluamos mediante diversas métricas los resultados obtenidos, fruto de un entrenamiento previo.
- Finalmente, ya podemos hacer uso de nuestra red neuronal. **Generación de predicciones**.

A continuación, vamos a conocer cada uno de los procesos indicados con anterioridad.

3.3.1 Entrenamiento

El entrenamiento consiste en un proceso iterativo de ajuste de los pesos. Vamos a verlo de forma más detallada.

Comenzamos el entrenamiento con unos datos de entrada y una arquitectura de red neuronal definida, al comenzar, cada neurona tiene asignado un valor de peso de forma aleatoria. La red neuronal comienza procesando dicha información a través de las capas de entradas, estas procesan información de bajo nivel hasta niveles más complejos. Cada neurona procesa la información de entrada y realiza sobre los datos una serie de transformaciones que quedan almacenado en los respectivos pesos, dichas transformaciones hacen que los datos difieran cada vez más de los datos de entrada y se hagan más informativas acerca del resultado final. El conjunto de la red, valores de entrada y unos pesos definidos, nos devolverá un valor de salida.

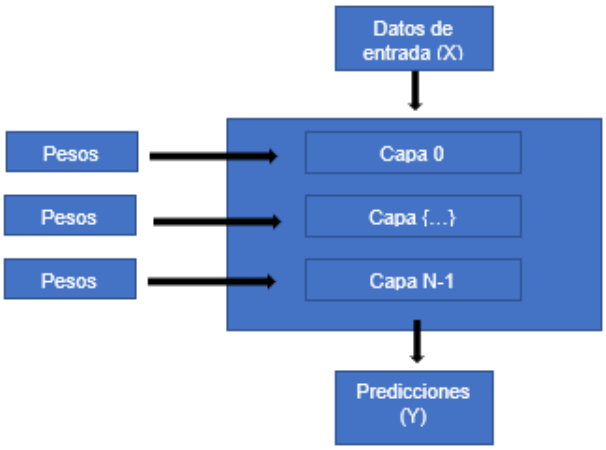


Figura 5. Entrenamiento (I)

Para poder observar el correcto funcionamiento de esta red neuronal, es decir si predice bien un resultado, debemos medir cuán lejos está esta la salida de la red a la salida esperada (etiquetas). Para ello hacemos uso de la función de pérdida, dicha función haya *la diferencia entre el valor de salida correcto a la entrada (etiqueta) y la salida que nuestra red ha predicho. Esto nos permite conocer si nuestra si nuestro modelo funciona bien o no*.

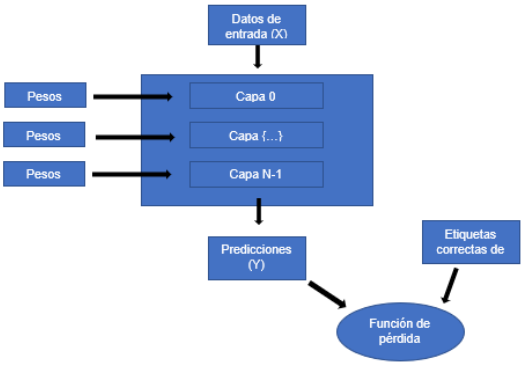


Figura 6. Entrenamiento (II)

Finalmente, y lo más fundamental, el optimizador, el valor que obtiene la función de pérdida es utilizado para retroalimentar la red y realizar un ajuste correcto de los pesos en función de este valor. Este ajuste de pesos es labor del optimizador que implementa el algoritmo de Propagación hacia atrás, algoritmo central del Aprendizaje profundo, el cual le hace diferir del resto.

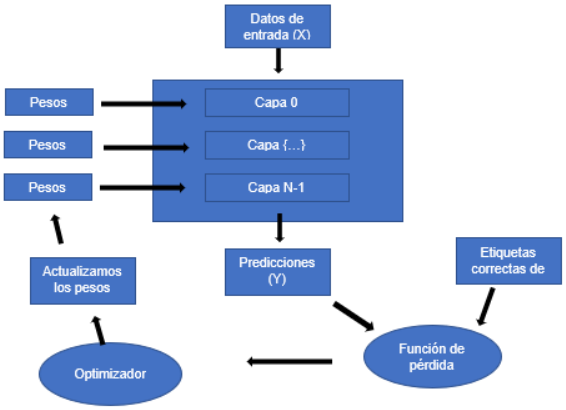


Figura 7. Entrenamiento (III

De forma más técnica, podemos decir que el proceso de aprendizaje de las Redes Neuronales consiste en realizar un mapeo de las entradas (datos de entrada) a los objetivos (etiquetas) a través de una sucesión profunda de transformaciones de datos, las cuales son parametrizadas por sus pesos. Tras la exposición a un gran número de ejemplo, el entrenamiento finaliza con los valores de peso que minimizan la función de pérdida. Quedando una red entrenada correctamente. (Chollet, 2018).

Resumiendo, podemos decir que el bucle del proceso de aprendizaje consta de 4 pasos los cuales se repiten hasta alcanzar como ya habíamos mencionado los valores de pesos que minimicen la función de error. Estos 4 pasos son:

- Dado unos datos de entrada. Asignar a cada dato de entrada X, su salida u objetivo correspondiente Y.
- Ejecutar una red con los datos de entrada X y obtener la predicción de salida de la red, Y_pred.
- Aplicar la función de pérdida entre Y_pred y Y.
- Actualizar los pesos de la red para reducir la pérdida.

Realmente es un proceso muy sencillo y fácil de entender, pero ¿Cómo sabe la red si debe aumentar o disminuir sus pesos en función del valor de la función de error que le da? Se podría probar congelar todos los pesos y uno de ellos modificarlo aumentando su valor un poco, ejecutando la red y si la función de error ha disminuido, ¡Enhorabuena! Has encontrado un peso que contribuye a minimizar la pérdida, ahora a probar con el resto, pero ¿Es posible aplicar esto en algoritmos de profundidad? No, es imposible, además de que sería extremadamente ineficiente. Por ello se propuso una solución mejor, dado que los datos son diferenciables, aplicamos el Cálculo del gradiente en la función de pérdida.

Vamos a explicar estos dos conceptos básicos matemáticos un poco más en profundidad:

3.3.1.1 Conceptos básicos

- ¿Qué es una derivada?

Dado una función continua F(X) = Y. Lo que supone que un campo en X provoca un cambio en Y, por lo tanto, un pequeño aumento en X, X_aum, supone un pequeño aumento en Y, Y_aum. Esto es lo que se entiende por una función continua.

Una Derivada nos permite, a través de la pendiente (A) en todo punto de la curva, observar el cambio o evolución que dicha función (F) toma con respecto a la variable que rodea una región muy pequeña cerca de un punto dado (P). Además, nos permite conocer información más relevante como la existencia de máximos y mínimos, esto se da cuando el valor de la pendiente es 0.

Apoyándonos en la definición, una derivada nos permitirá conocer los distintos valores que pueden tomar los pesos. El resultado del gradiente de la función de pérdida, con respecto a los coeficientes de la red, es decir, los pesos, nos indicará los distintos valores que pueden tomar los coeficientes, esta evolución de aumento o disminución del valor se realizará en sentido opuesto a dicho gradiente, disminuyendo así la pérdida.

La pendiente (A) se designa la derivada de la función (F) entorno a dicho punto (P). Si la pendiente (A) es negativa, significa que un pequeño cambio de X alrededor del punto (P) resultará una disminución de la función F(X). Si la pendiente (A) es positiva, significa que un cambio en la variable X alrededor del punto, supone un aumento de la función F(X).

Ya sabiendo que es una derivada, vamos a adentrarnos un poco más definiendo lo que es una derivada parcial, dado una función de un conjunto de variables, una derivada parcial, es la derivada respecto a cada una de las variables, contando el resto de las variables como constante.

- ¿Qué es un gradiente?

Como hemos mencionado, vamos a hacer uso de las derivadas para ver mediante su pendiente las modificaciones de los valores que debe realizar los coeficientes de la red, para alcanzar el mínimo de la función de error. La pendiente, es la derivada de la función en un punto. Como en este caso trabajamos con un conjunto vectorial de entradas, trabajaremos entonces con gradientes.

Un gradiente ∇f, es la generalización del concepto de derivada a funciones de entradas multidimensionales. Es decir, es el derivado de una operación tensorial,

funciones que toman tensores como entradas. De forma simplificada, es el vector de las derivadas parciales con la misma forma que la entrada (multidimensional), en este caso las derivadas parciales corresponden a las derivadas respecto a cada uno de los coeficientes de la red.

Aplicado a nuestro campo, dada una entrada matricial W, la derivada de F en el punto W0, es decir F(W0) es un gradiente tensorial con la misma forma que W, donde cada gradiente de los coeficientes F(W0) indica la dirección en la que debe evolucionar cada coeficiente para minimizar la función de pérdida El gradiente F(W0) es la pendiente de F(W) alrededor del punto W0.

Nuestro objetivo es minimizar la función de pérdida para ello su gradiente debe ser igual a 0, es decir F(W) = 0. Esta se trata de una ecuación polinómica de N variables, siendo N el número de pesos o coeficientes de la red. El número de pesos que tiene una red neuronal nunca baja a menos de miles de valores, es por ello que no podemos aplicarlo en bruto. Es su lugar aplicamos el algoritmo de bucle descrito con anterioridad, en el cual los valores de pérdida se van modificando y actualizando de forma iterativa en dirección opuesta al gradiente de la función de pérdida actual. Esta función de pérdida hará uso del algoritmo iterativo Descenso de gradiente Estocástico que a continuación vamos a explicar. (Chollet, 2018).

3.3.1.2 Descenso de gradiente

Algoritmo iterativo que aplica la función de pérdida, la cual llamaremos E. Esta función de pérdida nos devuelve el error que comete la Red neuronal definida por un conjunto de pesos W. El objetivo de aplicar este algoritmo en nuestra función de pérdida es encontrar el conjunto de coeficientes o pesos que correspondan al mínimo global de la función de pérdida E(W).

Sabemos que el mínimo de una función es aquel punto donde la derivada de la función entorno a ese punto es 0, es decir E(W0) = 0. Por ello debemos encontrar todos aquellos valores para los cuales su derivada sea 0, consiguiendo minimizar la función de error. Este concepto generalizado, aplicado a nuestro problema, consiste en que debemos encontrar la combinación de valores de pesos que hagan la función de pérdida sea mínima.

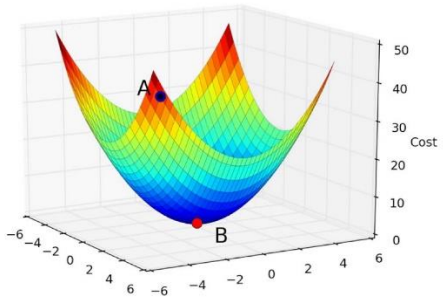


Figura 8. Descenso de gradiente

El algoritmo en la primera iteración W en t = 0, calculará el gradiente de la función E(W) para W en t = 0, es decir, E(W0), obteniendo así la pendiente de la función en ese punto. Teniendo la pendiente el algoritmo sabe hacia donde se tiene que mover para alcanzar ese mínimo local, esto es en sentido contrario al gradiente. Posteriormente, se generarán variaciones en los coeficientes de W, W(t+1) =W(t)-E(W), donde indica el pequeño factor de escala indicado en cada iteración, e iteramos de nuevo, de este modo se producirá un descenso por la superficie de error hasta alcanzar el mínimo global de la función de pérdida.

En nuestro caso haremos uso del algoritmo de Descenso de Gradiente Estocástico. El término estocástico, hace referencia a que cada lote de dato a evaluar se producirá de forma aleatoria.

Existen muchas variantes del algoritmo Descenso de Gradiente Estocástico (SGD), muchas se diferencian por el hecho de tener en cuenta las actualizaciones de los pesos, en lugar de únicamente observar el valor actual de los gradientes u otros muchos factores como el uso de impulso. Estas últimas son llamadas optimizadores y hablaremos de ellas en profundidad más adelante, al igual que del Descenso de gradiente estocástico de mini-batch.

3.3.1.3 Backpropagation

Hemos dicho que la función de pérdida hace uso del algoritmo de descenso de gradiente para encontrar la conformación de pesos que optimicen la función de pérdida. Esto se lleva a cabo mediante el gradiente de la función.

En la practica una función de red neuronal consiste en múltiples operaciones tensoriales encadenadas, donde cada una es derivable. Esta cadena de funciones puede ser derivada haciendo uso de la regla de la cadena.

$$F(g(x)) = f'(g(x)) \times g'(x)$$

El algoritmo de backpropagation o propagación hacia atrás, nace de aplicar la regla de la cadena al cálculo de los valores de gradiente de una red neuronal. Este algoritmo inicia con el valor de pérdida final hacia atrás, es decir comienza en las capas superiores hacia las capas inferiores, aplicando la regla de la cadena para calcular y conocer la aportación de cada neurona sobre el valor de la función de pérdida.

Vamos a verlo con un claro ejemplo. Dado la función

$$f(x, y, z) = (x + y) \times z$$

Podemos subdividirla en:

$$q(x, y) = x + y$$

$$f(x, y, z) = q \times z$$

Donde, sus derivadas parciales serian:

$$\frac{\partial q}{\partial \mathrm{x}} = 1 \quad \frac{\partial q}{\partial y} = 1 \quad \frac{\partial f}{\partial q} = z \quad \frac{\partial f}{\partial z} = 1$$

Para los valores de:

$$x = -2;\ y = 5;\ z = -4$$

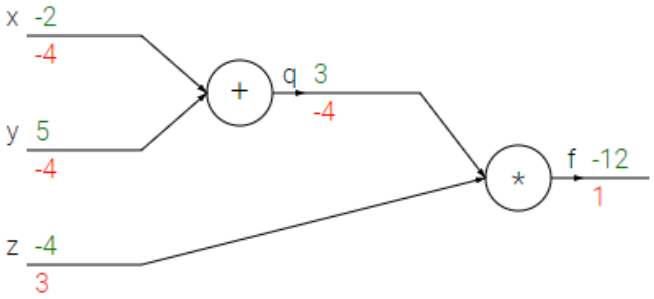


Figura 9. Backpropagation

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \times \frac{\partial q}{\partial x} = 1 \times z = 1 \times (-4) = -4$$

El gradiente local de la puerta de adicción q es para ambas puertas +1. El gradiente de salida final fue -4. Por lo tanto, la puerta de adicción toma ese valor y lo multiplica por los gradientes locales suyos, haciéndole conocer así a cada neurona la modificación a realizar sobre los pesos. Estas modificaciones de los pesos se realizan en función del gradiente, en este caso al ser multiplicado cada gradiente local por -4, siendo ahora su gradiente local -4. Los pesos de estas neuronas disminuirán sus valores respondiendo a sus gradientes con una fuerza de 4, provocando un aumento en la salida final de la red.

Podemos entender el algoritmo de la propagación hacia atrás, como un conjunto de neuronas que se comunican entre sí, mediante señales de gradiente, las cuales indican con que intensidad deben aumentar o disminuir sus pesos en busca de una la salida óptima.

Ahora que ya comprendemos como una red neuronal durante el entrenamiento consigue minimizar la función de pérdida y actualiza sus pesos. Vamos a determinar y a hablar de los diferentes parámetros a establecer para la configuración del proceso de aprendizaje. (Stanford CS)

3.3.1.4 Función de activación

Finalmente, dentro del proceso de aprendizaje hablaremos de la función de activación.

La función de activación realiza el proceso de propagar la salida de las neuronas hacia adelante, permitiendo así a la red obtener una salida. Esta introduce la no linealidad en las capas de modelo de red neuronal.

Hay diversos tipos de funciones de activación, pero las más usuales en modelo de redes neuronales son ReLu, Sigmoid y Softmax.

La función **Sigmoid** reduce los valores atípicos en datos validos sin ser estos eliminados. Convierte las variables independientes de rangos infinitos en una probabilidad binaria entre 0 y 1.

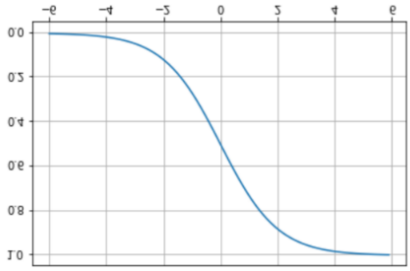


Figura 10. Función Sigmoid

Softmax es la función de activación más utilizada en la capa de salida de las redes neuronales. Esta a diferencia de la anterior, devuelve la distribución de probabilidad sobre las clases de salida sobre las que trabaje.

La función de activación rectified linear unit (**ReLu**), realiza la activación del nodo si dicha entrada se encuentra por encima a un determinado umbral. El comportamiento más habitual de esta función y el que se encuentra por defecto es **F(x) = max(0, x),** es decir, si el valor de entrada se encuentra por encima de un determinado umbral, la salida es una relación lineal de la forma F(x) = x, sin embargo, si el valor de entrada se encuentra por debajo del umbral, el valor de salida es 0. (Torres, 2020)

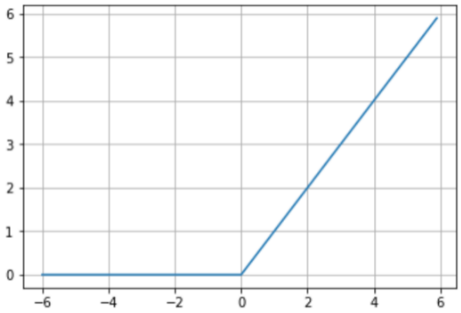


Figura 11. Función ReLu

3.3.2 Configuración del proceso de aprendizaje.

La configuración del proceso de aprendizaje consiste en determinar una serie de algoritmos determinados para los diferentes parámetros que actúan en el proceso de aprendizaje.

Estos parámetros son el tipo de función de pérdida, el algoritmo a aplicar para el descenso de gradiente, este parámetro es llamado optimizador y finalmente una métrica con la que nos guiaremos para observar el rendimiento del entrenamiento.

La **función de pérdida**, como ya hemos mencionado con anterioridad, nos determina cuan cerca esta la salida que redice la red neuronal de su salida ideal durante el proceso de entrenamiento. La función de pérdida debe ser elegida en función del tipo de error aceptado o no para el problema en concreto, variando entre un problema de clasificación binaria, multiclase, regresión …etc. (Torres, 2020)

Como **optimizador** podemos elegir diferentes tipos de algoritmos para el descenso

de gradiente, como vimos con anterioridad, por medio de este algoritmo buscamos el mínimo de la función de pérdida. La elección de establecer un tipo de algoritmo u otro depende de nosotros, para ello es bueno conocer que la eficacia de un optimizador se mide por la velocidad de convergencia (alcanzar un óptimo global) y la generalización (rendimiento dado unos datos nuevos)

Los algoritmos más utilizados son SGD, Adam, RMSprop, después conocemos múltiples variantes como AdaGrad, Adadelta, Adamax, Nadam.

SGD, es una variante del descenso de gradiente estocástico. Esta diferencia del algoritmo tradicional del descenso de gradiente realiza cálculos por conjuntos pequeños de datos. Una extensión de ella es RMSProp, la propagación cuadrática media.

Adam, es un algoritmo más popular y conocido hoy en día, este está basado en gradiente de funciones objetivo estocástica. Combina lo destacable de dos extensiones de SGD como la propagación cuadrática media (RMSProp) y el algoritmo AdaGrad. (Peng, 2019)

Finalmente, como último parámetro a determinar la **Métrica.** Esta es una herramienta que nos permite conocer la efectividad de nuestro modelo.

Problem type	Last-layer activation	Loss function
Binary classification	`sigmoid`	`binary_crossentropy`
Multiclass, single-label classification	`softmax`	`categorical_crossentropy`
Multiclass, multilabel classification	`sigmoid`	`binary_crossentropy`
Regression to arbitrary values	None	`mse`
Regression to values between 0 and 1	`sigmoid`	`mse` or `binary_crossentropy`

Cuadro 3. Función de activación y pérdida en función del tipo de problema.

3.3.3 Evaluación

Una vez configurado un modelo de red neuronal y su proceso de aprendizaje y haber entrenado a la red neuronal. Tenemos que evaluar dicha red ya entrenada.

En este proceso la red ya ha sido entrenada, es decir no adquiere más aprendizaje. Evaluamos el modelo con un conjunto de datos nunca visto, el conjunto de test.

Este proceso nos devuelve dos valores, el valor de pérdida y la métrica obtenida sobre los datos de test, normalmente como métrica se determina el porcentaje de aciertos que ha tenido la red (accuracy) por medio de estos dos valores podemos estimar el entrenamiento de la red, es decir, si la red ha sido correctamente entrenada o no.

Es conveniente, hacer uso de otras métricas aparte como puede ser una matriz de confusión. Esta nos permite visualizar la ejecución del algoritmo, no solo el número de ejemplos que han sido clasificados correctamente para cada clase, sino también el número de ejemplos clasificados incorrectamente. (Torres, 2020)

3.3.4 Predicciones

Llegados a este punto, hemos desarrollado un modelo de red neuronal cuyo

entrenamiento ha sido satisfactorio. Ya podemos hacer uso de el para satisfacer los objetivos por los cuales hemos realizado esta implementación, obteniendo unas predicciones correctas del modelo sobre el conjunto de datos que le facilitemos como entrada.

3.4 Redes neuronales convolucionales.

Ahora que ya entendemos que es una red neuronal tradicional y cuál es su funcionamiento, vamos a adentrarnos en una arquitectura específica de red neuronal en la que centraremos en nuestro trabajo e implementaremos, esta son las redes neuronales convolucionales (CNN).

Las redes neuronales convolucionales son el algoritmo de aprendizaje profundo más utilizado en tareas de visión computacional. Estas nacieron a finales de los años 90, pero no han sido explotadas hasta hoy en día. La capacidad computacional de los ordenadores actuales ha sido el detonante para el crecimiento de todo el campo del aprendizaje profundo, pero con más índole en las redes neuronales convolucionales, debido a la principal diferencia con las redes neuronales tradicionales, estas trabajan con imágenes, lo que supone la necesidad de un rendimiento computacional mayor.

Una red neuronal convolucional, tiene una arquitectura similar al perceptón multicapas de una red neuronal tradicional, donde está compuesta por unas capas de entradas, capas ocultas y capas de salida. La primera diferencia se encuentra en que las CNN trabajan con imágenes como datos de entrada y la segunda es que el tipo de capa por el que está compuesta las CNN no son capas tradicionales densamente conectadas, sino está formada por un conjunto de capas convolucionales.

Al igual que el entrenamiento de una red neuronal tradicional en el cual adquiere un aprendizaje de patrones. Las redes neuronales convolucionales realizan un aprendizaje de patrones y características en profundidad, por medio de un nivel de capas jerarquizado donde cada capa aprende un nivel de abstracción diferente cada vez más significativas.

La principal diferencia entre una red neuronal tradicional y una red neuronal convolucional son el tipo de capas por las que está formada. Una red neuronal tradicional está formada por un conjunto de capas densamente conectadas, esta realiza un aprendizaje de patrones globales en las características de entrada, sin embargo, una capa convolucional propia de las CNN detecta y realizan el aprendizaje sobre patrones locales de las características de entrada mostradas por medio de un Kernel. Los patrones locales, estos pueden variar de lugar en los datos de entrada, de tamaño, sin embargo, estas variaciones de los patrones en el espacio en redes neuronales tradicionales no serían detectados por lo que tendría que aprender el patrón de nuevo si apareciese en otro lugar.

Como podemos observar en la siguiente imagen, dado una imagen de entrada formada por un conjunto de píxeles que representan un gato. La primera capa corresponde a una capa convolucional donde los patrones aprendidos son pequeños patrones y características en profundidad locales, posteriormente una segunda capa con un mayor nivel de abstracción por medios de las características de la primera capa, la cual determina la salida, es un gato.

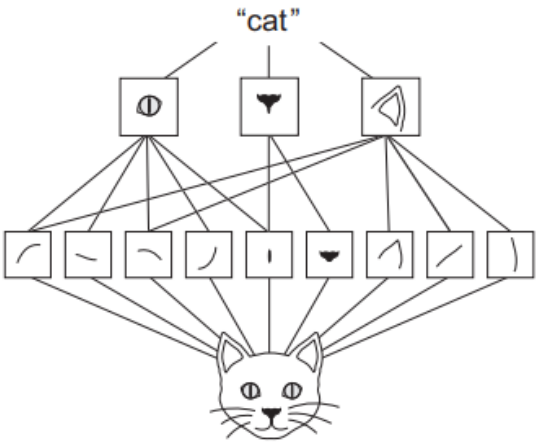


Figura 12. Aprendizaje Convolucional (patrones locales)

Esta es la principal característica de las redes neuronales convolucionales la cual le hace ser muy eficiente y destacar por encima de algoritmos presente en el aprendizaje profundo. (Chollet, 2018)

Estas capas no se conectan densamente como una Red Neuronal tradicional, en la que todas las neuronas de la capa de entrada se conectan con todas las neuronas de la capa oculta. Las Redes Neuronales convolucionales aprenden pequeños patrones mediante el uso de un Kernel. Un kernel es una pequeña ventana de dos dimensiones, por la cual muestra un conjunto reducido de neurona de los datos de entrada, sobre los cuales realiza el aprendizaje de patrones locales. Este Kernel hace que la conexión entre capas no sea densamente, sino, el kernel conecta el grupo de neuronas de la capa de entrada mostrada por la ventana, con una neurona de la capa oculta.

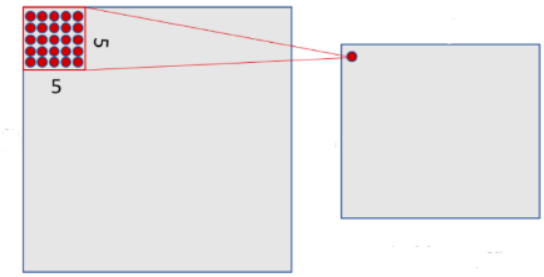


Figura 13. Conexión convolucional

El kernel va recorriendo los datos de entradas y por cada posición que toma durante el recorrido, es decir por cada grupo de neuronas de entrada en las que se para, es conectada a una neurona de la capa oculta que procesa la información y cuyo valor es el producto escalar entre el Kernel y el grupo de neuronas que procesaba, generando así una matriz de salida que será una nueva capa en las neuronas ocultas.

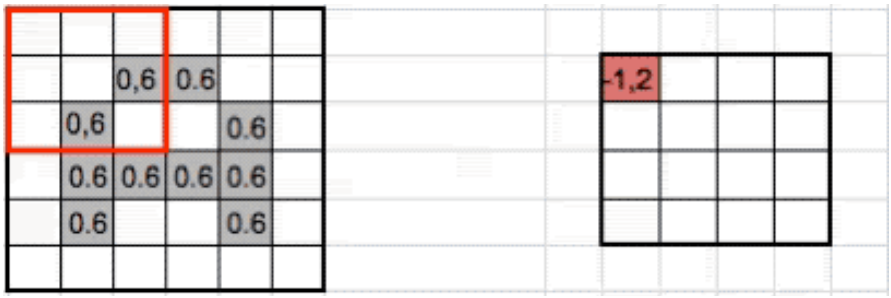


Figura 14. Proceso de ejecución de un Kernel

Las operaciones convolucionales operan sobre tensores 3D, llamado mapa de características, formado por los ejes altura, anchura y profundidad. En función del conjunto de datos con el que trabajemos, la dimensión del eje profundidad será de 1 si es escala de grises o 3 si es color.

El kernel recorrerá el mapa de características dado como entrada, este es conectado como hemos mencionado con una neurona de la capa oculta la cual realiza una serie de operaciones y transformaciones y devuelve otro mapa de características. Este será una nueva capa que forma el mapa de características.

Hay que destacar que no solo se aplica un Kernel por convolución, sino que aplicaremos tantos como queramos, donde cada uno dará como salida una matriz que compone el mapa de características.

Las diferentes matrices que componen el mapa de características generadas en una misma convolución han usado el mismo filtro o Kernel definido con la misma matriz de pesos y el mismo sesgo. Un filtro definido por un matriz de pesos y un sesgo, sólo posibilita detectar una característica concreta en una imagen, es por ello que se aplican más de un kernel sobre la misma entrada, aumentando así el número de características a detectar. (Torres, 2020)

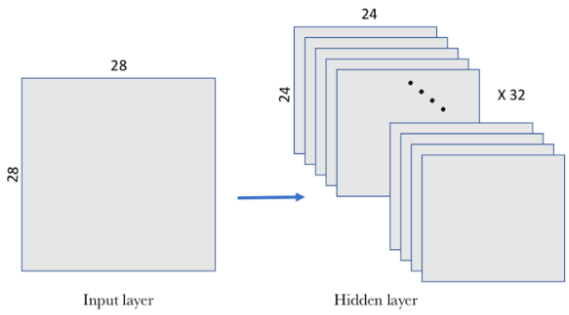


Figura 15. Mapa de características.

Resumiendo, podemos decir que la arquitectura de una red neuronal convolucional enfatiza en que dicha arquitectura trabaja con un tipo de capas específicas que son las convolucionales y los datos de entrada de una CNN, donde estos datos son imágenes.

En el siguiente capítulo veremos cómo implementar una red neuronal convolucional y que otro tipo de capas adicionales podemos aplicar para la construcción de un modelo robusto de red neuronal.

3.5 Herramientas:

A continuación, vamos a introducir las herramientas con las que trabajaremos y sobre la que nos apoyaremos para llevar a cabo la implementación de nuestra red neuronal convolucional.

Realizaremos en primer lugar la implementación de dicho trabajo mediante el uso de Keras. Este es un entorno de trabajo de alto nivel de aprendizaje profundo para Python, la cual nos proporciona de forma simple de definir y entrenar un modelo de aprendizaje profundo.

Keras es una biblioteca a nivel de modelado de alto nivel, que nos facilita conjuntos para llevar a cabo la construcción de modelos de alto nivel como los modelos de aprendizaje profundo. Esta no realiza operaciones de bajo nivel como puede ser la manipulación de tensores, para ello hace uso de bibliotecas de bajo nivel. Keras hace uso de diversos módulos, como los módulos de implementación backend Tensorflow, backend Theano y backend Microsoft Cognitive Toolkit.

Keras destaca por las siguientes características:

- Posibilita que el código se ejecute haciendo uso de la CPU o GPU.
- Tiene una API sencilla que posibilita la creación de modelos de aprendizaje profundo con rapidez.
- Soporta una gran variación de arquitecturas como redes convolucionales, redes recurrentes, redes arbitrarias… etc.

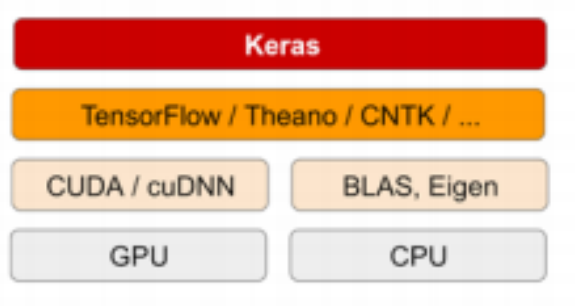


Figura 16. Backend Keras

Aunque Keras puede trabajar con diversos módulos de bajo nivel, es recomendable hacer uso de Tensorflow de forma predeterminada.

Tensorflow es una biblioteca de software de código abierto más conocida para la construcción y entrenamiento de modelos de aprendizaje profundo. Para ello utiliza gráficos de flujos de datos, donde los nodos en dichos gráficos representan operaciones matemáticas, mientras que los bordes de las gráficas representando tensores comunicados entre ellos. Su arquitectura de tensorflow le permite llevar a cabo el cálculo en una o más CPU o GPU de los distintos dispositivos finales. (Buhigas, 2018)

Esta herramienta fue creada debido a la investigación del aprendizaje profundo y el deseo por desarrollar el aprendizaje automático de forma fácil, sencilla y accesible a todo el mundo. (Chollet, 2018)

Tensorflow nos permite llevar a cabo la ejecución mediante el uso de una CPU o GPU, por lo que Keras puede ejecutar en ellas. El desarrollo de redes neuronales

computacionales requiere un alto nivel computacional, por ello es conveniente hacer uso de una GPU.

Para el desarrollo de este trabajo hemos hecho uso de la herramienta de Google Colab. Esta es una herramienta en la nube proporcionada por Google que permite ejecutar código en Python haciendo uso de sus GPU de forma gratuita.

Las condiciones para disfrutar de estas no son las más satisfactorias, ya que la disponibilidad de sus GPU es intermitente y tras un tiempo de inactividad se apaga, pero es útil sino se tiene otra alternativa al alcance.

4. Casos de estudio

El estudio de la temática para una posterior implementación experimental se ha llevado a cabo mediante diversos notebook y cursos tutoriales, enfocando la ejecución de dicha temática en diversos enfoques y con distintos objetivos, permitiendo así conocer los distintos niveles de profundidad.

En campo en el que estamos trabajando, como ya había comentado con anterioridad es un campo muy amplio, cuyos límites no se encuentran definidos hoy en día.

El análisis de los diversos estudios que mencionaremos a continuación nos ha permitido conocer en profundidad la implementación del Aprendizaje profundo.

A continuación, hablaremos de lo aprendido en los estudios más significativos.

4.1 Modelo de Red Neuronal Convolucional

En este estudio vamos a explicar la arquitectura básica de una red neuronal y cómo implementarla. Para ello, iremos profundizando en cada concepto que salga de forma teórica y su posterior implementación.

Para ello vamos a hacer uso del dataset Fashion de MNIST. Nos enfrentamos a un problema multiclase, donde nuestros datos de entrada son imágenes de ropa.

- **Entradas.**

Las capas convolucionales, operan sobre tensores 3D. Estos están formados por los ejes de altura, anchura y canal, este último hace referencia a la profundidad.

La entrada de una Red Neuronal Convolucional está formada por el conjunto de píxeles que forman una imagen, siendo cada uno de estos píxeles una neurona de la capa de entrada de la Red Neuronal Convolucional.

Una imagen de entrada de color de 28x28, tiene una dimensión (28, 28, 3) donde el eje profundidad está formado por 3 canales: rojo, verde y azul, por ello que tenga este eje una dimensión de 3. Una imagen con esta dimensión supone un total de 2352 neuronas en la primera capa de la red. Sin embargo, una imagen en blanco y negro de 28x28 píxeles, el canal profundidad tiene una dimensión de 1 (escala de grises), teniendo por lo tanto la imagen una dimensión de (28, 28, 1) con un número total de 784 neuronas en la capa de entrada.

En nuestro, dataset de Fashion_mode de MNIST, está formado por un conjunto de imágenes de dos dimensiones 28x28 píxeles (28, 28,1), con una dimensión en el eje canal de 1 debido a que estamos trabajando con imágenes en blanco y negro, las cuales serán las entradas a nuestra red neuronal.

Antes de alimentar una red, los datos deben ser formateados para convertirlos en tensores de punto flotante. Para ello debemos leer los datos, convertir el formato de la imagen en una matriz de escala de grises en nuestro caso, convertir la matriz en tensores de punto flotante y finalmente normalizar los datos. Los valores de los píxeles

van de 0 a 255, es por ello que haremos una transformación de esto normalizando los valores entre 0 y 1.

Aplicado a nuestro problema, en primer lugar, mediante el uso de la librería pandas, leemos los datos con la función ***read_csv***, la cual nos devuelve un dataframe.

```
train_file = "../input/digit-recognizer/train.csv"
raw_data = pd.read_csv(train_file)
```

Código 1. Fragmento de código. Lectura dataset

Posteriormente, mediante la función **data_prep**, preparamos nuestros datos, esta función nos devolverá en el parámetro out_x un paquete con el conjunto de imágenes, en la cual cada imagen ha sido remodelada mediante la función **reshape()** a una dimensión de (28. 28, 1), cada píxel ha sido posteriormente normalizado entre [0, 1], dividiendo su valor entre 255.

El segundo parámetro que nos devuelve la función data_prep es out_y, esta es una matriz binaria de la entrada, es decir, es una matriz con las etiquetas en binario de la clase a la que pertenece cada imagen de entrada, esto se realiza mediante la función proporcionada p*or keras* ***to_categorical()***

```
img_rows, img_cols = 28, 28
num_classes = 10

def data_prep(raw):
    out_y = keras.utils.to_categorical(raw.label, num_classes)

    num_images = raw.shape[0]
    x_as_array = raw.values[:,1:]
    x_shaped_array = x_as_array.reshape(num_images, img_rows, img_cols, 1)
    out_x = x_shaped_array / 255
    return out_x, out_y
```

Código 2. Fragmento de código. Preprocesamiento de los datos

- **Configuración del proceso de aprendizaje - Definimos un modelo:**

Tras haber hecho las transformaciones oportunas a nuestros datos de entrada, vamos a comenzar a describir nuestro modelo de red neuronal convolucional.

La estructura de nuestra red neuronal va a ser diseñada de forma secuencial, para ello la librería Keras nos proporciona la función ***Sequential***(), esta función implementa una estructura la cual está formada por una pila de capas donde cada una de ellas realiza una serie de transformaciones a sus datos de entrada proporcionando una salida.

```
from tensorflow.keras.models import Sequential
fashion_model=Sequential()
```

Código 3. Fragmento de código. Modelo secuencial

Keras nos proporciona el método ***add***() mediante el cual nos permite añadir las capas que necesitemos. Nosotros queremos hacer uso de las capas convolucionales.

- **Convoluciones**

Las capas convolucionales, es lo distintivo de una Red Neuronal Convolucional y por lo que destaca significativamente en los resultados. Cómo ya vimos estas son capaces de aprender patrones muy específicos y jerarquía de patrones manteniendo las relaciones con dichos patrones de niveles inferiores.

Vamos a ver cómo podemos implementar todo lo contado con anterioridad en nuestro problema.

Con Keras esto se realiza de forma muy sencilla, mediante el método ***add()*** añado a mi modelo secuencial una capa convolucional ***Conv2D***, en cuyos parámetros le indico un ***kernel*** de dos dimensiones 3x3, el número de veces que se va a aplicar un Kernel en esta convolución sobre los datos de entrada, siendo en este caso 12 y finalmente la dimensión de nuestros datos de entrada (28x28x1).

Sobre esto aplicamos la función de activación ***Relú***. Esta es una de las funciones de activación más utilizadas en este tipo de redes neuronales. Como ya mencionamos en el capítulo anterior, la función de activación Relu solo activa los nodos que están por encima de cierto umbral. El más usual **F(x) = max(0, x).** (Bagnato, 2018)

```
img_rows, img_cols = 28, 28
fashion_model.add(Conv2D(12, activation='relu', kernel_size=3, input_shape = (img_rows, img_cols, 1)))
```

Código 4. Fragmento de código. Capa de convolución.

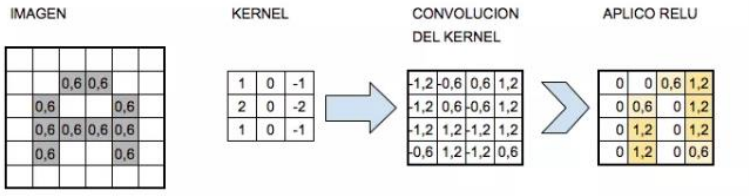


Figura 17. Proceso primera convolución.

Se añaden posteriormente más capas convolucionales tal y como acabamos de hacer. En este caso, dos capas convolucionales con 20 filtros cada una y una dimensión del *Kernel* de 3x3, con una función de activación Relu.

```
fashion_model.add(Conv2D(20, activation='relu', kernel_size=3))
fashion_model.add(Conv2D(20, activation='relu', kernel_size=3))
```

Código 5. Fragmento de código. Capas convolucionales.

- **Aplanar**

La salida de nuestra Red Neuronal en la última capa de convolución ha sido un tensor 3D formado por 20 filtros de 28x28x1.

Como la siguiente capa que vamos a aplicar es una capa densa, es decir una capa de neuronas tradicionales, tenemos que ajustar la salida de una capa la cual es un tensor 3D a la entrada de la siguiente, la cual requiere un tensor 1D. Es por ello que vamos a hacer uso de la función ***flatten()***, esta aplana la entrada dando como resultado una dimensionalidad de 1.

```
fashion_model.add(Flatten())
```

Código 6. Fragmento de código. Función Flatten

- **Capa tradicional**

Ahora que nuestra salida es una salida de una dimensión y podemos conectarla a una capa de red neuronal tradicional, realizamos la conexión densamente, donde todas las neuronas se encuentran conectada con la siguiente capa.

En nuestro caso añadimos una capa oculta densa de red tradicional de 100 neuronas conectadas densamente a la capa anterior, con una función de activación *Relu.*

```
fashion_model.add(Dense(100, activation='relu'))
```

Código 7. Fragmento de código. Capa oculta tradicional.

Finalmente, añadimos una capa de salida, formada por el número de neuronas en función de número de clases en las que se clasifica el problema. Aplicamos la función de activación Softmax, la cual nos devuelve el resultado en forma de probabilidad entre 0 y 1.

La salida de cada neurona de la capa de salida será un vector one-hot-encoding. (Bagnate, 2018)

```
fashion_model.add(Dense(100, activation='relu'))
fashion_model.add(Dense(10, activation='softmax'))
```

Código 8. Fragmento de código. Capa de salida

- **Monitorización del proceso de aprendizaje.**

Tras haber definido un modelo de Red Neuronal convolucional, monitorizamos el proceso de aprendizaje mediante el método ***compile()***, mediante el cual podemos ajustarnos más al problema a través de sus parámetros.

Para definir qué tipo de parámetros asignamos para monitorizar el proceso de aprendizaje, hay que tener en cuenta el tipo de problema en el que nos encontramos. En nuestro caso, al tratarse de un problema de clasificación de multiclase, la ***función de pérdida*** que más se ajusta a este caso es la ***categorical_crossentropy***, por otro lado, como algoritmo de optimización se ha escogido en este caso el ***optimizador adam*** este es un algoritmo cuya optimización se basa en gradientes de funciones objetivo estocásticas. Y finalmente para observar el proceso de aprendizaje de la Red Neuronal usaremos como ***métrica accuracy.*** (Torres, 2020)

```
fashion_model.compile(loss='categorical_crossentropy', optimizer='adam',metrics=['accuracy'])
```

Código 9. Fragmento de código. Compilación del proceso de entrenamiento.

- **Entrenamiento del modelo.**

Después de haber definido un modelo y monitorizado el proceso de aprendizaje, vamos a entrenarlo, es decir ajustar el modelo a los datos de entrenamiento. Para ello keras nos proporciona la función ***fit().*** Un modelo puede ser mejor o peor entrenado en función de los argumentos pasados a la función fit, por ello se llama que es un proceso de ajuste.

Cabe mencionar, que la función ***fit()*** nos devuelve un objeto ***History***, donde su atributo ***History.history*** es un registro de los valores de pérdida y métrica a lo largo de todo el entrenamiento almacenadas en sucesivas épocas. (Torres, 2020)

Los dos primeros parámetros de la función fit corresponden a los datos de entrenamiento y las etiquetas de estos (sus valores correctos de salida).

El argumento de entrada ***batch_size*** nos indica el tamaño de los lotes de entrada en el entrenamiento de la red, es decir el número de datos de entrada por lotes que usaremos para la actualización de los parámetros del modelo.

Por otra parte, el argumento ***Epochs*** nos indicará el número de veces que los datos de entrenamiento pasan por el entrenamiento de la red neuronal.

El proceso de ajuste consiste en encontrar los valores que optimicen el entrenamiento de la red neuronal convolucional.

En nuestro caso, la función fit ha recibido los siguientes parámetros de entrada.

```
fashion_model.fit(x,y, batch_size=100, epochs=4, validation_split=0.2)
```

Código 10. Fragmento de código. Entrenamiento. Función fit

Más adelante hablaremos del argumento *validation_split.*

- **Evaluación del modelo**

Como último paso dentro del proceso de entrenamiento de una red neuronal, es la evaluación.

Llegados a este punto, el modelo definido de red neuronal convolucional ya se ha entrenado. Ahora vamos a evaluar el entrenamiento dado un conjunto de datos de prueba, los cuales nunca ha visto. Esto se lleva a cabo con el método ***evaluate()***, el cual nos devuelve dos valores de salida, la pérdida y la métrica accuracy.

```
test_loss, test_acc = model.evaluate(x_test, y_test)
```

Código 11. Fragmento de código. Evaluación del modelo

Esta parte del código no pertenece al notebook del cual estamos hablando ya que no aplicaban la evaluación en él, pero para terminar de explicar el proceso lo he añadido (Torres, 2020)

4.2 Regularización - Técnica Dropout

En el siguiente estudio, volvemos a trabajar con el dataset fashion de MNIST. En el se configurará un modelo de Red Neuronal Convolucional más grande que el anterior, mediante la agregación de un mayor número de capas convolucionales.

Por otra parte, hará eso de una de las variadas técnicas que existen para controlar el sobreajuste de una red neuronal. Antes de profundizar en dicha técnica, vamos a hablar de ajuste, posteriormente, veremos cómo se implementa.

Ajuste. El ajuste de una Red Neuronal es la cuestión fundamental del aprendizaje, esta es la relación entre optimización y generalización. Por optimización entendemos que se refiere al proceso de ajuste de un modelo para obtener el mejor rendimiento en los datos de entrenamiento, por otra parte, la generalización es la capacidad que tiene un modelo entrenado de realizar su tarea correctamente en datos que nunca ha visto, es decir, como su nombre indica, generalizar los patrones aprendidos en datos nunca vistos.

El proceso de aprendizaje consiste en llegar a un equilibrio entre estos dos conceptos que permita a una red optimizar el aprendizaje y generalizar los patrones aprendidos en el aprendizaje.

Al principio del entrenamiento, estos dos conceptos van de la mano, de forma que a medida que va disminuyendo la función de pérdida de los datos de entrenamiento, en el cual se va optimizando los resultados, menor será también la función de pérdida de los datos de prueba, donde se observa la generalización de los patrones aprendidos, cuando esto ocurre se llama **underfitting**, esto indica que la red es deficiente, no ha procesado todos los patrones relevantes.

Tras un número de iteraciones en los datos de entrenamiento, la generalización deja de mejorar, comienza a degradarse y se dice que el modelo está en **overfitting** o sobreajuste, es decir está empezando a aprender patrones específicos de los datos de entrenamiento, debido a un exceso de optimización.

Estos dos conceptos los separa una fina línea. Es por ello que se dice que el ajuste se trata de un proceso de equilibrio entre la optimización y la generalización de los datos. (Chollet, 2018).

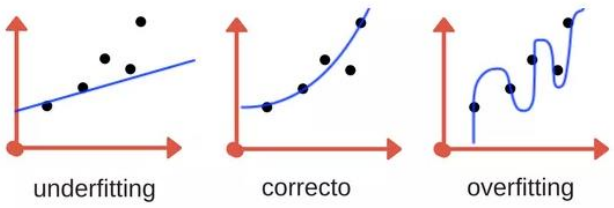


Figura 18. Ajuste del modelo

Existen varias técnicas para evitar el sobreajuste, las vamos a mencionar y más adelante conforme vayan saliendo hablaremos de ellas en profundidad. Aquí profundizaremos en la técnica de aumento de datos.

- Reducir el tamaño de la red: Reducir el número de parámetros a aprender por el modelo.
- Regularización de pesos: Limitamos la complejidad de la red, obligando a sus pesos a tomar valores pequeños, dando así regularidad en la distribución de los pesos.
- Aumento de datos. Hablaremos en profundidad más adelante de esta técnica
- Dropout. Vamos a hablar de ella.

Dropout es una de las técnicas de regularización de las redes neuronales. Esta consiste en poner de forma aleatoria a 0 un conjunto de valores de la salida de la capa de la red neuronal durante el entrenamiento.

Dado unos valores de entrada, la salida de una capa es un vector con un conjunto de valores. Al aplicar la técnica Dropout, de forma aleatoria un porcentaje de esos valores se pondrán a 0. Este porcentaje es indicado por nosotros y normalmente suele ser el 20% o 50% del total de los valores.

Esta técnica es eficaz ya que introducir ruido en los valores de salida de una capa permite que los patrones no significativos, aquellos que la red aprende ajustando en exceso, se rompan, solventando así el problema de sobreajuste. (Chollet, 2018).

Aplicar esta técnica mediante Keras es muy sencillo, ya que nos facilita la función ***Dropout***, en la cual como argumento de entrada le pasaremos el porcentaje de valores a poner a 0.

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

fashion_model_1 = Sequential()
fashion_model_1.add(Conv2D(16, kernel_size=(3, 3), activation='relu', input_shape=(img_rows, img_cols, 1)))
fashion_model_1.add(Conv2D(16, (3, 3), activation='relu', strides=2))
fashion_model_1.add(Flatten())
fashion_model_1.add(Dropout(0.5)
fashion_model_1.add(Dense(128, activation='relu'))
fashion_model_1.add(Dense(num_classes, activation='softmax'))

fashion_model_1.compile(loss=keras.losses.categorical_crossentropy,
              optimizer='adam',
              metrics=['accuracy'])

fashion_model_1.fit(x, y,
          batch_size=batch_size,
          epochs=3,
          validation_split = 0.2)
```

Código 12. Fragmento de código. Definición del modelo de red. Técnica Dropout

4.3 Validación cruzada - Max-pooling - Generadores

En el siguiente estudio, vamos a trabajar con el dataset de perros y gatos. Configuraremos un modelo de Red Neuronal Convolucional ajustado al problema, lo compilamos, ajuste de hiperparámetros y finalmente lo evaluaremos.

Muchos conceptos de los que aparecen ya hemos hablado de ellos por eso nos centraremos en lo que nos aporta diferente este notebook que son:

- Medidas de evaluación – Validación cruzada hold-out
- Uso de generadores.
- Capa de Max-Pooling

Estos conceptos nos han podido aparecer o lo hemos nombrado antes pero no hemos profundizado en ellos.

Tenemos un problema de clasificación binaria, donde nuestro dataset se encuentra formado por imágenes de perros y gatos. Nuestro objetivo es conseguir que nuestro modelo de red neuronal convolucional aprenda a clasificar un perro y un gato, dado una imagen nueva que vea. Para ello, los pasos a seguir, los cuales ya hemos explicado son los siguientes: Preprocesamiento de los datos, configuración de un modelo de red neuronal convolucional, monitorización del proceso de aprendizaje, entrenamiento y evaluación.

- **Validación cruzada**

Para entrenar una red neuronal comenzamos con el preprocesamiento del dataset. Originalmente, el entrenamiento y evaluación de una red neuronal se realiza sobre un conjunto de datos, es decir sobre nuestro dataset, este se encuentra organizado en dos carpetas, la carpeta de train y la de test, como ya hemos visto.

En la carpeta de train, se encuentran aproximadamente el 80% de los datos del dataset. En esta carpeta se encuentran los datos con lo que vamos a entrenar la red neuronal, es decir con lados con los que haremos los diferentes procesos de ajuste de hiperparámetros.

Por otra parte, la carpeta de test, se encuentran aproximadamente el 20% de los datos del dataset. En ella están los datos que nuestro modelo nunca ha visto y con el que realizaremos la evaluación si entrena correctamente o no.

Tras la organización y preprocesamiento del dataset, configuramos un modelo, monitorizamos el aprendizaje mediante el método ***compile()*** y por ejemplo como métrica indicamos ***accuracy*** (% de aciertos). Finalmente lo entrenamos, mediante el método ***fit()*** al cual le pasamos como argumentos de entrada ***train_x*** (imágenes de entrada) y ***train_y*** (etiquetas asociadas a las imágenes de entrada). Digamos por ejemplo que la métrica final del entrenamiento ha sido de un 80%, está bastante bien por lo que podemos hacer la evaluación a nuestro modelo mediante el método ***evaluate().***

Evaluamos nuestro modelo con el método ***evaluate()*** aquí la red ya no está adquiriendo conocimiento, sino, ponemos a prueba el conocimiento que ya ha adquirido con datos que nunca ha visto. Lo veremos más adelante en mayor profundidad.

Si el porcentaje de aciertos es más o menos un 10% que el porcentaje de aciertos que nos dió en el entrenamiento, has logrado entrenar correctamente un modelo de red neuronal convolucional. Si este porcentaje tiene un margen de error mayor de un 10% con respecto al porcentaje de aciertos del entrenamiento, esto nos indica que no ha entrenado bien.

Para solventar este problema sería ideal poder ir realizando una evaluación del proceso de entrenamiento, con datos distintos a los datos de entrenamiento. De aquí nace la idea de la evaluación del entrenamiento.

Hay diferentes métodos para la evaluación del entrenamiento del modelo como, por ejemplo, validación cruzada hold-out, validación cruzada k-fold, sin embargo, nosotros vamos a aplicar en este notebook la validación cruzada hold-out, aun siendo la validación cruzada k-folder la más utilizada, de esta daremos una breve explicación.

La validación cruzada hold-out, es una técnica que mide el comportamiento del modelo durante el entrenamiento. Esto nos permite conocer la evolución del entrenamiento, permitiendo así un mayor conocimiento para un posterior ajuste de los hiper parámetros.

Para llevar a cabo la validación hold-out, cogemos un porcentaje (normalmente suele ser el 20%) de los datos de entrenamiento. Este porcentaje de datos no serán usados para entrenar la red. Entrenaremos la red en cada iteración con el 80% restante de la carpeta de train y tras cada iteración mostraremos el 20% de los datos de validación (con los cuales la red no está siendo entrenada) y mediremos el accuracy obtenido sobre este conjunto de datos de validación.

El accuracy final es la media de los accuracy del conjunto total de iteraciones. Si este valor es similar en cada iteración significa que está entrenando correctamente. (Bagnato, 2019)

A diferencia de la validación cruzada hold-out, la validación cruzada k-fold lleva a cabo el mismo procedimiento, pero tiene una diferencia principal con la que gana mayor rendimiento el entrenamiento de la red neuronal.

En esta validación cruzada los datos de validación no son separados de los datos de entrenamiento, sino, el dataset es separado en conjunto de entrenamiento y conjunto de test. El conjunto de entrenamiento es dividido K subconjuntos, en los cuales aplica hold-out k veces, utilizando en cada iteración un subconjunto distinto para validar el modelo tras la época y utilizando los k-1 subconjuntos para el entrenamiento.

El método k-fold de validación cruzada obtiene mejores resultados que el método hold-out, ya que todos sus datos son utilizados para entrenar y validar, permitiendo así obtener resultados más representativos. Por otra parte, los datos mediante el método k-fold no se repiten por lo que favorece a evitar el sobreajuste. El método hold-out, se repite n veces según indiquemos sobre los mismos conjuntos de entrenamiento y validación. Finalmente, el método k-fold puede ser aplicado con conjunto de datos pequeños.

En nuestro caso, la organización del dataset se encuentra dividida en 3 carpetas train, validación y test. Con un porcentaje de 50, 25, 25 respectivamente. Originalmente el dataset estaba organizado en dos carpetas train y test, con un porcentaje de 75 y 25 respectivamente. Al hacer uso de la validación cruzada hold-out, un 25% de los datos son apartados como lo hemos explicado anteriormente. Estos datos de validación pueden ser divididos en otra carpeta como en nuestro caso, o indicar el porcentaje que queremos como datos de validación a la hora de entrenar. (Pérez, 2015)

Configuramos el modelo de red neuronal, el cual ya hemos explicado, aun así, lo veremos más adelante para introducir una nueva función a aplicar en las capas de la red neuronal.

Posteriormente compilamos el modelo y lo entrenamos, cuando realizamos el entrenamiento aplicamos la validación cruzada. Esto se realiza de varias formas, vamos a ver en primer lugar la aplicada en nuestro notebook de perros y gatos, donde los datos de validación se encuentran separados de los datos de train.

En primer lugar, realizamos el preprocesamiento de los datos. Tanto para los datos de train como de validación, reescalamos los datos a un tensor 3D en punto flotante, cuyos píxeles se encuentran normalizados entre [0, 1]. Este preprocesamiento de datos se realiza mediante el uso de ImagenDataGen, que veremos más adelante.

Mediante un generador, generamos lotes de datos de entrada a nuestra red, tanto para los datos de train como para los datos de validación, este concepto de generador, lo veremos tras esta explicación.

```
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,)

test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        #Directorio
        train_dir,
        #Redimentionamos las imagenes a 150x150
        target_size=(150, 150),
        #Número de imagenes en cada lote
        batch_size=32,
        #Como estamos en un problema de clasificación binaria, necesitamos etiquetas binarias
        class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
        #Directorio
        validation_dir,
        #Redimentionamos las imagenes a 150x150
        target_size=(150, 150),
        #Número de imagenes en cada lote
        batch_size=32,
        #Como estamos en un problema de clasificación binaria, necesitamos etiquetas binarias
        class_mode='binary')
```

Código 13. Fragmento de código. Preprocesamiento de datos.

Posteriormente, realizamos el entrenamiento con los datos de train y validación en el entrenamiento con los datos de validación. Para el entrenamiento hacemos uso de la función ***fit()*** como ya habíamos visto. En este caso como trabajamos con generadores (vamos a explicarlo ahora a continuación) hacemos uso de la función ***fit_generator()*** la cual funciona de la misma manera que la función ***fit()***, pero le entran lotes de datos.

En esta función ***fit_generator()*** pasamos como argumentos de entrada el directorio donde se encuentran los datos para el entrenamiento, en este caso es ***train_generator***. En el argumento ***steps_per_epoch*** indicamos el número total de muestras por época, en este caso es 100. En ***epochs*** indicamos el número de épocas o iteraciones que se realizarán y finalmente la validación cruzada. Mediante el argumento ***validation_data*** facilitamos los datos con lo que se realizarán la validación y mediante ***validation_step*** el número de lotes de validación que genera cuando termine una época, es decir cuando se vaya a realizar la validación.

```
entrenamiento_DA = CNN_DA.fit_generator(
        #Directorio
      train_generator,
        #Número total de muestras por epocas
      steps_per_epoch=100,
        #Número de iteraciones
      epochs=100,
        #Datos con los que realiza la validación
      validation_data=validation_generator,
        #Número de lotes de validaciñon que generará cuando termine una epoca
      validation_steps=50)
```

Código 14. Fragmento de código. Validación cruzada hold-out

La aplicación de la validación cruzada hold-out nos lo podemos encontrar en redes neuronales donde su dataset se encuentra solo dividido en train y test. A través de un argumento de la función fit al igual que antes (en este caso fit porque este trozo de

código no pertenece a este caso y no trabaja con generadores), este argumento es ***validation_split=0.2*** el modelo separará el 20% (0.2 x 100) de la carpeta train y lo usará para llevar a cabo la validación cruzada

```
fashion_model.fit(x,y, batch_size=100, epochs=4, validation_split=0.2)
```

Código 15. Fragmento de código. Validación cruzada hold-out, sin división del dataset en datos validación

- **Generador**

Keras nos proporciona la función ***imagen_data_generator***, la cual convierte los ficheros de imágenes en tensores preprocesados.

Esta función tiene un método llamado ***flow_from_directory()*** la cual nos devuelve lotes de imágenes en tensores preprocesados de las imágenes que se encuentran en el directorio dado junto con sus etiquetas correspondiente a cada subdirectorio. En nuestro caso las carpetas de train y validación tiene subdirectorios de perros y gatos cada una. Por lo que esta función nos dará de salida una etiqueta correspondiente al subdirectorio al que pertenezca cada imagen, siendo por ejemplo 0 gato y 1 perro.

Como podemos observar a la función ***ImageDataGenerator*** con el método ***flow_from_directory()*** le pasamos como argumento de entrada, el directorio donde se encuentra los datos mediante los cuales vamos a generar lotes de tensores preprocesados, es decir lotes de datos, como ***train_dir*** para el generador de la carpeta train y ***validation_dir*** para el generador de la carpeta validación. Indicamos mediante ***batch_size,*** el número de imágenes que queremos en cada lote generado. Y finalmente mediante ***class_mode*** el tipo de etiqueta de salida, en nuestro caso etiquetas binarias.

```
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,)

test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        #Directorio
        train_dir,
        #Redimentionamos las imagenes a 150x150
        target_size=(150, 150),
        #Número de imagenes en cada lote
        batch_size=32,
        #Como estamos en un problema de clasificación binaria, necesitamos etiquetas binarias
        class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
        #Directorio
        validation_dir,
        #Redimentionamos las imagenes a 150x150
        target_size=(150, 150),
        #Número de imagenes en cada lote
        batch_size=32,
        #Como estamos en un problema de clasificación binaria, necesitamos etiquetas binarias
        class_mode='binary')
```

Código 16. Fragmento de código. Generadores

- **Pooling**

En la configuración del modelo de red neuronal, como ya hemos visto, definimos una estructura secuencial mediante la función sequential() para posteriormente, añadir mediante el método add() en conjunto de capas que consideremos, en este caso capas convolucionales. Estas capas convolucionales tienen un kernel de dos dimensiones 3x3 y una función de activación Relu, cada capa convolucional difiere en el número de filtros aplicados en esa convolución, teniendo el primer 32 filtros y el segundo 64 como podemos observar.

```
from keras import layers
from keras import models
from keras.utils.vis_utils import plot_model

CNN = models.Sequential()
CNN.add(layers.Conv2D(32, (3, 3), activation='relu',
                      input_shape=(150, 150, 3)))
CNN.add(layers.MaxPooling2D((2, 2)))
CNN.add(layers.Conv2D(64, (3, 3), activation='relu'))
CNN.add(layers.MaxPooling2D((2, 2)))
CNN.add(layers.Conv2D(128, (3, 3), activation='relu'))
CNN.add(layers.MaxPooling2D((2, 2)))
CNN.add(layers.Conv2D(128, (3, 3), activation='relu'))
CNN.add(layers.MaxPooling2D((2, 2)))
CNN.add(layers.Flatten())
CNN.add(layers.Dense(512, activation='relu'))
CNN.add(layers.Dense(1, activation='sigmoid'))

plot_model(CNN, to_file='CNNCatsDogs_plot.png', show_shapes=True, show_layer_names=True)
```

Código 17. Fragmento de código. Definición modelo de red. Capa pooling.

Tras cada capa de convolución añadimos una capa de pooling para reducir el tamaño de la próxima capa de neuronas.

La capa de pooling realiza una simplificación de la información que se encuentra en la capa que convolución inmediatamente anterior y devuelve una versión reducida de la misma con las características más relevantes.

Hay varios tipos para realizar la operación de pooling. Nosotros sin embargo usaremos la más habitual llamada max-pooling. Max-pooling trabaja almacenando los valores máximos por cada grupo de pixeles que recorra.

Al igual que el Kernel, para max-pooling definimos una ventana del tamaño que queramos, esta ventana recorre el conjunto de neuronas que forma la capa quedándose con el valor más alto en cada agrupación. A diferencia del desplazamiento de la ventana del kernel, la ventana max-pooling se desplaza de forma proporcional a su dimensión.

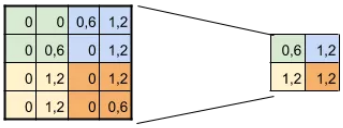


Figura 19. Max-Pooling

Esta capa de max-pooling se añade mediante MaxPooling2D(). En nuestro caso, la ventana es de 2D, esta tiene una dimensión de 2x2, por lo que se desplazará de izquierda a derecha y de arriba a abajo cada dos píxeles o neuronas. Reduciendo el tamaño de la capa a la mitad. Esto se aplica tras cada convolución, en nuestro caso todas las capas de max-pooling tiene la misma configuración de ventana. (Bagnate, 2018)

```
CNN.add(layers.MaxPooling2D((2, 2)))
```

Código 18. Fragmento de código. Capa Max-Pooling

4.4 Modelos Pre-entrenados

Mediante este tutorial aprenderemos a hacer uso de un modelo pre-entrenado, en este caso, del modelo pre-entrenado ResNet50. Al no entrenar un modelo como tal, no haremos uso de grandes cantidades de funciones que nos proporciona Keras para ello, sin embargo, las veremos más adelante.

ResNet (Residual Network), es un modelo de Red Neuronal Clásica cuya arquitectura es utilizado como soporte para labores de visión computacional. Las Redes extremadamente profundas son difíciles de entrenar debido al problema de la fuga de gradiente, sin embargo, ResNet destaca debido a sus satisfactorios resultados al entrenar este tipo de redes. Esto es debido a la aplicación del concepto Omisión de conexión.

Este concepto se fundamenta en el aumento del número de capas de la red neuronal introduciendo una conexión residual, la cual tiene una capa identidad. Esto permite que el gradiente pase a la siguiente capa directamente, mejorando el proceso de aprendizaje. Esto se realiza añadiendo una entrada a la salida del bloque de convolución, omitiendo como dice su nombre la conexión anterior. (Utrera, 2018)

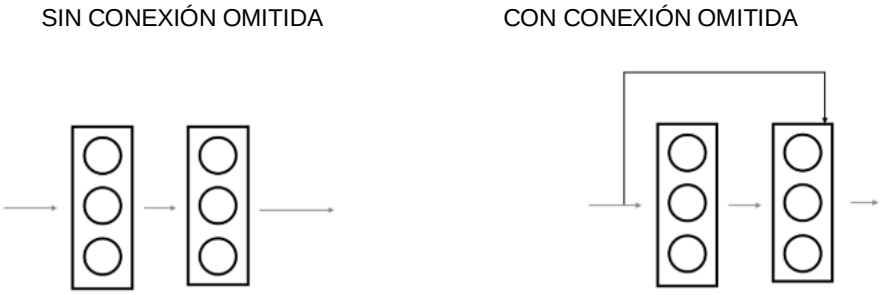


Figura 20. ResNet - Concepto de omisión de conexión

Esta arquitectura obtiene mejores resultados ya que con esta se elimina la fuga de gradiente con la transmisión de la información a capas posteriores, mediante el camino de acceso directo que facilita su estructura y permite que el modelo aprenda una función identidad que garantice que las capas superiores como inferiores funcionarán con la misma destreza.

ResNet-50, nace de la combinación de los bloques identidad y convoluciones. La arquitectura de ResNet-50 está formada por 5 etapas, cada una con un bloque de convolución e identidad, cada bloque con 3 capas de convolución. (Dewivedi, 2019)

Su arquitectura quedaría de la siguiente manera:

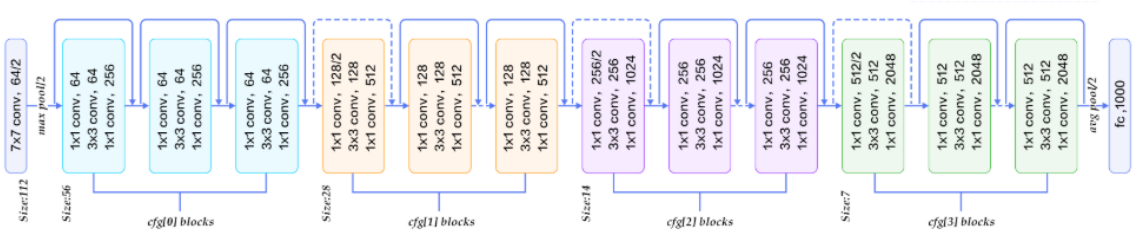


Figura 21. Arquitectura ResNet50

Ya que conocemos la arquitectura de ResNet, porque destaca y su funcionamiento, vamos a ver como cargar dicho modelo para su uso.

- En primer lugar, importamos ***preprocess_input()*** de ResNet-50, el cual procesa un tensor o una matriz de Numpy que codificará el lote de imágenes con las que trabajaremos

```
from tensorflow.python.keras.applications.resnet50 import preprocess_input
```

Código 19. Fragmento de código. Preprocesador de tensor

- Mediante ***ResNet50()*** creamos las instancias de la arquitectura que vamos a usa, en este caso la arquitectura del modelo ResNet-50

```
my_model=ResNet50(weights='./directoriopesos')
```

Código 20. Fragmento de código. Importación arquitectura ResNet50

En ***weights*** indicamos la ruta al archivo de los pesos con los que vamos a inicializar nuestro modelo. En el caso de indicar *none,* inicializamos el modelo con pesos aleatorios.

Este tiene otros muchos otros argumentos como:

- ***include_top,*** para indicar si incluimos la capa absolutamente conectada en el modelo original
- ***input_tensor***, tensores para usar como entradas de imágenes para el modelo
- ***input_shape***, tupla para indicar las dimensiones
- ***pooling*****,** modo de agrupación en el caso de que *include_top = false*.
- ***class*****,** número de clases en la que clasifica el modelo.

- Leemos y procesamos los datos

Previamente, definimos la función ***read_and_prep_images(),*** en la que determinamos las dimensiones en las que se restablecen las imágenes, las convertimos en una matriz de Numpy mediante np.array y sobre esa aplicamos la función preprocess_input, ya mencionado con anterioridad, adapta las imágenes al formato que requiere el modelo.

```
test_data = read_and_prep_images(img_paths)
```

```
image_size = 224

def read_and_prep_images(img_paths, img_height=image_size, img_width=image_size):
    imgs = [load_img(img_path, target_size=(img_height, img_width)) for img_path in img_paths]
    img_array = np.array([img_to_array(img) for img in imgs])
    output = preprocess_input(img_array)
    return(output)
```

Código 21. Fragmento de código. Lectura y preprocesamiento de datos

- Obtenemos las predicciones del modelo mediante el método ***predict()***

```
preds = my_model.predict(test_data)
```

Código 22. Código 21. Fragmento de código. Predicciones del modelo

- Mediante ***decode_predictions()*** decodificamos la predicción del modelo. Esto se realiza extrayendo las mayores probabilidades de cada imagen.

```
most_likely_labels = decode_predictions(preds, top=3)
```

Código 23. Fragmento de código. Descodificación de predicciones

En nuestro caso indicamos que nos de las 3 probabilidades más altas para cada imagen procesada.

4.5 Transferencia de aprendizaje.

En este capítulo del curso de Keras sobre Deep Learning, haremos uso de la transferencia de aprendizaje de ImageNet con arquitectura ResNet50, esta consiste en transferir el aprendizaje de un problema a otro, siendo este aprendizaje adaptado al problema final.

Haremos uso de parte del aprendizaje de ResNet50, para satisfacer el objetivo de clasificar una imagen si está en posición vertical y que imágenes se encuentran en otra posición.

Recordemos que la estructura de una Red Neuronal está formada por capas organizadas de forma jerárquica donde el aprendizaje de patrones de representación se va haciendo más complejo y significativos en las capas superiores que finalizan con la clasificación. Toda Red Neuronal tienen similares patrones visuales de bajo nivel. Esto permite la transferencia de aprendizaje mediante la reutilización de estas capas de patrones de bajo nivel.

En primer lugar, escogeremos la arquitectura base de ResNet50, se eliminará la capa de salida de dicha arquitectura y sobre ella añadiremos un conjunto de capas que nosotros definiremos adaptadas a nuestro problema y objetivo. Una vez que tengamos nuestro modelo de Red Neuronal formado por unas capas transferidas del modelo escogido y unas capas finales añadidas en función de nuestro problema, el procedimiento consta de dos etapas:

Primera etapa: Congelamos los parámetros de las capas transferidas del modelo de entrenamiento ResNet50. Aprovechando, como habíamos indicado ese aprendizaje adquirido en el entrenamiento con Imagenet.

Segunda etapa: Entrenamos el modelo al completo (Durán, 2019)

Vamos a verlo de forma más detallada.

- Definimos un modelo Secuencial, mediante la función ***Sequential()***. Esta está formada por una secuencia de capas, es decir, una pila de capas simples.

```
from tensorflow.keras.models import Sequential
my_new_model = Sequential()
```

Código 24. Fragmento de código. Modelo de red secuencial

- A este modelo definido (*my_new_model)* le agregamos una pila de capas del modelo pre-entrenado *ResNet50*. Esto lo llevamos a cabo mediante ***add(ResNet50()).*** En esta transferencia de aprendizaje indicamos diversos parámetros como:
  - ***include_top = false***, excluimos las últimas capas que realizan la predicción del modelo que estamos transfiriendo, quedándonos así con las capas cuyo patrón de aprendizaje no son tan significativos.
  - ***weigths***, en este caso le pasamos la ruta de un archivo cuyos pesos no están incluidos.
  - ***pooling = 'avg'***, indicamos que en la última capa del bloque convolucional se aplicará el promedio global, siendo la salida un tensor 1D.

```
from tensorflow.keras.applications import ResNet50

resnet_weights_path = '../input/resnet50/resnet50_weights_tf_dim_ordering_tf_kernels_notop.h5'
my_new_model.add(ResNet50(include_top=False, pooling='avg', weights=resnet_weights_path))
```

Código 25. Fragmento de código. Trasnferencia de aprendizaje ResNet50 Imagnet

- Añadimos una capa densamente conectada con dos nodos. Aplicamos una función de activación *softmax* en la capa de salida, la cual nos permite convertir los resultados en probabilidades.

```
from tensorflow.keras.layers import Dense, Flatten, GlobalAveragePooling2D

num_classes = 2
my_new_model.add(Dense(num_classes, activation='softmax'))
```

Código 26. secuencialCódigo 25. Fragmento de código. Transferencia de aprendizaje. Capa densa

- Congelamos el aprendizaje adquirido en el entrenamiento de Imagnet con ResNet50. Aprovechando así dicho aprendizaje de bajo nivel. Esto se puede llevar a cabo con ***trainable = false***, indicando que los pesos en dichas capas no se actualicen durante el entrenamiento.

```
my_new_model.layers[0].trainable = False
```

Código 27. Fragmento de código. Congelación de capas de la transferencia de aprendizaje

- Compilamos el modelo, mediante el método ***compile ()***. En la compilación especificamos cómo queremos que nuestro modelo realice la actualización de los pesos. En este caso con una función de pérdida ***categorical_crossentropy***,

un optimizador de descenso de gradiente estocástico ***sgd*** y con una métrica ***accuracy***.

```
my_new_model.compile(optimizer='sgd',
                     loss='categorical_crossentropy',
                     metrics=['accuracy'])
```

Código 28. Fragmento de código. Compilamos el modelo

- Finalmente, tras haber realizado una transferencia de aprendizaje y tener el modelo de Red Neuronal definido y compilado, solo hay que entrenarlo mediante los datos de train haciendo uso de la función ***fit()***.

```
from tensorflow.keras.applications.resnet50 import preprocess_input
from tensorflow.keras.preprocessing.image import ImageDataGenerator

image_size = 224
data_generator = ImageDataGenerator(preprocess_input)

train_generator = data_generator.flow_from_directory(
                                    directory=../input/dogs-gone-sideways/images/train,
                                    target_size=(image_size, image_size),
                                    batch_size=10,
                                    class_mode='categorical')

validation_generator = data_generator.flow_from_directory(
                                    directory="../input/dogs-gone-sideways/images/val",
                                    target_size=(image_size, image_size),
                                    class_mode='categorical')

fit_stats = my_new_model.fit_generator(train_generator,
                                   steps_per_epoch=22,
                                   validation_data=validation_generator,
                                   validation_steps=1)
```

Código 29. Fragmento de código. Entrenamiento del modelo con transferencia de aprendizaje.

4.6 Aumento de datos

En el siguiente capítulo hablaremos del aumento de datos. Esto lo aplicaremos en el problema anterior de la clasificación automática de una imagen girada. Recordemos que este modelo de red neuronal definido tenía una transferencia de aprendizaje de Imagenet con arquitectura ResNet50

El aumento de datos es una de las técnicas para mitigar el desbalanceo de los datos y el sobreajuste de la red. En primer lugar, vamos a ver que son esos dos conceptos por los cuales surge esta técnica.

El **desbalanceo de datos** ocurre cuando nuestro dataset está formado por un conjunto de datos divididos en distintas clases, donde las clases posee un número de datos muy dispares, destacando una clase minoritaria en el dataset.

Entrenar una Red Neuronal con un dataset desbalanceado, conlleva un mal entrenamiento, ya que la red no ha “visto” y tampoco ha podido procesar tanta información de la clase minoritaria, impidiendo así que llegue a obtener el mismo nivel de profundidad de patrones de dicha clase.

Viéndolo con un ejemplo, podemos tener un dataset de 1000 imágenes de animales, donde 990 son de Loros y 10 son de Águilas. La red neuronal no conseguirá diferenciar una clase de otra, respondiendo siempre que la imagen de entrada es un Loro. Esto además nos lleva a tener una mala confianza en la métrica accuracy ya que en este caso tendría un acierto del 99%, lo que es bastante bueno, pero realmente no lo es ya que la clase minoritaria no ha sabido clasificar. (Bagnato, 2019)

Ahora que ya entendemos lo que se pretende mitigar con la técnica del Aumento de datos, vamos a hablar de ella y cómo implementarla.

El aumento de datos es una técnica que se suele aplicar en modelo con dataset pequeños. Esta técnica consiste en generar más datos de entrenamiento a partir de nuestro dataset, esto se realiza mediante un conjunto de transformaciones en las imágenes de nuestro dataset, produciendo nuevos datos.

Esta técnica mitiga el problema de desbalanceo de datos, aplicando el aumento de datos en la clase minoritaria produciendo un aumento de dicha clase, suprimiendo el desbalanceo. Por otra parte, se aplica para solventar el problema de sobreajuste, ya que permite que, durante el entrenamiento debido al aumento de datos, el modelo nunca vea exactamente la misma imagen. (Torres, 2020)

En keras esto se puede hacer configurando un conjunto de transformaciones que se ejecutan de forma aleatorias sobre las imágenes que lee el generador ***ImageDataGenerator().*** Esas transformaciones se indican en los argumentos de la función, en función de las modificaciones que queramos realizar.

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

data_generator_with_aug = ImageDataGenerator(preprocessing_function=preprocess_input,
                                             horizontal_flip = True,
                                             width_shift_range = 0.1,
                                             height_shift_range = 0.1)
```

Código 30. Fragmento de código. Aumento de datos

Hay que recalcar que esta técnica solo se aplica en el proceso de entrenamiento, únicamente sobre las imágenes de Train y validación en todo caso.

Posteriormente, mediante el método de ***ImagenDataGeneratos Flow_from_directory()*** especificamos el directorio de los datos con los que queramos generar lotes de datos con esas transformaciones.

```
train_generator = data_generator_with_aug.flow_from_directory(
        directory = '../input/dogs-gone-sideways/images/train',
        target_size=(image_size, image_size),
        batch_size=12,
        class_mode='categorical')


validation_generator = data_generator_no_aug.flow_from_directory(
        directory = '../input/dogs-gone-sideways/images/val',
        target_size=(image_size, image_size),
        class_mode='categorical')

my_new_model.fit_generator(
        train_generator,
        epochs = 3,
        steps_per_epoch=19,
        validation_data=valid_generator)
```

Código 31. Fragmento de código. Directorio al que le realizaremos el aumento de datos y preprocesamiento de los datos

Finalmente, mediante la función fit_generator, debido a que estamos trabajando con generadores de lotes de imágenes, esta funciona igual que la función fit. Entrenamos nuestro modelo

```
my_new_model.fit_generator(
        train_generator,
        epochs = 3,
        steps_per_epoch=19,
        validation_data=valid_generator)
```

Código 32. Fragmento de código. Entrenamiento del modelo

Hasta ahora, hemos visto alguno de los estudios más relevantes que hemos trabajado, profundizando así en las diferentes técnicas que han ido apareciendo. Además de esto hemos trabajado con el problema “hola mundo” del aprendizaje profundo llamado notebook MNIST reconocimiento de dígitos. Por otra parte, dimos un curso online de Webinar de dos sesiones con Miguel Ángel Martínez del Amor y finalmente estudios de notebook de tensorflow con apoyo visual. Estos otros estudios solo han sido mencionados ya que no tienen nada más que aportar de lo que hemos visto.

5. Reto Kaggle

El reto Kaggle al que nos enfrentamos, lo hemos llamado Detección de neumonía imágenes de pulmón mediante Aprendizaje profundo. Este estudio parte del propósito de optimizar el problema temporal que presentan los facultativos neumólogos en la detección de patologías como la neumonía en imágenes de pulmón.

Para ello hacemos uso del campo de la Inteligencia artificial, mediante la aplicación del modelo computacional perteneciente al Aprendizaje profundo, las redes neuronales convolucionales.

Este reto se centra en el campo de patologías pulmonares, por lo que vamos a dar una breve introducción de que son los pulmones y en que se basa la patología de la neumonía.

Los pulmones son un órgano esencial, localizados en la cavidad toráxica, encargados de mantener la oxigenación de la sangre. Estos están formados por unas cavidades por donde pasa el aire, llamada bronquios y unos pequeños sacos al final de dichos bronquios donde se produce el intercambio de gases, llamado alveolos.

Al respirar el aire entra en el cuerpo por la nariz o la boca y llega hasta los pulmones, donde a través de los bronquios principales llega a los alveolos. En estos últimos, los alveolos, es donde se realiza el intercambio de gases, pasando el oxígeno respirado al torrente sanguíneo y el dióxido de carbono del torrente sanguíneo es pasado al alveolo, donde lo expulsamos mediante la expiración.

La Neumonía es una infección del pulmón que produce una inflamación del tejido pulmonar, provocando una alteración en el intercambio de gases. Esta alteración es debido a que, en el proceso de la inspiración, los alveolos no se llenan del oxígeno respirado sino de pus.

La Neumonía se detecta por medio de una radiográfica del tórax. Si el paciente presenta dicha patología, se podrá observar en la radiografía del tórax unas manchas pulmonares u opacidades blanquecinas, no propias del pulmón. (López, 2019)

La aplicación de Redes Neuronales convolucionales nos proporcionaría una detección clara debido a su alta capacidad de aprendizaje de patrones locales, detectando dichas opacidades blanquecinas propias de pulmones con neumonía y permitiéndonos una clasificación correcta de los pulmones que presentan dicha patología y los que no.

En el siguiente capítulo desarrollaremos una serie de modelos de redes neuronales convolucionales partiendo de la configuración de parámetros e hiperparámetros mejor obtenida. Esta estructura será desarrollada, entrenada y evaluada en el modelo 1, y posteriormente en el modelo 2 y modelo 3, aplicaremos a la configuración del modelo 1 y las diversas técnicas para solventar los puntos débiles de nuestro conjunto de datos y modelo, que son sobreajuste y desbalanceo de datos respectivamente, mediante las técnicas aprendidas en el estudio previo del aumento de datos y técnica de regularización de dropout. En todos estos modelos aplicaremos la validación cruzada

de hold-out para conocer en mayor profundidad la evolución del entrenamiento y conocer el proceso de ajuste de este.

5.1 Definición del problema.

El objetivo de nuestro trabajo es conferir de inteligencia artificial a una máquina a través del aprendizaje profundo, mediante la estructura de redes neuronales convolucionales. Su objetivo es llevar a cabo la labor de clasificar imágenes de pulmón en dos categorías, si dicha imagen parece neumonía o no.

Para ello Kaggle nos facilita un dataset con 5856 imágenes de pulmón, dividida de dos clases, imágenes de pulmón con neumonía y sin neumonía.

Nos enfrentamos a un tipo de problema de clasificación binaria. Esto determinará ciertas características de nuestro aprendizaje como la salida que tendrá nuestra red neuronal a definir, como el tipo de parámetros que determinaremos en la monitorización del aprendizaje como la función de pérdida.

A la red neuronal que entrenaremos para alcanzar nuestro objetivo, le pasaremos como parámetros de entrada las distintas imágenes de pulmón que posee nuestro dataset y una etiqueta con la clase correspondiente cada una, siendo la clase 0 imágenes que padecen neumonía y la clase 1 imágenes de no parecen neumonía. Estas etiquetas las extraemos mediante el uso de generadores a la hora que generemos nuestros lotes de datos de entrada.

El dataset nos proporciona unos datos muy informativos por lo que creo que podemos llevar a cabo el correcto entrenamiento de la red neuronal, sin embargo, disponemos de una dataset pequeño, cuyos datos se encuentran desbalanceados, esto nos puede suponer un problema a la hora del correcto entrenamiento de la red neuronal convolucional. Lo veremos a continuación en nuestro modelo de trabajo.

El problema al que nos enfrentamos está basado en la detección de la enfermedad de neumonía. Esta enfermedad se encuentra bastante asentada como para tener la confianza de que las predicciones futuras que llevemos a cabo guardan el mismo comportamiento de patrones básicos. Esto nos permite tener un problema con solución.

5.2 Medida de éxito

Como hemos dicho nos encontramos en un problema que nos permite tener solución debido a la estabilidad del conocimiento de la enfermedad en concreto. Por otra parte, nos encontramos con un dataset cuyas condiciones no nos favorece mucho. Es por ello que debemos definir unas medidas de éxito, las cuales nos permitirán un control y seguimiento de nuestro modelo.

Como medida de éxito, mediante la cual podremos ver el comportamiento de nuestro modelo y determinar si los resultados han sido satisfactorios o no, utilizaremos el porcentaje de aciertos (accuracy) que ha tenido nuestro modelo de red neuronal. Como ya explicamos con anterioridad debido al desbalanceo de datos que tienen nuestro dataset, donde 20% corresponden a la clase 1 y el 80% corresponde a la clase 0, para determinar el éxito de nuestro modelo este debe tener un porcentaje de accuracy superior al 80% ya que determinaría que ha captado patrones para clasificación de la clase minoritaria.

Combinado con la medida de éxito accuracy que nos permite observar el correcto comportamiento de nuestro modelo no solo en su evaluación sino también en su entrenamiento, haremos uso en la evaluación de la matriz de confusión, mediante la

cual determinaremos el éxito de nuestra red o no, ya que permite la visualización de la ejecución del modelo, no solo mostrándonos el accuracy sino también otras métricas como la precisión y la sensibilidad.

5.3 Protocolo de evaluación

Ahora que conocemos nuestro problema en profundidad, somos conscientes de las posibles dificultades que nos podemos encontrar, es conveniente establecer un protocolo de evaluación más exhaustivo durante el entrenamiento, lo cual nos permitirá observar la evolución del entrenamiento.

Al encontrarnos con un dataset pequeño, vamos a hacer uso del protocolo de evaluación validación cruzada hold-out durante el entrenamiento. Esto nos permitirá conocer el desarrollo de nuestro modelo por cada época.

Tras hacernos una idea de nuestro problema, sus pros y sus contras y contextualizar un poco en él vamos a comenzar su implementación.

5.4 Organización del dataset

Nuestro dataset está formado por 5856 imágenes de pulmón en formato JPGE, dividido en 3 carpetas, la carpeta train, donde se encuentran el conjunto de datos con los que realizaremos el entrenamiento del modelo configurado de red neuronal, la carpeta validación, donde se encuentra el conjunto de datos que usaremos para conocer la correcta evolución del entrenamiento y finalmente la carpeta test cuyos datos usaremos para evaluar el modelo una vez entrenado. Estas carpetas se encuentran divididas en dos clases con conjunto de datos diferentes, en la clase 0 corresponde a las imágenes de pulmón con neumonía y en la clase 1 las imágenes de pulmón sin neumonía.

Este es un dataset pequeño el cual se encuentra desbalanceado, el 27% del total corresponden a imágenes sin neumonía y el 73% a imágenes con neumonía.

Las carpetas se encuentran divididas con porcentajes muy dispares. Para las imágenes sin neumonía, la carpeta de test tiene un 14.8%, el 0.5% para la carpeta de validación y finalmente un 84.7% para la carpeta de train. Igual pasa para las imágenes de pulmón con neumonía, la carpeta de test tiene un 9.2%, el 0.2% para la carpeta de validación y finalmente para la carpeta de train un 90.6%.

Como podemos observar, el dataset se encuentra muy mal organizado en el conjunto de carpetas en las que está subdividido. Por ello vamos a organizarlo de forma que tanto para las imágenes de pulmón con neumonía o sin ellas, la división en las diferentes carpetas de train, validación y test, sean de un 60%, 20% y 20% respectivamente.

Tras haber organizado nuestro dataset, comenzamos con el preprocesamiento de los datos.

5.5 Preprocesamiento de los datos

Para realizar un correcto preprocesamiento de datos debemos conocer diferentes aspectos de nuestras imágenes como, las dimensiones, el número de canales que posee y finalmente el tipo de datos (unit8, int8..etc) para re-escalarlos posteriormente.

En primer lugar, vamos a visualizar las imágenes que tenemos. Para ello mediante el método ***listdir()*** del módulo ***os***, listamos el conjunto de imágenes sé que encuentran en el directorio facilitado como argumento de entrada y mediante el método ***path.join*** unimos el nombre de la imagen que se encuentra en la lista al directorio completo en el que se encuentra. Finalmente almacenamos en ***imagenes_train_neu*** el nombre de cada imagen junto con su directorio en una lista.

Posteriormente hacemos uso de ***random.randint()*** para que nos escoja un valor entre el rango dado que se almacenará en x. Almacenamos en ***imagen_neumonia*** la imagen que se encuentre en dicho valor de la lista ***imagenes_train_neu***.

Finalmente, mediante cv2.imread() leemos la imagen ***imagen_neumonia*** y mediante la librería ***Pyplot*** la visualizamos. Esto lo llevamos a cabo para las dos clases que posee nuestro dataset.

Importamos las librerías necesarias para este apartado.

```
from skimage.io import imread
from skimage.transform import resize
from keras.preprocessing.image import ImageDataGenerator,load_img, img_to_array
import cv2
import random
import imageio
```

Código 33. Fragmento de código. librerías importadas para el preprocesamiento de datos

Imagen del pulmón con neumonía.

```
imagenes_train_neu = [os.path.join(new_train_neu, fname) for fname in os.listdir(new_train_neu)]
x = random.randint(1, len(imagenes_train_neu))
imagen_neumonia = imagenes_train_neu[x]
print(imagen_neumonia)

img_neu = cv2.imread(imagen_neumonia)
plt.imshow(img_neu, cmap='gray')
```

Código 34. Fragmento de código. Visualización imagen de pulmón con neumonía

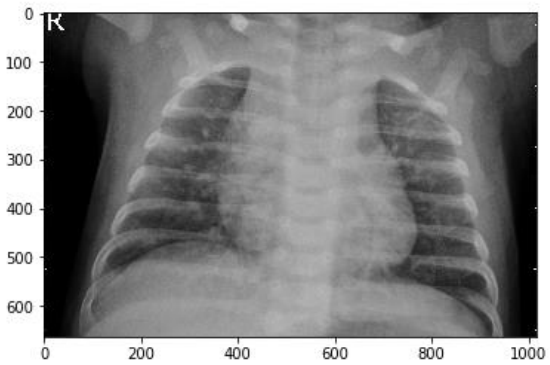


Figura 22. Pulmón que padece Neumonía

Imagen de pulmón sin neumonía.

```
imagenes_train_nor = [os.path.join(new_train_nor, fname) for fname in os.listdir(new_train_nor)]
x = random.randint(1, len(imagenes_train_nor))
imagen_normal = imagenes_train_nor[x]
print(imagen_normal)

img_nor = cv2.imread(imagen_normal)
plt.imshow(img_nor, cmap='gray')
```

Código 35. Fragmento de código. Visualización de imagen de pulmón sin neumonía

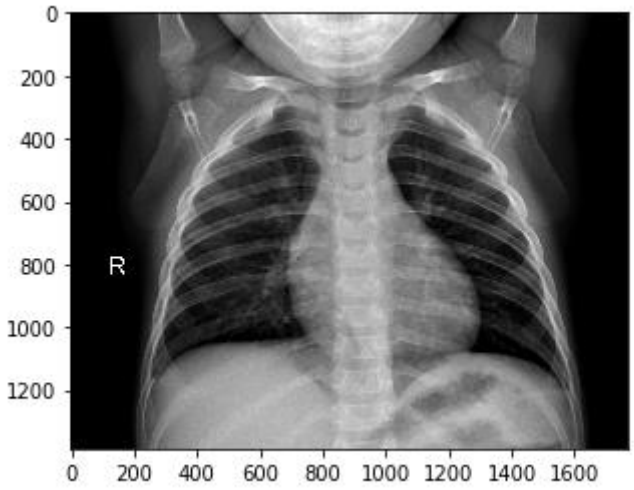


Figura 23. Pulmón que no padece neumonía

Sobre estas dos imágenes imagen_nor y imagen_neu vamos a estudiar de forma breve el formato haciendo uso del método ***type(),*** su dimensión y número de canales mediante el método ***shape()*** y tipo de datos mediante el método ***dtype().*** Para conocer más en profundidad nuestros datos y hacer posteriormente un correcto preprocesamiento.

```
#formato
print("La imagen img_neu es de tipo:", type(img_neu))
print("La imagen img_nor es de tipo:",type(img_nor))

#Dimension
print("La imagen img_neu tiene una dimensión:", img_neu.shape)
print("La imagen img_nor tiene una dimensión:", img_nor.shape)

#tipo de datos que hay que redimensionar
print("El tipo de dato de la imagen img_neu es", img_neu.dtype)
print("El tipo de dato de la imagen img_nor es", img_nor.dtype)
```

Código 36. Conocer los datos con los que vamos a trabajar

```
La imagen img_neu es de tipo: <class 'numpy.ndarray'>
La imagen img_nor es de tipo: <class 'numpy.ndarray'>
La imagen img_neu tiene una dimensión: (688, 1192, 3)
La imagen img_nor tiene una dimensión: (1816, 1920, 3)
El tipo de dato de la imagen img_neu es uint8
El tipo de dato de la imagen img_nor es uint8
```

Como podemos observar ambas son una matriz 3D con dimensiones diferentes. Estás posee 3 canales con dimensión 3 en el eje de profundidad, es decir estamos trabajando con imágenes cuyo canal de profundidad está formado por 3 canales, rojo, verde y azul como ya hemos comentado con anterioridad Finalmente el tipo de dato es

Uint8, esto es fundamental para saber cómo reescalaremos los valores posteriormente. Uint8 se encuentra en el rango de valores [0, 255].

Ahora que conocemos el conjunto de datos con el que vamos a trabajar, vamos a comenzar el preprocesamiento. Como ya sabemos, para alimentar la red las imágenes tienen que ser formateadas para ser convertidos en tensores de punto flotante. Para ello debemos leer los ficheros, decodificar el formato JPGE en una matriz de RGB, convertir la matriz en tensores de punto flotante y finalmente reescalar los valores de [0, 255] ya que el tipo de dato es Uint8 a [0,1].

Para llevarlo a cabo, hacemos uso de la función ***ImageDataGenerator***() que nos proporciona Keras. Esta función es un generador que de forma automática convierte los ficheros en tensores preprocesados y nos devuelve un conjunto de lote de datos los cuáles serán las entradas a nuestro modelo de red neuronal.

Preprocesaremos nuestros datos en tensores de punto flotante y normalizados entre [0, 1]. Estos datos tendrán una dimensión de 64x64, organizados en lotes de 32 imágenes por lote

```
#SIN Aumento de datos, reescalar en punto flotante y normalización
train_datagen = ImageDataGenerator(rescale = 1./255)

#Reescalar en punto flotante y normalización
test_datagen = ImageDataGenerator(rescale=1./255)

ejemplos_train = 3512
ejemplos_validacion = 1170
batch = 32

train_generator = train_datagen.flow_from_directory(
        #Directorio
        new_train,
        #Redimension de las imágenes a 64x64
        target_size=(64, 64),
        #Número de imágenes por lote de datos
        batch_size=batch,
        # Etiquetas binarias
        class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
        #Directorio
        new_val,
        #Redimension de las imágenes a 64x64
        target_size=(64, 64),
        #Número de imágenes por lote de datos
        batch_size=batch,
        #Etiquetas binarias
        class_mode='binary')

test_generator = test_datagen.flow_from_directory(
    #Directorio
    new_test,
    #Redimension de las imágenes a 64x64
    target_size = (64, 64),
    #Número de imágenes por lote de datos
    batch_size = batch,
    #Etiquetas binarias
    class_mode = 'binary')
```

```
Found 3512 images belonging to 2 classes.
Found 1170 images belonging to 2 classes.
Found 1172 images belonging to 2 classes.
```

Código 37. Fragmento de código. Preprocesamiento de datos y generación de lotes de imágenes

Como resultados obtendremos por cada generador paquetes de imágenes de (32, 64, 64, 3), es decir por cada lote tendremos 32 imágenes con una dimensión de 64x64 con 3 canales y un paquete de etiquetas de tamaño 32.

Ya tenemos los tensores de datos de entrada y sus etiquetas correctamente, por lo que podemos comenzar a configurar nuestro modelo.

5.6 MODELO 1

5.6.1 Configuración del modelo de red neuronal convolucional 1

El modelo de red neuronal convolucional diseñado para cumplir nuestro objetivo, tiene una estructura secuencial adquirida mediante la función ***Sequential(),*** en la que todas las capas se disponen en una pila.

El conjunto de capas añadidas mediante el método ***add()*** las cuales conforman este modelo, está formado por dos capas convolucionales ***Conv2D*** con 32 filtros por cada convolución. Cada convolución tiene configurado un ***Kernel*** de dos dimensiones 3x3 y una función de activación ***Relu.***

A cada capa convolucional le sigue una capa de ***MaxPooling2D***, esta capa como ya hemos visto reduce el tamaño de la salida de la capa anterior, en este caso la reducción se aplica a la mitad, determinado por su tamaño pool_size 2x2.

Tras dos capas de convoluciones y dos capas de pooling, obtenemos como salida un tensor 3D. Como ya hemos visto debemos aplanar esta salida para poder conectar esta última capa a una arquitectura de red convolucional tradicional, para ello hacemos uso de la función ***Flatten().***

Posteriormente añadimos una capa oculta tradicional densamente conectada ***Dense()***, con 128 neuronas y una función de activación ***Relu***.

Finalmente, al tratar con un problema de clasificación binaria, nuestra salida a de la red neuronal convolucional está formada por una única neurona densamente conectada, con una función de activación ***Sigmoid*** la cual proporciona una probabilidad de salida.

```
#Modelo secuencial
modelo1 = Sequential()
#Primera convolución
modelo1.add(Conv2D(32, (3, 3), activation="relu", input_shape=(64, 64, 3)))
#Capa de pooling
modelo1.add(MaxPooling2D(pool_size = (2, 2)))
#Segunda convolucion
modelo1.add(Conv2D(32, (3, 3), activation="relu"))
#Capa de pooling
modelo1.add(MaxPooling2D(pool_size = (2, 2)))
#Aplanamos
modelo1.add(Flatten())
# Conectamos densamente
modelo1.add(Dense(activation = 'relu', units = 128))
#Salida
modelo1.add(Dense(activation = 'sigmoid', units = 1))
```

Código 38. Definición del modelo1 de red neuronal convolucional

Podemos ver visualmente las entradas y salidas a cada capa de nuestra red

```
plot_model(cnn7, to_file='modelo.png', show_shapes=True, show_layer_names=True)
```

Código 39. Fragmento de código. Visualización de la arquitectura de red definida

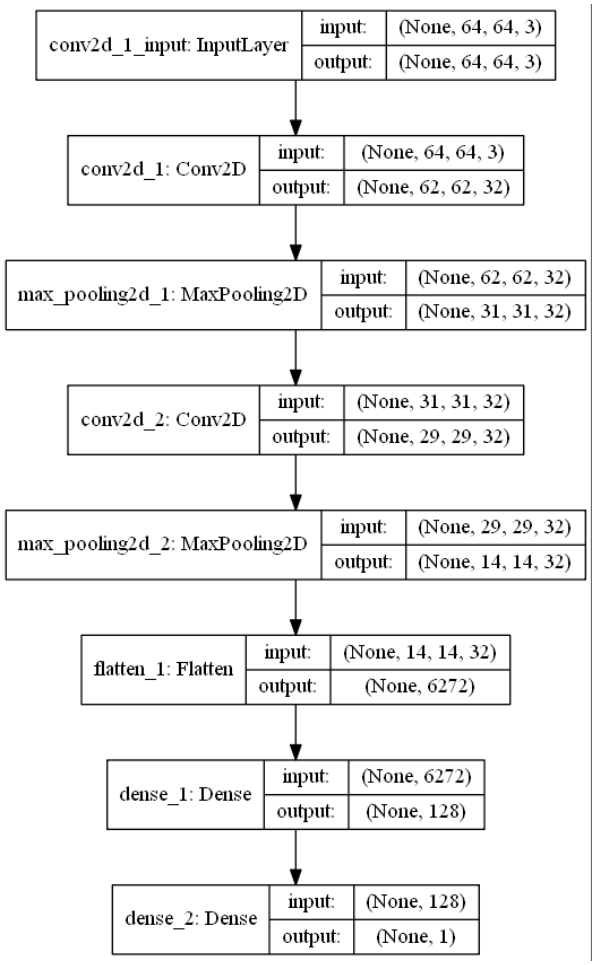


Figura 24. Modelo de red neuronal convolucional definido

5.6.2 Monitorización del proceso de aprendizaje

Tras haber configurado nuestro modelo de red neuronal convolucional, tenemos que definir el conjunto de parámetros que llevarán a cabo el proceso de aprendizaje de la red haciendo uso del método ***compile().***

Para la elección de estos 3 parámetros, es necesario tener en cuenta con qué tipo de problema estamos trabajando. En nuestro caso nos encontramos en un problema de clasificación binaria, por lo que, para la función de pérdida, la cual nos indica lo cerca que está la red de su óptimo global, escogemos la más apropiada para este tipo de problemas ***binary_crossentropy.*** Como optimizador escogemos el algoritmo ***Adam,*** mediante este la red se actualizará así misma y a la función de pérdida. Finalmente, la métrica establecida para conocer el éxito durante el entrenamiento es ***accuracy***, esta nos dará una información de evaluación preventiva tras cada época, indicándonos el número de aciertos que ha tenido la red.

```
modelo1.compile(
    optimizer = 'adam',
    loss = 'binary_crossentropy',
    metrics = ['accuracy'])
```

Código 40. Fragmento de código. Monitorización del proceso de aprendizaje del modelo 1

5.6.3 Entrenamiento del modelo definido

Finalmente, tras la configuración del modelo de red neuronal y la configuración de los parámetros para monitorizar el aprendizaje, vamos a entrenar la red mediante ***fit_generator()*** en vez de con el método ***fit()*** ya que recordemos que estamos trabajando con generadores de datos.

Establecemos un total de 50 épocas mediante ***epochs*** con un total de 109 lotes de imágenes por época.

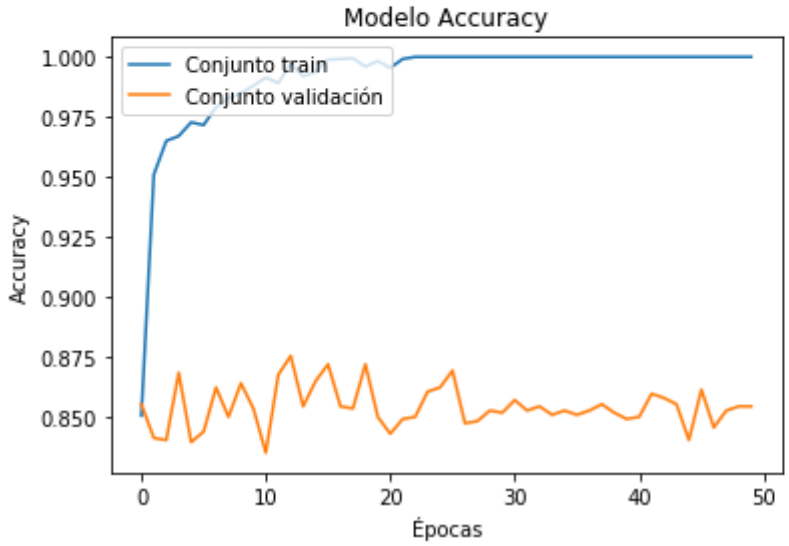
Para medir el comportamiento del modelo durante el entrenamiento hacemos uso de la validación cruzada mediante el método hold-out. Esto como sabemos nos permitirá conocer la evolución del entrenamiento. Mediante ***validation_data*** le pasamos el conjunto de datos destinados a validación con un total de 50 lotes de validación al finalizar una época, indicado por medio de ***validation_steps***.

```
ejemplos_train = 3512
ejemplos_validacion = 1170
batch = 32

modeloF1 = modelo1.fit_generator(
    train_generator,
    steps_per_epoch = ejemplos_train // batch,
    epochs =50,
    validation_data = validation_generator,
    validation_steps = ejemplos_validacion // batch )
```

Código 41. Fragmento de código. Entrenamiento del modelo definido

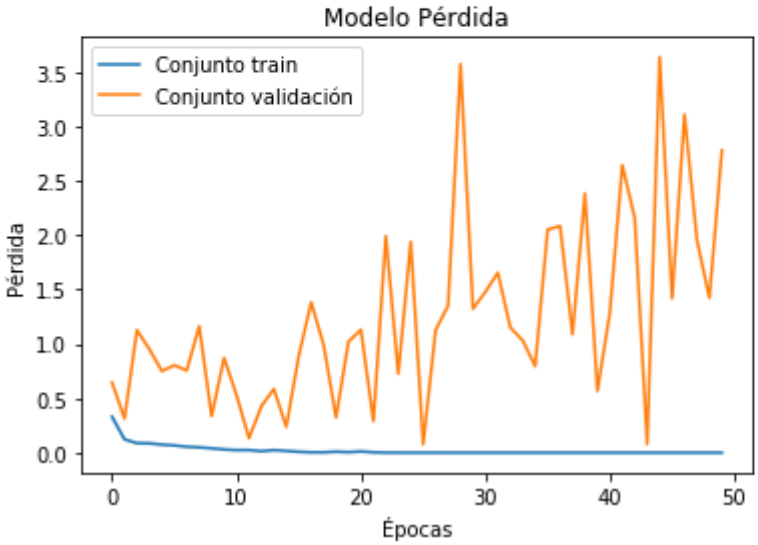
Para observar el comportamiento del modelo durante el entrenamiento vamos a representar la pérdida y la precisión experimentada durante el entrenamiento. Para ello hacemos uso de la librería **matplotlib**.

```
plt.plot(modeloF1.history['accuracy'])
plt.plot(modeloF1.history['val_accuracy'])
plt.title('Modelo Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Épocas')
plt.legend(['Conjunto train', 'Conjunto validación'], loc='upper left')
plt.show()
```



Cuadro 4. Gráfica Métrica accuracy durante el entrenamiento

```
plt.plot(modeloF1.history['loss'])
plt.plot(modeloF1.history['val_loss'])
plt.title('Modelo Pérdida')
plt.ylabel('Pérdida')
plt.xlabel('Épocas')
plt.legend(['Conjunto train', 'Conjunto validación'], loc='upper left')
plt.show()
```



Cuadro 5. Gráfica función de pérdida durante el entrenamiento

Las gráficas anteriores nos muestran la evolución del entrenamiento en función de los valores de pérdida y precisión sobre el conjunto de datos de entrenamiento y validación.

Estas gráficas como podemos observar presentan característica de un modelo sobreajustado. Esto se puede apreciar con gran claridad en la gráfica que representa la precisión de la red sobre los datos de entrenamiento y validación. En dicha gráfica se puede observar como la precisión sobre el conjunto de entrenamiento se va afinando cada vez más aproximándose al 100%.

La precisión del conjunto de datos de validación comienza creciendo, presentando unas oscilaciones más estables a partir de la época 10 hasta la 20, donde su precisión se mantiene entre un 84-86%, sin embargo, a partir de la época 20 esta precisión comienza a degradarse.

Esta apreciación se puede observar también en la gráfica de pérdida. En ella la evolución de los valores de pérdida con el conjunto de datos de entrenamiento va decreciendo de forma contínua aproximándose a valores cercanos al 0, sin embargo, el valor de pérdida del conjunto de validación comienza decreciendo hasta que a partir de la época 20 empieza a dispararse.

5.6.4 Evaluación del modelo 1

Finalmente, vamos a evaluar el modelo dado un conjunto de datos que nunca ha visto, el conjunto de datos de test. Recordemos que, llegados a este punto, el modelo definido de red neuronal convolucional ya se ha entrenado. Esto se lleva a cabo con el método evaluate_generator(), el cual actúa igual que el método evaluate(), la única diferencia es que estamos trabajando con generadores de datos. Este método nos devolverá dos valores de salida, la pérdida y la métrica accuracy.

```
test_accu1 = modelo1.evaluate_generator(test_generator)
```

```
test_accu1
```

```
[6.519802354887361e-06, 0.9752559661865234]
```

Código 42. Fragmento de código. Evaluación del modelo 1

El modelo desarrollado presenta una evaluación del conjunto de datos de test del 97%, sin embargo, no debemos dejar de tener en mente que estamos tratando con un conjunto de datos desbalanceados y que como hemos visto en la evolución del entrenamiento la red se encuentra sobreajustada, por ello vamos a hacer uso de diferentes técnicas para solventar el problema del desbalanceo de datos y el sobreajuste.

5.7 MODELO 2

En el siguiente modelo vamos a hacer uso de la misma configuración de parámetros e hiperparámetros establecidos en el modelo 1 pero vamos a utilizar en este caso la técnica para solventar el problema del desbalanceo de datos. Con el uso de esta técnica pretendemos mejorar el entrenamiento de nuestra red, disminuyendo el desbalanceo y el sobreajuste como causa de esto.

5.7.1 Aumento de datos

Como habíamos comentado con anterioridad, nuestros datos se encuentran desbalanceados, esto es un gran problema a la hora de entrenar una red, perjudicando la clasificación de la clase minoritaria. Para solventar este problema hacemos uso del aumento de datos facilitado por keras a través de la clase ImagenDataGenerator. Su objetivo es aumentar el tamaño del conjunto de datos de entrenamiento, mediante la creación de versiones modificadas de imágenes en el conjunto de entrenamiento.

Las transformaciones realizadas son:

- Ángulo de corte en sentido antihorario.
- Zoom aleatorio
- Volteo aleatorio de las entradas horizontalmente

Se podrían haber aplicado muchas otras más transformaciones a una imagen, pero no eran útiles en nuestro caso.

```
#Aumento de datos, reescalar en punto flotante y normalización
train_datagen = ImageDataGenerator(rescale = 1./255,
                                   shear_range = 0.2,
                                   zoom_range = 0.2,
                                   horizontal_flip = True)
```

Código 43. Fragmento de código. Aumento de datos modelo 2

Este aumento de datos es aplicado únicamente al conjunto de entrenamiento.

```
ejemplos_train = 3512
ejemplos_validacion = 1170
batch = 32
train_generator = train_datagen.flow_from_directory(
        #Directorio
        new_train,
        #Redimension de las imágenes a 64x64
        target_size=(64, 64),
        #Número de imágenes por lote de datos
        batch_size=batch,
        # Etiquetas binarias
        class_mode='binary')
```

Código 44. Fragmento de código. Aumento de datos aplicado a conjunto train.

5.7.2 Configuración del modelo de red neuronal convolucional 2

Vamos a hacer uso de la misma configuración del modelo 1 desarrollado para observar la influencia de la técnica del aumento de datos a este y observar si mejora.

Este estaba formado por dos capas convolucionales de 32 filtros cada una, con una capa de pooling a continuación de cada convolución. Finalmente aplanamiento, una capa oculta tradicional de 100 neuronas y una capa de salida de una única neurona con una función de activación sigmoid.

```
#Modelo secuencial
modelo2 = Sequential()
#Primera convolución
modelo2.add(Conv2D(32, (3, 3), activation="relu", input_shape=(64, 64, 3)))
#Capa de pooling
modelo2.add(MaxPooling2D(pool_size = (2, 2)))
#Segunda convolucion
modelo2.add(Conv2D(32, (3, 3), activation="relu"))
#Capa de pooling
modelo2.add(MaxPooling2D(pool_size = (2, 2)))
#Aplanamos
modelo2.add(Flatten())
# Conectamos densamente
modelo2.add(Dense(activation = 'relu', units = 128))
#Salida
modelo2.add(Dense(activation = 'sigmoid', units = 1))
```

Código 45. Fragmento de código. Configuración modelo red neuronal convolucional 2

5.7.3 Monitorización del proceso de aprendizaje 2.

Realizamos la misma configuración del proceso de aprendizaje que en el modelo 1.

```
modelo2.compile(
    optimizer = 'adam',
    loss = 'binary_crossentropy',
    metrics = ['accuracy'])
```

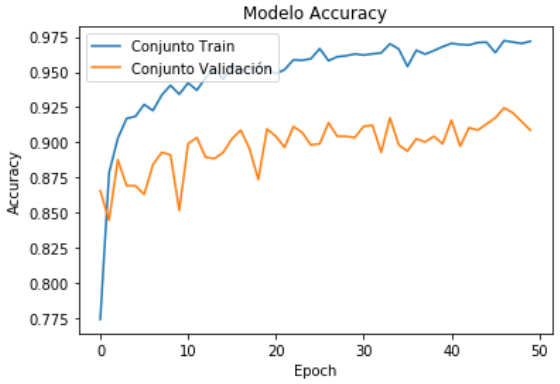
Código 46. Fragmento de código. Monitorización del proceso de aprendizaje del modelo 2

5.7.4 Entrenamiento del modelo definido.

```
ejemplos_train = 3512
ejemplos_validacion = 1170
batch = 32

modeloF2 = modelo2.fit_generator(
    train_generator,
    steps_per_epoch = ejemplos_train // batch,
    epochs =50,
    validation_data = validation_generator,
    validation_steps = ejemplos_validacion // batch )
```

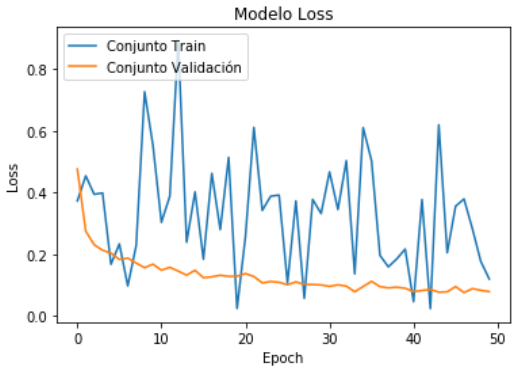
Código 47. Fragmento de código. Entrenamiento modelo 2

```
plt.plot(modeloF2.history['accuracy'])
plt.plot(modeloF2.history['val_accuracy'])
plt.title('Modelo Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Épocas')
plt.legend(['Conjunto Train', 'Conjunto Validación'], loc='upper left')
plt.show()
```



Cuadro 6. Gráfica Evolución accuracy modelo 2

```
plt.plot(modeloF2.history['loss'])
plt.plot(modeloF2.history['val_loss'])
plt.title('Modelo Pérdida')
plt.ylabel('Pérdida')
plt.xlabel('Épocas')
plt.legend(['Conjunto Train', 'Conjunto Validación'], loc='upper left')
plt.show()
```



Cuadro 7. Gráfico evolución función de perdida modelo 2

Como podemos observar nuestro modelo de entrenamiento ha mejorado considerablemente al aplicarle la técnica del Aumento de dato. Esta técnica solventa el problema del desbalanceo de datos que a su vez ayuda a eliminar el sobreajuste de la red.

La gráfica de precisión presenta una evolución de la precisión sobre el conjunto de entrenamiento crece de forma continua aproximándose a valores cercanos al 100%, al igual que la evolución de la precisión sobre el conjunto de datos de validación, sin embargo, esta presenta una evolución rápida y creciente hasta la época 20 donde a partir de entonces el crecimiento disminuye sin llegar a degradarse. Mantiene una precisión sobre los datos de validación de un 89-91%.

Por otra parte, en la gráfica de la evolución del valor de pérdida durante el entrenamiento. La evolución del valor de pérdida sobre el conjunto de datos de entrenamiento disminuye de forma continua aproximándose al 0, sin embargo, la evolución de la pérdida en el conjunto de datos de validación presenta unos rangos muy dispares al principio los cuales poco a poco conforme evoluciona van disminuyendo hasta la época 20, a partir de entonces se mantiene en un rango estable entre el 0.2-0.5.

Como hemos dicho, nuestro modelo ha mejorado considerablemente, pero sigue presentando características de modelo sobreajustado.

5.7.5 Evaluación del modelo 2

```
test_accu2 = modelo2.evaluate_generator(test_generator)
```

```
test_accu2
```

```
[0.2273104339838028, 0.9210069179534912]
```

Código 48. Fragmento de código. Evaluación modelo 2

El modelo desarrollado presenta una evaluación del conjunto de datos de test del 92%, un porcentaje menor que el modelo anterior, aun presentando mejor evolución a lo largo del entrenamietno. Aunque el modelo haya mejorado considerablemente con respecto al modelo 1, sigue presentando sobreajuste. Vamos a aplicar a continuación una técnica de regularización para mejorar el modelo eliminando el sobreajuste.

5.8 MODELO 3

Debido a los buenos resultados obtenidos en el modelo anterior debido a la aplicación de la técnica del Aumento de datos, vamos a reproducir en este modelo el modelo anterior con los mismos parámetros e hiperparámetros y la técnica del Aumento de datos y vamos a añadir una técnica para disminuir el sobreajuste, esta técnica es llamada **Dropout**.

5.8.1 Aumento de datos

```
#Aumento de datos, reescalar en punto flotante y normalización
train_datagen = ImageDataGenerator(rescale = 1./255,
                                   shear_range = 0.2,
                                   zoom_range = 0.2,
                                   horizontal_flip = True)
```

```
ejemplos_train = 3512
ejemplos_validacion = 1170
batch = 32
train_generator = train_datagen.flow_from_directory(
        #Directorio
        new_train,
        #Redimension de las imágenes a 64x64
        target_size=(64, 64),
        #Número de imágenes por lote de datos
        batch_size=batch,
        # Etiquetas binarias
        class_mode='binary')
```

Código 49. Fragmento de código. Aumento de datos modelo 2

5.8.2 Configuración del modelo de red neuronal convolucional 3 – Técnica Dropout

Vamos a hacer uso de la misma configuración del modelo 1, pero en este caso le aplicaremos la técnica de Dropout. Esta es una de las técnicas de regularización de las redes neuronales. Esta consiste en poner de forma aleatoria a 0 un conjunto de valores de la salida de la capa de la red neuronal durante el entrenamiento. Al poner a 0 un conjunto de valores introducimos ruido, esto permite que los patrones no significativos, aquellos que la red aprende ajustando en exceso, se rompan, solventando así el problema de sobreajuste.

Como nos encontramos en un problema con desbalanceo de datos y dataset pequeño, para evitar el sobreajuste de la red hemos añadido una capa ***Dropout()*** al 0.5, que como hemos visto pone a 0 el 50% de la salida.

```
#Modelo secuencial
modelo3 = Sequential()
#Primera convolución
modelo3.add(Conv2D(32, (3, 3), activation="relu", input_shape=(64, 64, 3)))
#Capa de pooling
modelo3.add(MaxPooling2D(pool_size = (2, 2)))
#Segunda convolucion
modelo3.add(Conv2D(32, (3, 3), activation="relu"))
#Capa de pooling
modelo3.add(MaxPooling2D(pool_size = (2, 2)))
#Aplanamos
modelo3.add(Flatten())
# Conectamos densamente
modelo3.add(Dense(activation = 'relu', units = 128))
#Técnica dropout
modelo3.add(Dropout(0.5))
#Salida
modelo3.add(Dense(activation = 'sigmoid', units = 1))
```

Código 50. Fragmento de código. Configuración modelo de red neuronal convolucional 3

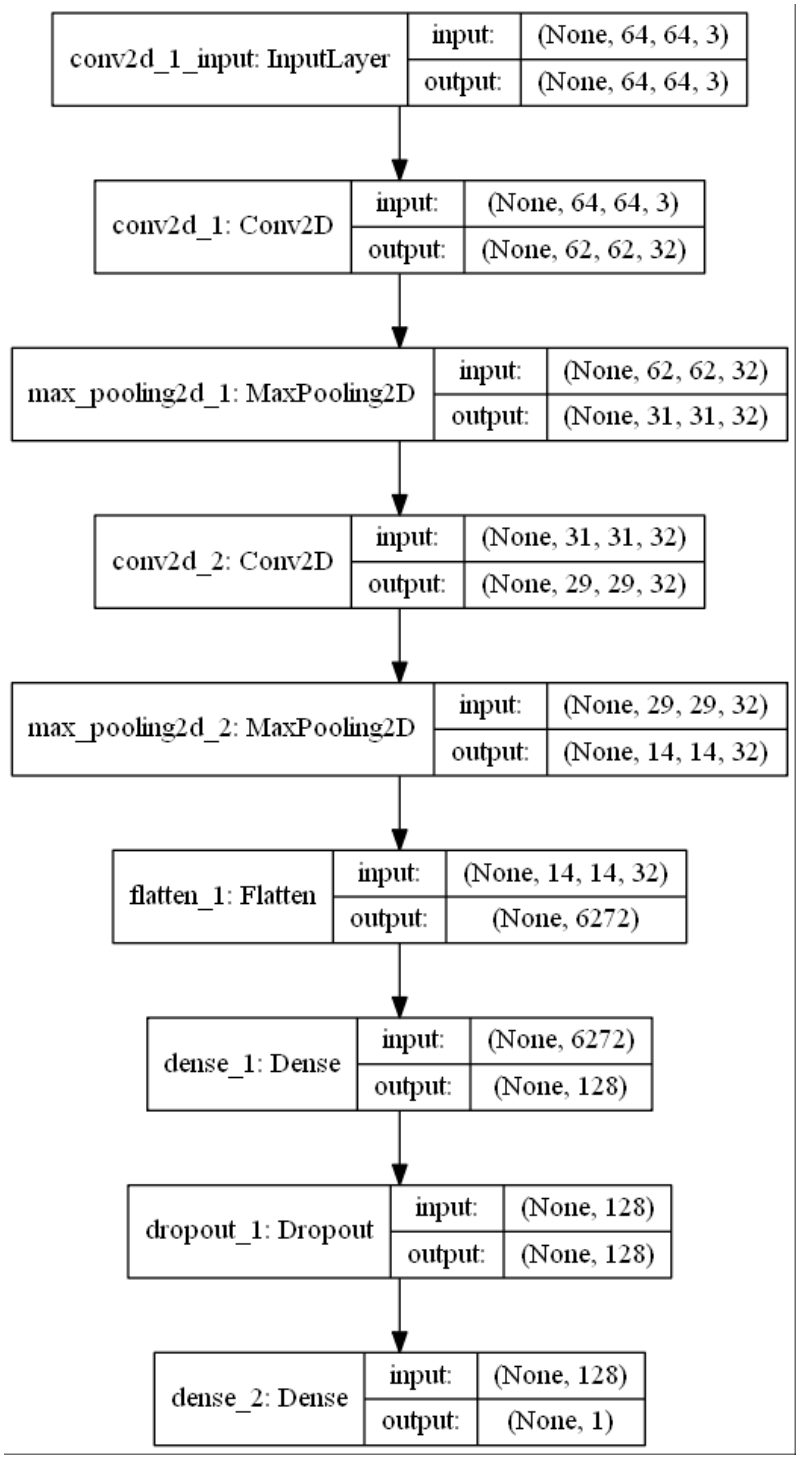


Figura 25. Modelo red neuronal convolucional 3

5.8.3 Monitorización del proceso de aprendizaje.

```
modelo3.compile(
    optimizer = 'adam',
    loss = 'binary_crossentropy',
    metrics = ['accuracy'])
```

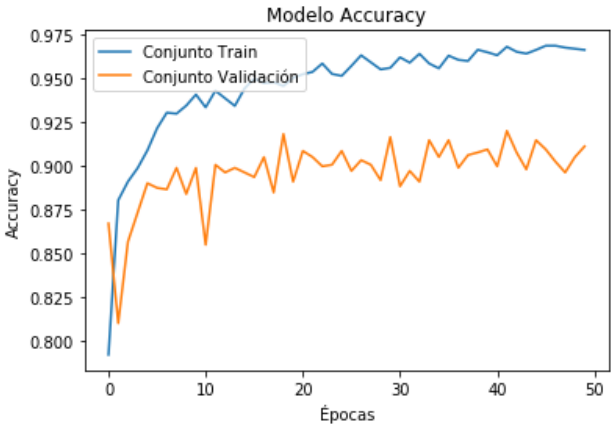
Código 51. Fragmento de código. Monitorización del proceso de aprendizaje modelo 3

5.8.4 Entrenamiento del modelo definido.

```
ejemplos_train = 3512
ejemplos_validacion = 1170
batch = 32

modeloF3 = modelo3.fit_generator(
    train_generator,
    steps_per_epoch = ejemplos_train // batch,
    epochs =50,
    validation_data = validation_generator,
    validation_steps = ejemplos_validacion // batch )
```

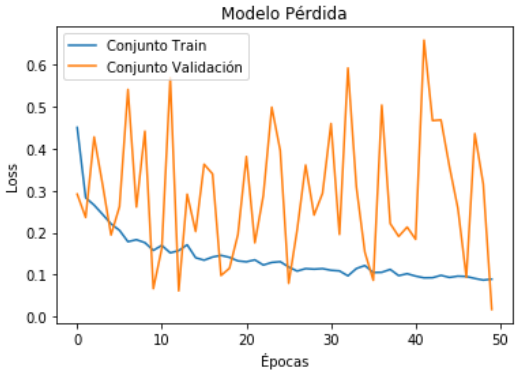
Código 52. Fragmento código. Entrenamiento modelo 3

```
plt.plot(modeloF3.history['accuracy'])
plt.plot(modeloF3.history['val_accuracy'])
plt.title('Modelo Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Épocas')
plt.legend(['Conjunto Train', 'Conjunto Validación'], loc='upper left')
plt.show()
```



Cuadro 8. Grafica evolución precisión durante el entrenamiento del modelo 3

```
plt.plot(modeloF3.history['loss'])
plt.plot(modeloF3.history['val_loss'])
plt.title('Modelo Pérdida')
plt.ylabel('Loss')
plt.xlabel('Épocas')
plt.legend(['Conjunto Train', 'Conjunto Validación'], loc='upper left')
plt.show()
```



Cuadro 9. Gráfica evolución de perdida durante el entrenamiento del modelo 3

Finalmente, el modelo 3 en el cual se han usado técnicas de aumento de datos y regularización como a técnica dropout, presenta una evolución de precisión y perdida a lo largo del entrenamiento poco homogéneo. Aunque estas técnicas hayan mejorado considerablemente el modelo, las gráficas nos muestran características de un modelo sobreajustado, debido a la dispersión de valores que presenta la precisión y la perdida sobre los datos de validación con respecto a los datos de entrenamiento.

Como se pueden observar, en la gráfica que nos muestra la evolución de la precisión del modelo durante el entrenamiento, se puede observar como la precisión de entrenamiento crece de forma continua, linealmente con el tiempo hasta alcanzar valores cercanos al 100%, sin embargo, el crecimiento progresivo de la precisión de validación se detiene entre el 89-91%.

Por otra parte, la pérdida de validación alcanza el mínimo al igual que antes aproximadamente posterior a décima época, mientras que la de entrenamiento continúa cayendo, la pérdida de validación comienza a oscilar entre un margen de error de 0.05-0.5.

5.8.5 Evaluación del modelo 3.

```
test_accu3 = modelo3.evaluate_generator(test_generator)
```

```
test_accu3
```

```
[0.12408211827278137, 0.9626736044883728]
```

Como podemos observar la métrica de la evaluación del modelo es de un 96%, esto es una métrica muy buena, pero no debemos olvidar que estamos trabajando con un dataset que se encuentra desbalanceado y que las gráficas representadas con anterioridad presentaban características de modelos sobreajustados, por lo que es conveniente aplicar otras métricas con las que podamos observar la ejecución del modelo

5.8.6 Medidas de rendimiento.

Como medidas del éxito del rendimiento de nuestra red neuronal convolucional, habíamos determinado la métrica accuracy, esta como sabemos te devuelve el porcentaje de aciertos que ha tenido la red.

Para establecer una medida de rendimiento hay que tener en cuenta el problema con el que estamos trabajando y las condiciones en las que se encuentra. En nuestro caso como ya hemos nombrado con anterioridad, trabajamos con un dataset pequeño cuyos datos se encuentran desbalanceados. Aunque hayamos aplicado diferentes técnicas para solventar estos problemas es mejor ser precavido y utilizar otra medida de rendimiento. En nuestro caso vamos a hacer uso de la matriz de confusión como ya habíamos mencionado.

Una matriz de confusión es una herramienta que permite la visualización de la ejecución de un algoritmo de clasificación empleado en aprendizaje supervisado. Por medio de ella podemos observar:

- Verdaderos positivos: porcentaje de valores positivos clasificados correctamente
- Falsos positivos: porcentaje de valores positivos clasificados incorrectamente.
- Falsos negativos: porcentaje de valores negativos clasificados incorrectamente.

- Verdaderos negativos: porcentaje de valores negativos clasificados correctamente.

La representación de una red neuronal correctamente entrenada tendría valores solo en la diagonal, es decir en verdaderos positivos y verdaderos negativos. (Barrios, 2019)

		Predicción	
		Positivos	**Negativos**
Observación	**Positivos**	Verdaderos Positivos (VP)	Falsos Negativos (FN)
	Negativos	Falsos Positivos (FP)	Verdaderos Negativos (VN)

Figura 26. Matriz de confusión

La implementación de la matriz de confusión es necesario dos conjuntos de datos, los valores predichos por la red y las etiquetas asociadas a dichos valores.

Para obtener los valores predichos de la red neuronal ya entrenada a través de nuestro conjunto de datos test, hacemos uso de la función predict(). Estas predicciones las almacenaremos en la variable preds.

```
preds = modelo3.predict(test_data)
```

Código 53. Fragmento de código. Predicciones del modelo creado

Por otra parte, almacenamos las etiquetas correspondientes al conjunto de test ***test_labels***. Posteriormente obtenemos los índices de los valores máximos a lo largo del eje indicado mediante el método ***argmax()*** quedando estos almacenados en ***preds_val*** y ***orig_labels.***

```
preds_val = np.argmax(preds, axis = 1)
orig_labels = np.argmax(test_labels, axis=1)
```

Código 54. Fragmento de código. Obtención de índices de valores máximos

Finalmente, mediante el uso de la función ***confusion_matriz*** a la cual le damos como parámetros de entrada ***preds_val*** y ***orig_labels*** generamos una matriz de confusión, en ella se comparan los resultados predichos por la red y los reales, posteriormente lo representamos.

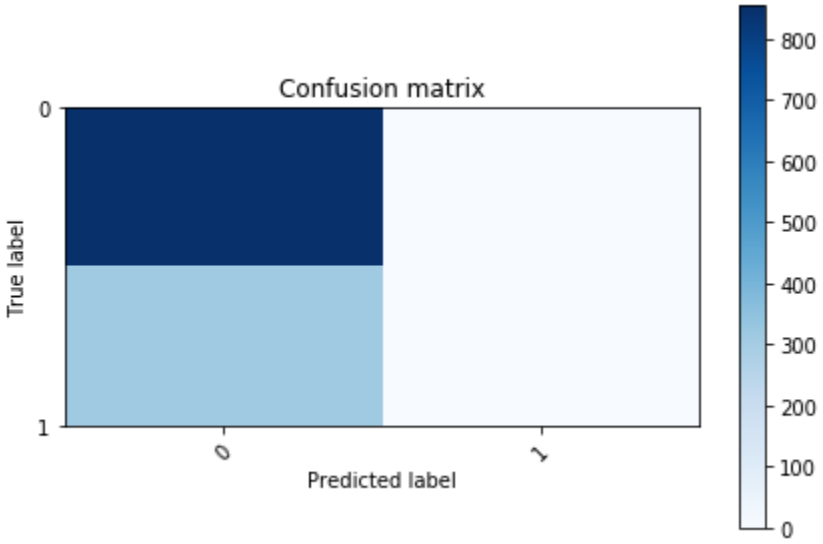
```
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

```
matriz_de_confusion = confusion_matrix(orig_labels, preds_val)
```

```
plot_confusion_matrix(matriz_de_confusion, classes = range(2))
```

Código 55. Fragmento de código. Representación e implementación de la matriz de confusión

El resultado de nuestra matriz de confusión es el siguiente.



Cuadro 10. Matriz de confusión de nuestro modelo

En la matriz de confusión podemos corroborar lo mencionado anteriormente que no sabíamos con total certeza. Nuestra modelo de red neuronal convolucional ha sido sobreajustado en el entrenamiento, debido al desbalanceo de datos.

El conjunto de datos de test, estaba formado por:

- 1170 imágenes
  - 855 clase 0
    - Bien clasificadas 800
  - 315 a la clase 1.
    - Bien clasificadas 50

Como podemos observar nuestro modelo de red neuronal solo ha podido clasificar con éxito aproximadamente 850 imágenes, siendo 800 de la clase 0, la clase mayoritaria y 50 imágenes correctamente clasificadas de la clase 1, clase minoritaria.

Efectivamente nuestro modelo de red neuronal ha sido sobreajustado en el entrenamiento y debido a la presencia de los datos desbalanceados, ha realizado un aprendizaje casi al 100% de la clase mayoritaria, pero no de la minoritaria.

En este caso hemos podido observar el motivo por el que no nos debemos fiar de la métrica acurracy con más índole cuando se tratan de problemas desbalanceados, ya que el porcentaje de aciertos predicho la red (72.6%) no es tan mal resultado, sin embargo, al visualizar la ejecución del algoritmo, podemos observar que la red no ha sido correctamente entrenada ya que es capaz de clasificar al 100% la clase mayoritaria pero no la minoritaria. Teniendo un mal funcionamiento.

6. Discusión

A lo largo de la ejecución de nuestro trabajo basado en el desarrollo de un modelo de red neuronal convolucional que cumpla nuestro objetivo previsto, realizar una correcta clasificación en las imágenes de radiografía de pulmón en las clases con patología de neumonía o sin patología de neumonía, nos hemos ido encontrando con una serie de dificultades, debido a las cuales hemos realizado diversas pruebas hasta alcanzar el resultado que considerábamos óptimo y que acabamos de explicar en el capítulo anterior.

Estas dificultades no solo nos las hemos encontrado durante el entrenamiento del modelo de red configurado, proceso en el que ya conocemos que es un proceso de ajuste es un continuo de hiperparámetros, en el que se alcanza la mejor configuración mediante un proceso continuo de prueba error, sino también, hemos encontrado dificultades en nuestro dataset, como ya hemos comentado.

A continuación, vamos a explicar de forma progresiva el conjunto de dificultades encontradas, opciones a optar y camino escogido.

6.1 Validación cruzada.

En primer lugar, al conocer las características que presentaba nuestro dataset, desbalanceo de datos y dataset pequeño, era conveniente conocer la evolución del proceso de entrenamiento, para poder observar el proceso de ajuste que se está llevando a cabo.

Para ello, podemos hacer uso de dos tipos de técnicas, ya explicadas con anterioridad a lo largo del documento:

- Validación cruzada de hold-out
- Validación cruzada k-fold

La técnica utilizada fue la validación cruzada de hold-out, ya que es la más utiliza si el problema no presenta grandes dificultades, sin embargo, la validación cruzada k-fold, presenta mejores resultados para evitar el sobreajuste.

Aplicamos la validación cruzada hold-out desde el primer momento. Esta nos permite conocer cómo evoluciona el entrenamiento tras cada época, ofreciéndonos una información relativa en la cual se observa donde alcanza su óptimo la red o a partir de donde comienza a degradarse.

6.2 Ajuste de hiperparámetros.

El ajuste de los hiperparámetros ha sido la dificultad encontrada de mayor peso en el trabajo, debido a su alto coste temporal, como hemos dicho el correcto entrenamiento de una red neuronal se alcanza mediante un proceso de prueba y error. Configuramos un modelo, lo entrenamos y evaluamos, si no es correcto el entrenamiento, ajustamos los hiperparámetros y de nuevo lo entrenamos y evaluamos, sucesivamente hasta encontrar la configuración de hiperparámetros que optimicen el entrenamiento.

El conjunto de modificaciones de los hiperparámetros realizados para la configuración del entrenamiento han sido las siguientes:

Número de capas convolucionales	2, 4, 6
Número de filtros en capa convolucional	20, 32
Batch-size	10, 20, 32
Épocas	20, 50, 100
Optimizador	Adam, RMSPROP

Cuadro 11. Ajuste hiperparámetros

La métrica y función de pérdida fueron escogidas desde el primer momento. Como métrica utilizamos **accuracy** (porcentaje de aciertos) y como función de perdida utilizamos el tipo de función de pérdida que se ajusta al problema con el que trabajamos, en este caso trabajamos con un problema de clasificación binaria por lo que como función de perdida haremos uso de **binary_crossentropy.**

Finalmente, la configuración de hiperparámetros con la que se obtuvieron mejores resultados, fueron las siguientes, presentadas en el capítulo anterior.

Número de capas convolucionales	2
Número de filtros en capa convolucional	32
Batch-size	32
Épocas	50
Optimizador	Adam

Cuadro 12. Configuración del modelo final

6.3 Desbalanceo de datos

Posteriormente, nos encontramos con la dificultad que presentaba nuestro dataset. Como ya había comentado, nos encontrábamos con un dataset pequeño cuyos datos estaban desbalanceados. El **desbalanceo de datos** nos lleva a un mal entrenamiento de la red neuronal configurada perjudicando a la clase minoritaria.

El conjunto de técnicas que se podían emplear para solventar este problema fue:

- Ajuste de parámetros del modelo: Esta consiste en ajustar la métrica de la función de pérdida mediante la penalización de la clase mayoritaria durante el entrenamiento, con el objetivo de equilibrar ambas clases.

- Modificar el dataset: Consiste en eliminar muestras de la clase mayoritaria hasta alcanzar el equilibrio con la clase minoritaria.

- Aumento de datos: Consiste en realizar muestras sintéticas por medio de un conjunto de transformaciones aplicadas en nuestros datos originales

- Balanced ensembled methods: Ensambla métodos. Es decir, entrena diversos modelos y entre todos obtienen un resultado final. (Bagnato, 2019)

De estas cuatro posibilidades para solventar el problema del desbalanceo de datos, el camino escogido fue hacer uso del **Aumento de datos.**

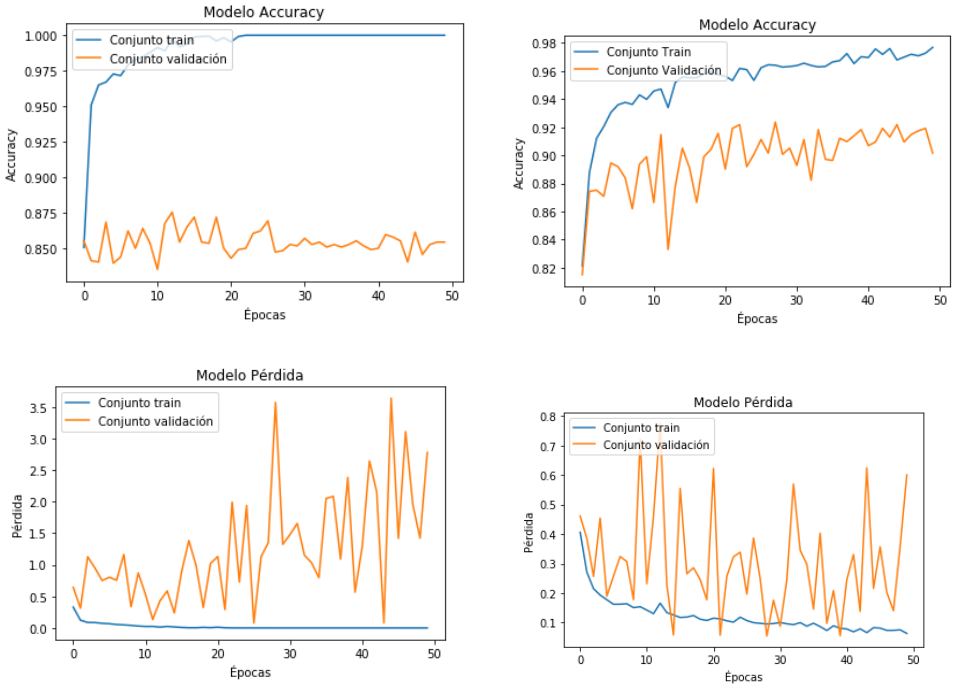
La técnica de modificación del dataset no era conveniente aplicarla en nuestro caso ya que trabajábamos con un dataset pequeño y aunque esa técnica solventaba el problema del desbalanceo nos podía suponer otro problema de generalización de patrones por falta de datos.

La técnica Balance ensembled methods conlleva un rendimiento computacional muy alto y no podíamos hacer frente a ello.

Posteriormente, la técnica de ajuste de parámetros podía ser aplicada a nuestro problema, sin embargo, la técnica de Aumento de datos es la más utilizada y también perfectamente aplicable a nuestro problema. Por ello, finalmente, nos decantamos por implementar la técnica del Aumento de datos.

Tras aplicar la técnica de aumento de datos se pudo observar una mejora considerable en el proceso de entrenamiento del modelo de red neuronal convolucional definido.

Modelo 1 **Modelo 2**



Cuadro 13. Gráficas comparativas modelo sin aumento de datos y con aumento de datos.

Como podemos observar en esta imagen, las gráficas que se encuentran a la izquierda representan la evolución de la precisión y el valor de perdida durante el entrenamiento en un modelo sin el uso del aumento de datos y las graficas que se encuentran a la derecha representan la evolución de la precisión y el valor de perdida durante el entrenamiento en un modelo con el uso del aumento de datos.

La mejora del entrenamiento es notablemente satisfactoria, el modelo 1 el cual no presenta el aumento de datos tiene una evolución de la precisión sobre los datos de validación en un rango de 84-86% y presenta unos valores de función de pérdida muy dispares encontrándose en un rango de 1-3. Sin embargo, el modelo 2 el cual hace uso del aumento de datos presenta una evolución de la precisión sobre los datos de validación entorno a unos valores dentro del rango 88-91% y unos valores de la función de pérdida en torno a 0.1-0.6.

6.4 Regularización

Un entrenamiento óptimo se alcanza obteniendo un equilibrio entre la generalización y la optimización. Alcanzar este equilibrio es condenablemente difícil ya que lo que separa un ajuste perfecto del sobreajuste o subajuste es mínimo.

Las técnicas más usadas para evitar este sobreajuste son:

- Reducción del tamaño de la red
- Regularización de los pesos
- Dropout.

Como mencionamos con anterioridad, la técnica de reducción del tamaño no es aplicable en nuestro modelo ya que nos encontramos trabajando con un dataset pequeño. Regularización de los pesos y dropout son las técnicas más usadas, destacando por encima la técnica Dropout debido a sus satisfactorios resultados.

La técnica de **Dropout** es una de las técnicas de regularización de las redes neuronales. Esta consiste en poner de forma aleatoria a 0 un conjunto de valores de la salida de la capa de la red neuronal durante el entrenamiento. Al poner a 0 un conjunto de valores introducimos ruido, esto permite que los patrones no significativos, aquellos que la red aprende ajustando en exceso, se rompan, solventando así el problema de sobreajuste.

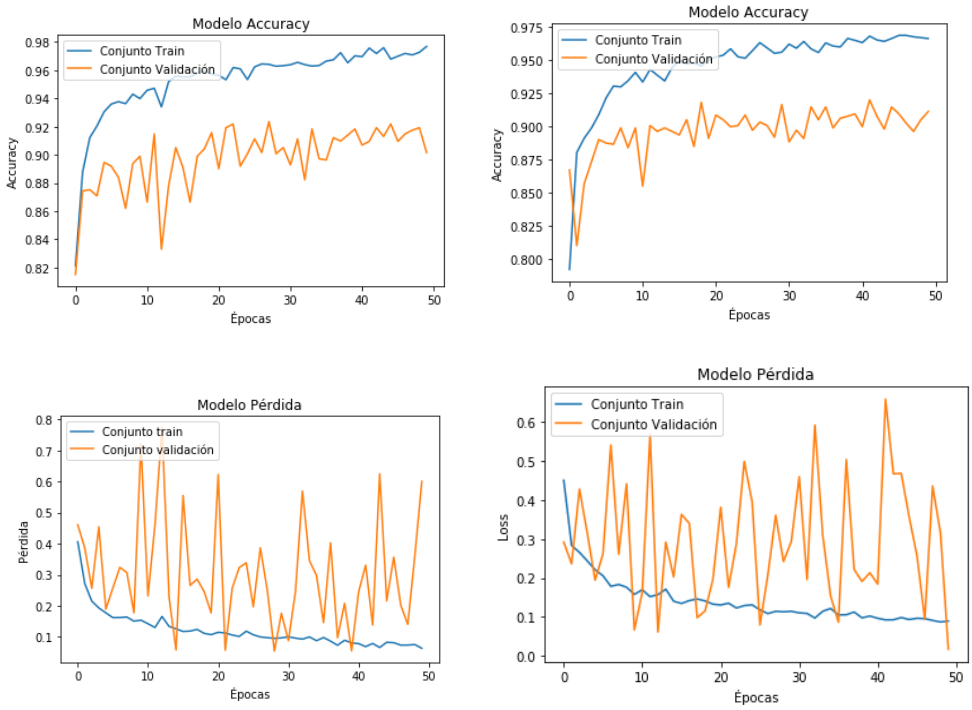
Aplicamos la técnica de Dropout al 20% de las salidas y al 50% de las salidas. Finalmente obtuvimos mejores resultados aplicando dropout al 50%.

Dropout	0.2, 0.5

Cuadro 14. Valores dropout

Tras aplicar esta técnica el sobreajuste que presentaba nuestro modelo mejoro, pero no se eliminó.

Modelo 2 **Modelo 3**



Cuadro 15. Gráficas comparativas modelo sin técnica dropout y con técnica dropout.

Las gráficas que se encuentran a la derecha muestran la evolución de la precisión y pérdida sobre el conjunto de datos de validación durante el entrenamiento del modelo 2 (modelo con aumento de datos sin dropout), a la derecha podemos observar la evolución de la precisión y pérdida sobre el conjunto de datos de validación durante el entrenamiento del modelo 2 (modelo con aumento de datos y con dropout).

Como podemos ver el sobreajuste no es eliminado como se pueden observar en las dos primeras gráficas, donde la precisión del entrenamiento para ambos modelos se mantiene dentro del rango de los 89-91%, pero el margen de error disminuye en el modelo 3. Podemos decir que la técnica dropout no ha erradicado el sobreajuste, pero si disminuido obteniendo resultados levemente mejores.

6.5 Temporalización

De forma paralela al ajuste de hiperparámetros, nos encontramos con la dificultad temporal.

Cabe destacar con gran índole el coste temporal que conlleva el ajuste de hiperparámetros. Esta requiere una alta capacidad de cómputo, por ello es necesario hacer uso de una GPU. Para solventar el problema del coste computacional diluyendo así el coste temporal, teníamos dos alternativas:

- Comprar una GPU
- Google colab

Finalmente, comenzamos a hacer uso de la herramienta facilitada por Google llamada Google colab.

Google colab es una herramienta proporcionada por Google en la nube que permite ejecutar código en Python haciendo uso de sus GPU de forma gratuita. Esta al ser gratuita no tiene un rendimiento muy alto, además de no tener una disponibilidad

inmediata, pero hacía frente a la resolución de este problema, al no encontrar otra solución.

Finalmente, tras las diversas dificultades encontradas y distintos caminos estudiados, el modelo obtenido que más se ajusta a nuestro objetivo presenta:

Número de capas convolucionales	2
Número de filtros en capa convolucional	32
Batch-size	32
Épocas	50
Optimizador	Adam
Función de pérdida	binary_crossentropy
Métrica	Accuracy
Aumento de datos	Si
Dropout	0.5
Validación cruzada	Hold-out

7. Conclusiones

El trabajo realizado basado en el material de los diferentes notebooks de los cursos tutoriales al igual que en las referencias indicadas, detallados en los diversos capítulos. Nos ha posibilitado adquirir un aprendizaje y conocimiento en profundidad de Deep Learning. No solo alcanzando una formación teórica en profundidad, sino también, experimental.

El objetivo de este trabajo por una parte consistía en adquirir un aprendizaje en profundidad de la temática a desarrollar. Tanto a nivel documental, detallado en los capítulos “Introducción a la temática” y “Marco teórico”, como a nivel de implementación. Este último ha sido satisfecho por medio del conjunto de notebook desarrollados en el capítulo “Estudio previo”. En ellos hemos aprendido el desarrollo de diferentes modelos de redes neuronales convolucionales en función del tipo de problema, uso de diversas técnicas de regularización, técnicas de desbalanceo de datos, técnicas para solventar el sobreajuste de una red y transferencia de aprendizaje.

Este como ya hemos comentado, es un campo muy amplio el cual se encuentra en continuo crecimiento. Lo aprendido en dicho trabajo consiste en los conceptos y herramientas generales para desarrollar un modelo de red neuronal convolucional.

Por otra parte, como segundo objetivo, se basaba en el desarrollo de un sistema de clasificación de imágenes de pulmón en dos clases, con neumonía y sin neumonía haciendo uso del aprendizaje profundo.

El problema correspondía a un problema de clasificación binaria. Y su dataset estaba formado por un conjunto de imágenes de pulmón desbalanceadas.

En el presente estudio, concretamente en los capítulos “Reto kaggle” y “Discusión” hemos llevado a cabo el desarrollo de diferentes modelos de redes neuronales convolucionales que satisficieran nuestro segundo objetivo. Tras diversas configuraciones, entrenamiento y evaluación de modelos, el modelo que más se ajustaba a nuestro objetivo ha sido el modelo 3. A este modelo le han sido aplicadas diversas técnicas aprendidas en el estudio previo, estas técnicas han sido aplicadas en los puntos más vulnerables del dataset que nos llevaban a un mal entrenamiento, como el desbalanceo cuya técnica aplicada ha sido el Aumento de datos y el sobreajuste en el cual hemos aplicado la técnica de Dropout.

El modelo 3 alcanza una precisión del 96%, sin embargo, debido al desbalanceo que presentaba el dataset, el modelo no ha sido entrenado correctamente pese al 96% de acurracy ya que, al presentar unos datos desbalanceados, ha presentado casi una clasificación al 100% de la clase mayoritaria, perjudicando a la clase minoritaria, la clase 1 (sin neumonía).

El haber realizado este estudio y no haber obtenido resultados satisfactorios en cuanto al desarrollo de un modelo de red neuronales convolucional, me incita a continuar el trabajo de cara al futuro, desarrollando un modelo de red neuronal que satisfaga mis objetivos y realice de forma óptima la clasificación de las imágenes de pulmón en función de su patología.

De cara al futuro realizaremos modificaciones en la configuración de los hiperparámetros del modelo, implementaremos diversas técnicas de reducción del sobreajuste no solo Dropout y continuaremos el estudio para conocer otras existentes no detalladas en dicho estudio y evaluar la eficacia de estas.

Probaremos diversas técnicas de evaluación del entrenamiento como el método de validación cruzada k-fold, el cual se ajusta más a nuestro problema ya que es aplicado fundamentalmente en dataset pequeños y no instiga el sobreajuste.

Para la mitigación del desbalanceo, la técnica del aumento de datos de la más conocida, sin embargo, esta es más eficiente cuando la aplicación es solo y únicamente en la clase minoritaria.

Por otra parte, además de experimentar con diversos métodos y técnicas, podemos tratar la transferencia de aprendizaje.

El estudio contado en este documento es solo una pequeña parte del campo del Aprendizaje profundo. Este se encuentra en continuo crecimiento y está teniendo una gran repercusión en los diferentes campos hoy en día debido a sus satisfactorios resultados. Esto nos suscita a seguir trabajando en él, conocer más en profundidad los diferentes campos dentro del Aprendizaje profundo tanto a nivel teórico como experimental.

8. Referencias

Rouhiainen, L. (2018). Inteligencia artificial. Madrid: Alienta Editorial.

López, R. (2017, Abril 30). *Introducción a la Inteligencia Artificial.* Recuperado de: https://iaarhub.github.io/capacitacion/2017/04/30/introduccion-a-la-inteligencia-artificial/

López, R. (2017, Junio 13). *Introducción al Deep Learning.* Recuperado de: https://relopezbriega.github.io/blog/2017/06/13/introduccion-al-deep-learning/

Hernández, A. (2018, Julio 5). *Neurociencia, inspiración para el machine learning.* Recuperado de: https://mlearninglab.com/2018/07/05/neurociencias-un-camino-de-inspiracion-para-el-machine-learning/

De Paz, I. (2019, Febrero 13). *El papel de la Inteligencia Artificial en la actualidad.* Recuperado de: https://www.xeridia.com/blog/el-papel-de-la-inteligencia-artificial-en-la-actualidad

Julián, G. (2014, Diciembre 30). *Las redes neuronales: qué son y por qué están volviendo.* Recuperado de: https://www.xataka.com/robotica-e-ia/las-redes-neuronales-que-son-y-por-que-estan-volviendo

Chollet, F. (2018). Deep learning with Python . Shelter Island, N.Y: Manning Publications.

Utrera, J. (2018, Diciembre 5). *Deep Learning básico con Keras (Parte 4): ResNet.* Recuperado de https://enmilocalfunciona.io/deep-learning-basico-con-keras-parte-4-resnet/

Dewivedi, P. (2019, Enero 4). *Understanding and Coding a ResNet in Keras*. Recuperado de: https://towardsdatascience.com/understanding-and-coding-a-resnet-in-keras-446d7ff84d33

Durán, J. (2019, Enero 29). *Qué es la transferencia de aprendizaje y como aplicarla a tu red neuronal*. Recuperado de: https://medium.com/metadatos/qu%C3%A9-es-la-transferencia-de-aprendizaje-y-c%C3%B3mo-aplicarla-a-tu-red-neuronal-e0e120156e40

Bagnato, J. (2019, Mayo 16). *Clasificación con datos desbalanceados*. Recuperado de: https://www.aprendemachinelearning.com/clasificacion-con-datos-desbalanceados/

Torres i Viñals, J. (2020). Python deep learning: Introducción práctica con Keras y TensorFlow 2. Barcelona: Marcombo.

Bagnato, J. (2018, Noviembre 29). *¿Cómo funcionan las Convolutional Neural Networks? Visión por ordenador.* Recuperado de: https://www.aprendemachinelearning.com/como-funcionan-las-convolutional-neural-networks-vision-por-ordenador/

Peng, T. (2019, Marzo 7). *ICLR 2019 - "Fast as Adam & Good as SGD" – New optimizer Has both.* Recuperado de: https://medium.com/syncedreview/iclr-2019-fast-as-adam-good-as-sgd-new-optimizer-has-both-78e37e8f9a34

Pérez Planells, L., Delegido, J., Rivera-Caicedo, J. P., & Verrelst, J. (2015). Análisis de métodos de validación cruzada para la obtención robusta de parámetros biofísicos. Revista Española de Teledetección, 2015, vol. 44, p. 55-65.

López, J. (2019, Diciembre 11). ¿Qué son las llamadas manchas en los pulmones y por qué aparecen?. Recuperado de: https://www.topdoctors.es/articulos-medicos/que-son-las-llamadas-manchas-en-los-pulmones-y-por-que-aparecen

Stanford CS. *CS231n Convolutional neural networks for visual recognition.* Recuperado de: https://cs231n.github.io/optimization-2/

Buhigas, J. (2018, Febrero 14). *Todo lo que necesitas saber sobre tensorflow, la plataforma para inteligencia artificial de Google.* Recuperado de: https://puentesdigitales.com/2018/02/14/todo-lo-que-necesitas-saber-sobre-tensorflow-la-plataforma-para-inteligencia-artificial-de-google/