Proyecto Fin de Carrera

Grado en Ingeniería de las Tecnologías de Telecomunicación

Clasificación de imágenes mediante Redes Neuronales Convolucionales y técnicas de Deep Learning avanzadas: Transformers

Autor: Iván Matas González

Tutor: Francisco José Simois Tirado

Dpto. Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2021

Proyecto Fin de Carrera
Grado en Ingeniería de las Tecnologías de Telecomunicación

Clasificación de imágenes mediante Redes Neuronales Convolucionales y técnicas de Deep Learning avanzadas: Transformers

Autor:
Iván Matas González

Tutor:
Francisco José Simois Tirado
Profesor Contratado Doctor

Dpto. de Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2021

Proyecto Fin de Carrera: Clasificación de imágenes mediante Redes Neuronales Convolucionales y técnicas de Deep Learning avanzadas: Transformers

Autor: Iván Matas González

Tutor: Francisco José Simois Tirado

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Agradecimientos

Estar escribiendo estas palabras significa que se acaba, se acaba esta preciosa etapa en la cual no solo he aprendido conocimientos teóricos, sino que también he podido aprender que soy capaz de todo lo que me proponga por muy difícil que sea, he aprendido que soy mucho más fuerte de lo que pensaba y que pese a las dificultades que se pongan en el camino siempre habrá una solución y con esfuerzo y disciplina se conseguirá.

Dar las gracias a todos y cada uno de los profesores que me han dado clase durante estos cuatro años de grado por transmitirnos sus conocimientos de la mejor forma posible, intentando así hacernos personas lo más competentes posible dentro del campo de las comunicaciones. Sobre todo, me gustaría agradecerle a mi tutor, Francisco José Simois, haberme enseñado este maravilloso mundo del Deep Learning y porque pese a las dificultades me ha ayudado en todo lo posible y como no a Juan Antonio Becerra por ser como es, por siempre intentar hacernos mejores en todos los sentidos, por profesores como tú merece la pena esta etapa.

También agradecer a los compañeros de carrera que gracias a vosotros todo esto ha sido más fácil, todas y cada una de las personas que me ha acompañado en este camino ha sido maravillosa y ha sabido aportar su granito de arena y como no, a mis amigos de Puertollano que pese a la distancia y a pasarme estudiando el 90% del tiempo siempre han estado ahí para apoyarme y hacer que todo esto sea mucho más fácil.

Y como no podría ser de otra forma, dar las infinitas gracias a mis padres porque sin ellos nada de esto habría sido posible, porque siempre me han ayudado y apoyado, tanto en los momentos buenos, como en los malos, porque hasta cuando yo no tenía más fuerzas para continuar, ellos estaban ahí para darme esas fuerzas que necesitaba. Pero sobre todo a mi madre, Yolanda González Sánchez, que mejor que nadie sabe que sin ella no sería la persona que soy hoy ni estaría donde estoy hoy, simplemente gracias por ser tú, este TFG va por ti mamá allá donde estés.

Iván Matas González

Sevilla, 2021

Resumen

En este documento se profundizará en el campo del Deep Learning, desde los conceptos más básicos como son el modelo Perceptrón, las redes Multilayer Perceptron, etc. Hasta llegar a uno de los modelos más avanzados de la actualidad, como son las arquitecturas Transformer y Visual Transformer (ViT). Por lo cual este documento estará centrado prácticamente en su totalidad en la clasificación de imágenes, imágenes médicas en este caso. Para cada uno de los modelos que se verán durante el desarrollo del mismo se realizará tanto una explicación teórica profunda como un análisis matemático de sus partes más importantes. Además, una vez realizada dicha explicación se realizará una implementación en código Python de los conceptos desarrollados en los apartados anteriores y con una base de datos de imágenes médicas clasificándolas según la patología (normal - neumonía), pudiendo así ser capaces de realizar una comparación experimental de una Red Convolucional con una arquitectura ViT, poseyendo ambas las mismas condiciones de partida.

Con la realización de este documento se pretende haber llegado a comprender el funcionamiento característico de esta nueva arquitectura (modelo codificador-decodificador) y ser capaces de mostrar el potencial de estas aun estando en un nivel de desarrollo muy temprano.

Abstract

In this paper we will go deep into the Deep Learning field, from the most basic concepts such as the Perceptron model, Multilayer Perceptron networks and so on. Until reaching one of the most advanced models of today, such as the Transformer and Visual Transformer (ViT) architectures. Therefore, this document will be focused almost entirely on image classification, in this case medical images. For each of the models that will be seen during the development of this paper, both a deep theoretical explanation and a mathematical analysis of its most important parts will be carried out. In addition, once this explanation is done, an implementation in Python code of the concepts developed in the previous sections and with a database of medical images, classifying them according to the pathology (normal - pneumonia), will be carried out, thus being able to make an experimental comparison of a Convolutional Network with a ViT architecture, both having the same starting conditions.

With the realization of this paper, we intend to have come to understand the characteristic performance of this new architecture (encoder-decoder model) and to be able to show the potential of these still at a very early stage of development.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Lista de Acrónimos

IA	**Intelligence Artificial**
ANN	**Artificial Neural Network**
CNN	**Convolutional Neural Network**
Tanh	**Tangente Hiperbólica**
ReLu	**Rectified Linear Unit**
RFF	**Red neuronal Feed-Forward**
FFN	**Feed-Forward Network**
NLP	**Natural Language Processing**
ViT	**Visual Transformers**
MLP	**Multi Layer Perceptron**
MSA	**Multi Self-Attention**
LN	**Layer Normalization**

Notación

b_k	Bias (sesgo)
$\sum_i$	Sumatorio
x_{ki}	Señal i-ésima de entrada
w_{ki}	Peso i-ésimo
α	Valores de la diagonal en la técnica Attention
Q	Vector Query de la técnica Attention
K	Vector Key de la técnica Attention
V	Vector Value de la técnica Attention
K_N	N-ésimo valor del vector Key de la técnica Attention
V_N	N-ésimo valor del vector Value de la técnica Attention
K^T	Vector Key de la técnica Attention traspuesto
d_k	Dimensión del vector Query, Value o Key
K_i	i-ésimo valor del vector Key de la técnica Attention
V_i	i-ésimo valor del vector Value de la técnica Attention
Q_i	i-ésimo valor del vector Query de la técnica Attention
↔	Sí y solo sí
∀	Para todo
|	Tal que
∈	Perteneciente a
[x,y]	Intervalo entre x e y, ambos incluidos en el intervalo
e^x	Exponencial de x
W_i^Q	i-ésima posición del vector de proyección para el vector Query
W_i^K	i-ésima posición del vector de proyección para el vector Key
W_i^V	i-ésima posición del vector de proyección para el vector Value

1 INTRODUCCIÓN

Durante el desarrollo de este apartado se detallarán los motivos por los cuales se tomó la determinación de realizar este TFG sobre el campo del Deep Learning y por qué la elección de las arquitecturas que han sido implementadas para tanto la parte experimental como para la explicación teórica. Por último, se mostrarán los objetivos que se pretenden cubrir con la realización del mismo y un breve resumen de la estructura del trabajo por puntos.

1.1 Motivación

En los últimos años se está escuchando que en multitud de servicios o bien se ha implementado el Deep Learning o bien se está pensando cómo se podría implementar, por lo cual, el objetivo una vez vista la introducción, es implementar diferentes tipos de arquitecturas, Redes Neuronales Convolucionales y Transformer, con el fin de cubrir un mismo problema y así poder ver diferentes puntos de vista.

En cuanto a la elección de la base de datos se tuvieron multitud de opciones, pero al final se eligió una base de datos con imágenes médicas. Estas presentaban radiografías con dos patologías, por un lado, se tienen radiografías de personas en estado "normal" y por otro lado personas con un estado de "neumonía".

Una vez realizadas las arquitecturas y visto como trabaja cada una de ellas, se puede hacer la técnica denominada "Transfer Learning", es decir, coger la arquitectura y adaptarla a cualquier BBDD con la que se quiera implementar, ajustando también sus parámetros obviamente. Con ello, se pueden ajustar a diferentes objetivos como, por ejemplo, imágenes médicas con diferentes tipos de lunares y pudiendo así predecir cual puede conllevar a tumores de piel o bien un BBDD con resonancias magnéticas del cráneo pudiendo así detectar meningiomas peligrosos para la vida. Como se puede apreciar la cantidad de aplicaciones es innumerable.

1.2 Objetivos

El principal de este objetivo de este trabajo es aprender y adentrarse en este maravilloso mundo del Machine Learning y del Deep Learning. Durante la realización de este se pretenden cubrir diferentes metas u objetivos dentro del trabajo:

1. Aprendizaje y aclaración de los conceptos principales del campo en cuestión.
2. Implementación de una red convolucional y el posterior análisis de los resultados.
3. Desarrollo teórico-matemático de la arquitectura Transformer.
4. Desarrollo teórico-matemático de la arquitectura ViT (Visual Transformer).
5. Implementación de la arquitectura ViT y el posterior análisis de los resultados.

1.3 Desarrollo del trabajo

Para la correcta comprensión de este documento se recomienda leerlo en la forma en la cual se presenta, para ello se ha seguido una básica clasificación general, en primer lugar, se realizará una descripción teórica-matemática de los conceptos y de las arquitecturas que van a ser implementadas. En segundo lugar, se mostrará la implementación de ambas arquitecturas con sus diferentes resultados y, por último, se encontrarán los códigos de los cuales se han hablado durante el desarrollo de este último punto.

Para ser más concreto en el contenido que se puede encontrar en este documento se muestra a continuación un breve resumen por puntos:

1. **Introducción.** Apartado en el cual se está desarrollando este punto. Se han detallado tanto la motivación como los objetivos de este TFG.
2. **Conceptos básicos.** En este apartado se pretenden repasar y aclarar los conceptos más básicos del campo del Deep Learning con los cuales serán desarrollados el resto de los apartados como son el perceptrón, la MLP (Multi Layer Perceptrón), las redes convolucionales y la técnica Data Augmentation.
3. **Modelos propuestos.** Con el desarrollo de este apartado se pretende dar una clara explicación de los modelos que se han propuesto para este trabajo los cuales son las redes convolucionales y la nueva arquitectura Transformer.
4. **Propuesta de trabajo.** Se hace una aclaración del equipo Hardware empleado para la realización del documento y la realización del código y su posterior experimentación, además, se hará una aclaración del Software empleado tanto las librerías empleadas, como los paquetes y los programas extras.
5. **Código y resultados.** En este apartado se muestran diferentes capturas del código y se realiza una detallada explicación de cada uno de los fragmentos intentando así solventar cualquier tipo de dudas que puedan surgir.
6. **Conclusiones y líneas futuras.** Como último apartado se realizará una evaluación final tanto del trabajo como de cada una de las arquitecturas y, por último, se hablará de las líneas futuras de trabajo de las redes Transformer que como se verá es muy amplio.

2 CONCEPTOS BÁSICOS

Durante el desarrollo de ese apartado, se detallará tanto la evolución histórica como los diferentes modelos que han sido desarrollados para las diferentes necesidades que han surgido a lo largo de la evolución tecnológica, desde el modelo más básico, el perceptrón, hasta uno de los cambios más revolucionarios en este campo, las redes neuronales convolucionales.

2.1 Historia

Alrededor del año 1950 comenzó la evolución de la informática y con ella se empezó a hablar del término Inteligencia Artificial (IA). Alan Turing fue de los primeros en preguntarse si un ordenador era o no, una maquina inteligente. En 1950 inició su investigación, la cual fue anotada en el artículo "Computing machinery and intelligence". Para llevar a cabo dicha investigación creó el denominado, test de Turing, dicho test consiste en meter a varias personas en una habitación cerrada herméticamente y ponerlas a conversar mediante un chat con otra persona o un ordenador, si esas personas no son capaces de identificar que detrás de ese chat era un ordenador, entonces, se considera que el ordenador es inteligente. [1] Como se puede apreciar, esta definición no tiene nada que ver con la actual sobre inteligencia artificial, es por esto por lo que del terminó IA apareció el nuevo subconjunto denominado Machine Learning que a su vez se subdividió en el famoso Deep Learning.

Sobre el año 1980 comenzó el desarrollo del Machine Learning, este subconjunto se centra en la capacidad que tiene una determinada computadora para recibir datos preprocesados con antelación ser capaz de aprender por si sola, ofreciendo al final de dicho entrenamiento el modelo más óptimo posible. El primero en implementar dicha tecnología fue Gerald Dejong mediante el método "Explanation Based Learning" en el año 1981, dicho método era capaz una vez finalizado el entrenamiento, crear unas reglas para descartar los datos menos importantes de cualquier base de datos dada. [2] [3]

En 2006, Geoffrey Hinton fue el primero en hacer uso de la palabra Deep Learning para definir un nuevo de tipo de redes neuronales capaces de aprender mucho mejor y más rápido. La creciente evolución de esta tecnología se vio fomentada por el aumento de la computación gracias a las tarjetas gráficas. En el año 2011 el ordenador Watson (de la empresa IBM) fue el primero en derrotar a humanos en el concurso Jeopardy contestando preguntas formuladas por el presentador, es decir, reconocimiento de voz humana a tiempo real. [2] [3]

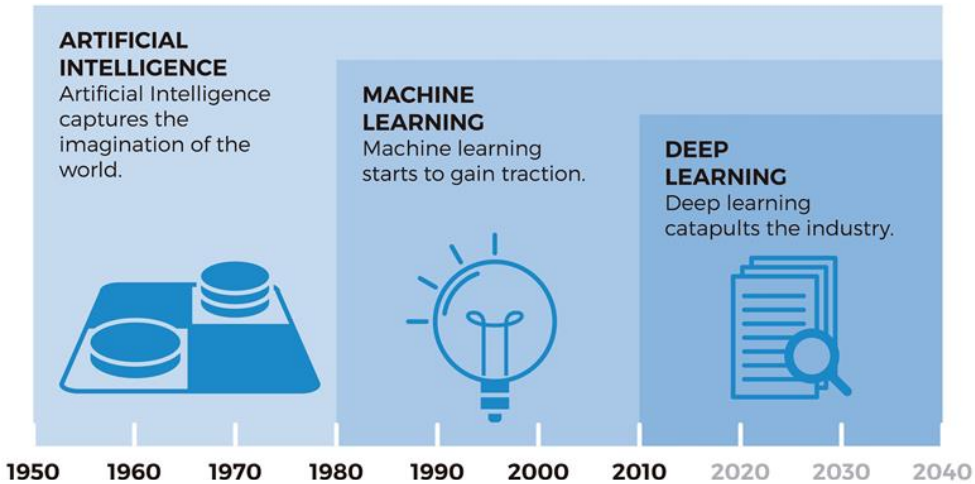


Figura 1. Evolución histórica [4]

En la actualidad, miles de empresas usan estas tecnologías para ofrecer mejor servicio a sus clientes, para tratamiento de datos con fin de mejorar sus prestaciones, automatización de ciudades o fábricas, etc. Es por ello, que se espera que, en un futuro no muy lejano, esta tecnología tenga una importancia aún mayor y permita desarrollarse ampliamente en otros campos que hoy en día no son aplicables.

2.2 Redes Neuronales Artificiales

Como su propio nombre indica, las redes neuronales artificiales (ANN), basan su estructura y funcionamiento en el concepto biológico de una neurona. Para la realización de los modelos de las redes neuronales existen diferentes tipos y existen diferentes técnicas de optimización para conseguir el resultado esperado.

Como norma general, todos los tipos de ANN que se nombrarán a continuación tienen una estructura principal muy similar:

- **Capa de entrada:** Es la primera capa de la red, esta debe de tener el tamaño correcto acorde con el vector de entrada.
- **Capas escondidas:** Son denominadas las capas intermedias, el número de estas no es trivial y dependerá del problema a abordar.
- **Capa de salida:** Es la última capa de la red y es la encargada de devolver el resultado del problema tratado.

2.2.1 Perceptrón

Las neuronas (perceptrones) empleadas en los modelos, siguen el modelo neuronal de McCulloch-Pitts, este es el primer modelo matemático planteado para una neurona artificial (véase la figura 2).

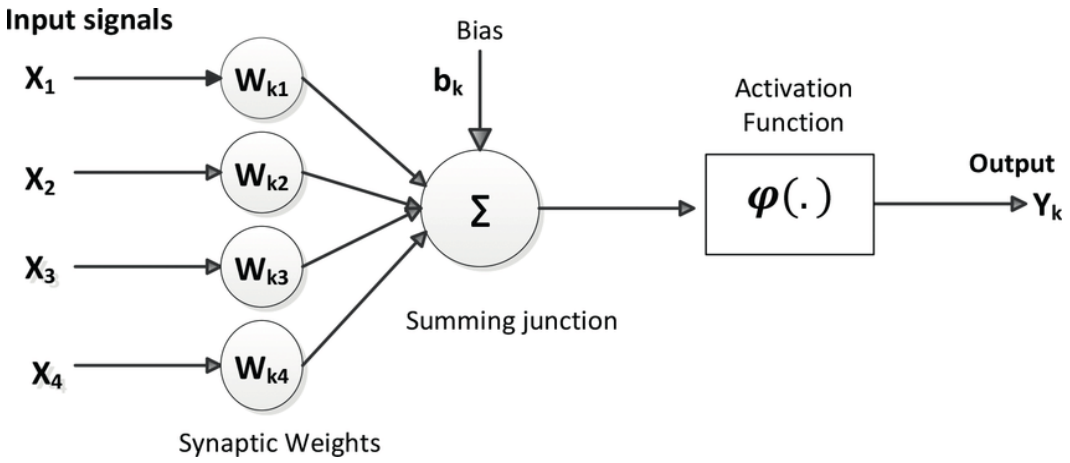


Figura 2. Modelo neuronal de McCulloch-Pitts [6]

$$f(x) = b_k + \sum_i w_{ki} x_{ki} \tag{2.1}$$

Como se puede ver en la figura anterior, el sistema se basa principalmente en lo que se denominan pesos (W_{kn}) que equivalen a las conexiones sinápticas de una neurona biológica. Estos pesos son asignados a las entradas que recibe la neurona, que posteriormente se suman, fórmula 2.1, y son pasadas por la denominada función de activación, estas funciones de activación tienen el objetivo de hacer perder la linealidad de la solución ofrecida por el perceptrón en cuestión. Existen numerosas funciones de activación que se han ido proponiendo a lo largo de la evolución de las redes neuronales para los diferentes tipos de problemas, se pueden ver algunas de ellas en la figura 3. [5]

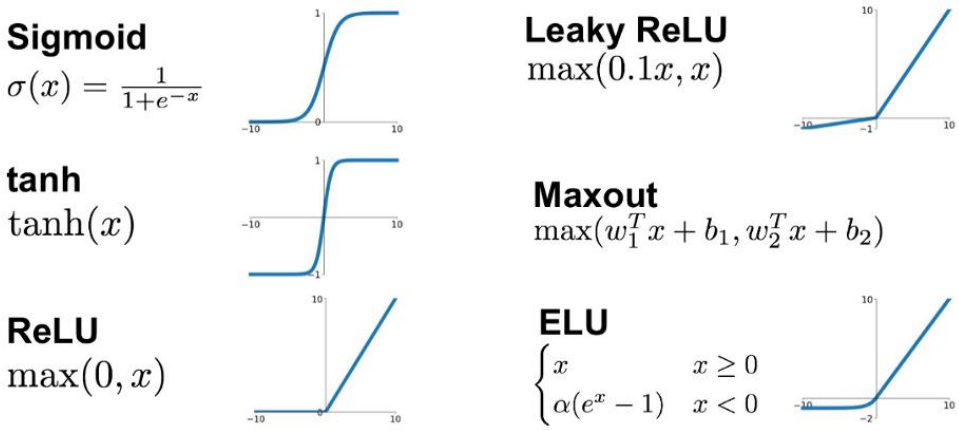


Figura 3. Tipos de funciones de activación [9]

- La primera y por tanto más antigua de todas es la **Sigmoide**. Normaliza todos los valores de entrada en un rango de salida entre 0 y 1.
- La **Tanh** es muy parecida a la Sigmoide, pero normaliza en un rango entre -1 y 1. Por esta razón es la función óptima para problemas de decisión entre dos opciones.
- La función **ReLu** es una función causal, es decir, anula los valores negativos y los positivos los deja tal y como se reciben. Este cambio hace que la computación sea mucho menos costosa.
- **Leaky ReLu** es una función muy parecida a la vista anteriormente, función ReLu, con la diferencia de que los valores negativos son multiplicados por un pequeño valor. Este cambio subsana uno de los problemas de la función ReLu que al tener mucha carga se perdían valores y algunas neuronas "morían" por inactividad, este fenómeno es denominado "Dying ReLu problem".
- **Maxout** es una función de activación que usa el máximo de las entradas recibidas.
- La función **ELU** es muy parecida a la anterior Leaky ReLu, pero es más suave en el cambio de valores negativos hacia positivos, esto hace que sea más precisa en sus resultados.

Esto solo son algunos ejemplos, existen muchas otras como la Softmax, PReLU, Tangente hiperbólica, etc. Cada una de ellas soluciona un tipo de problema o son más eficientes en un tipo de problema en concreto. Además, algunas de ellas nacieron de los errores de las primeras, como pasa con la Leaky ReLu y la ReLu, por ejemplo. [7] [8]

Este primer modelo de red neuronal sirvió para resolver problemas básicos, los cuales eran prácticamente binarios, clasificación entre dos categorías, rectas de regresión, análisis básico de datos o incluso como implementación de puertas lógicas. La limitación de este modelo llegó con los problemas no lineales, con las clasificaciones como puertas XOR o con más de una categoría.

2.2.2 Perceptrón multicapa

Esta estructura de red (más de una capa) más básica que existe y es la evolución del perceptrón simple. En este modelo la información solo sigue un camino, hacia adelante, no existen bucles ni retroalimentación, también puede ser denominada red neuronal Feed-forward (RFF).

Este tipo de red está formado por capas como se comentó anteriormente, en cada una de estas capas existen diversas neuronas (perceptrones) de las analizadas en el punto anterior (apartado 2.2.1). Tanto el número de capas como el número de neuronas dependerá del problema a analizar, ya que cuanto más complejo sea más grande tendrá que ser la red y por lo tanto necesitará una computación.

Existen diversas ventajas que hacen que este modelo se siga usando hoy en día, entre ellas se encuentra la baja complejidad (dependiendo del problema), realizar este modelo de red hoy en día es una tarea sencilla tanto de programar como de mantenimiento debido a que es un tipo de red muy común y de la cual se tiene gran cantidad de información. Además, al ser propagación en un único sentido es más rápida que otros modelos. Como principal desventaja, se puede nombrar que al ser uno de los primeros modelos planteados, no es válido para Deep Learning, ya que serían necesarias capas densas y propagación inversa.

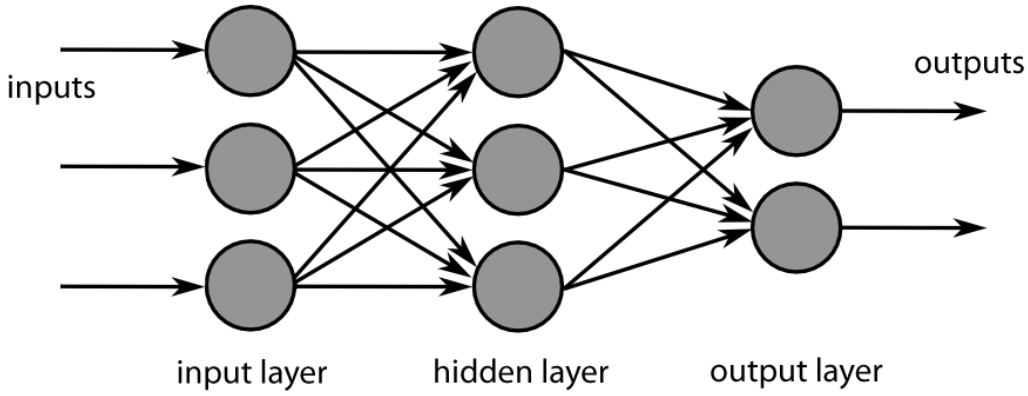


Figura 4. Red neuronal Feed-Forward o Multilayer Perceptron (MLP) [10]

2.2.3 Redes Neuronales Convolucionales

Cuando se habla de redes neuronales convolucionales (CNN), posiblemente se esté hablando de uno de los cambios más revolucionarios en el campo de las ANN. Este tipo de modelo fue creado exclusivamente para el reconocimiento de imágenes. Esta arquitectura surgió debido a que con las ANN convencionales había que definir las características visuales a mano para cada uno de los modelos en vez de aprenderlos conforme avanza el entrenamiento como sucede con las CNN.

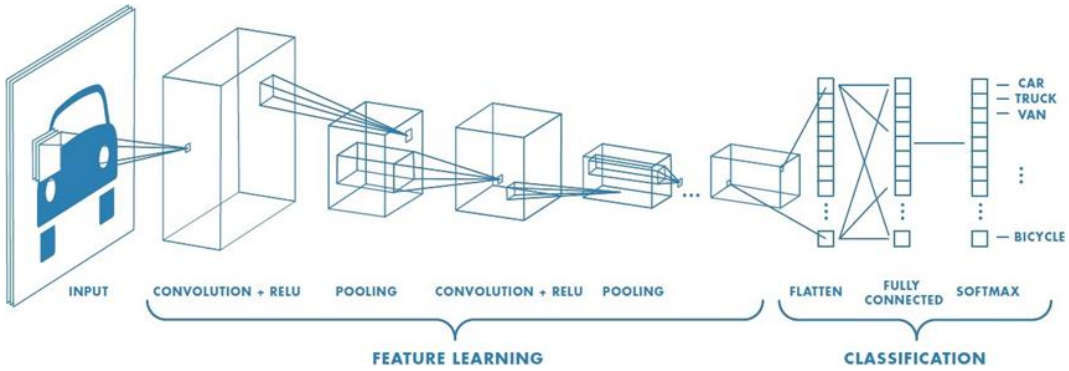


Figura 5. Arquitectura de una CNN [17]

En la figura 5 se puede ver cómo es la arquitectura de las CNN, como se puede apreciar, esta arquitectura es muy diferente a la vista hasta el momento en las RFF (figura 4). Como se vio en el apartado anterior, las ANN convencionales basaban su funcionamiento en una suma ponderada de los elementos de entrada por unos ciertos pesos, los cuales se iban actualizando durante el entrenamiento para obtener el mejor resultado posible. En este nuevo modelo, cada capa se encargará de detectar unos tipos de características de la imagen que se tenga como entrada y secuencialmente se irán recorriendo todas y cada una de las capas diseñadas. Una vez se recorra la red convolucional, la salida se le pasa a una ANN convencional que se encargará de hacer la clasificación. Es decir, la CNN se encarga de detectar las características y la salida se le pasa a la ANN convencional para que clasifique la imagen según las características extraídas. [11] [12] [13]

En primer lugar, el concepto más importante en este nuevo modelo de ANN es el denominado Kernel o filtro. El kernel es una matriz de tamaño NxN, el tamaño depende del problema en cuestión, el cual es convolucionado por toda la imagen creando un único píxel por cada una de estas convoluciones (véase la figura 6). Con esta característica lo que se consigue es que conforme se avanza de capa, la imagen analizada se va reduciendo su resolución. Pero, ¿es esto positivo? Aunque en un primer momento parezca que no porque se piense que se está perdiendo información no es así por las características que se verán a continuación. [14]

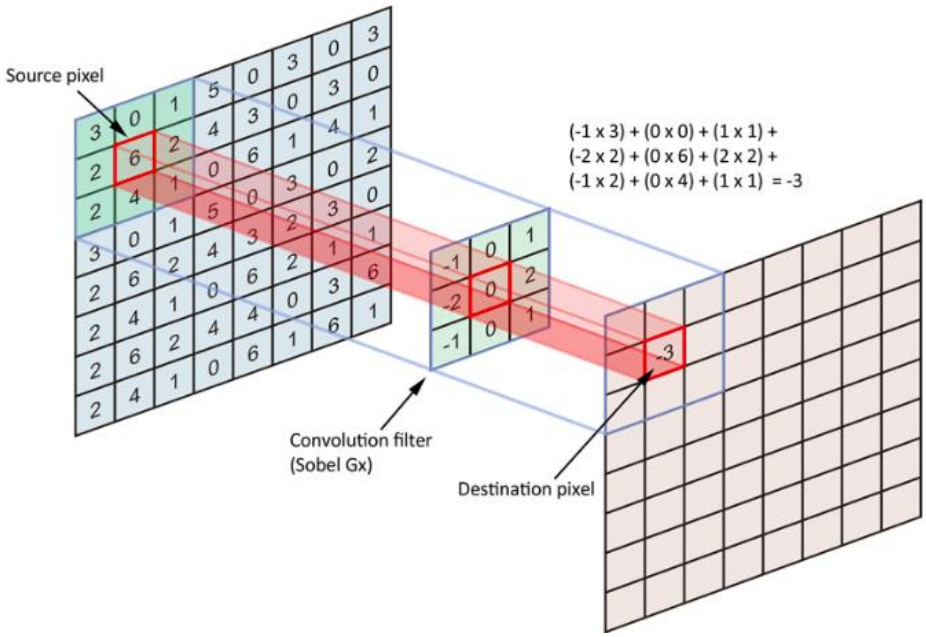


Figura 6. Uso del Kernel [15]

En segundo lugar, si se analiza el funcionamiento, es necesario hacer referencia de nuevo al Kernel. Cuando se ha explicado anteriormente que era esta herramienta, se ha visto que hacía la imagen más pequeña conforme pasaba por las capas, pero su principal función es que es capaz de sacar características de la imagen que está analizando. Es decir, si se comienza el análisis por las primeras capas en las cuales la imagen todavía es grande (resolución), las características que detecta el Kernel son básicas como pueden ser cambios de niveles, contornos, líneas, etc. Pero conforme se va haciendo más pequeña la imagen, el mapa de características (características analizadas en cada capa) se va haciendo más ancho y esto conlleva a que las características analizadas sean más detallistas, así como ojos, boca o pelo en el caso de personas; ruedas, puertas o marca en el caso de un coche, por ejemplo. [14] [13]

Es por esto por lo que la arquitectura de las CNN es denominada como forma cónica, porque conforme aumenta el número de capas, disminuye su tamaño (resolución), pero aumenta el grosor de estas debido al aumento del mapa de características de cada una de estas capas. En la figura 7 se puede apreciar el gran cambio entre las primeras capas y las últimas y con ello la característica forma cónica.

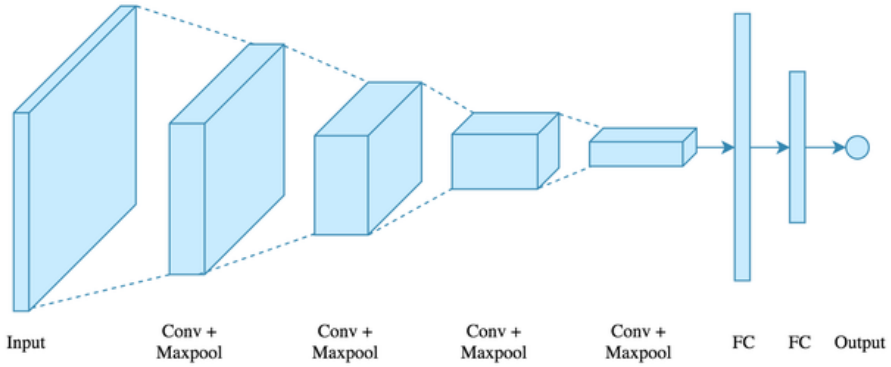


Figura 7. Cambio en la arquitectura de una CNN [16]

A continuación, en la figura 8 se va a mostrar gráficamente el funcionamiento de los mapas de características en cada una de las capas que recorrería la imagen. Es importante recalcar, que es un ejemplo teórico, las características analizadas por la red dependerá del diseño que se programe en cada momento, pero es cierto que en la capa de entrada y su colindante siempre se detectan características básicas y en la capa de salida y su anterior se detectan características muy detallistas, la "incertidumbre" está en las capas escondidas. A la salida de esta parte convolucional se tiene una ANN convencional que se encargaría de realizar la clasificación

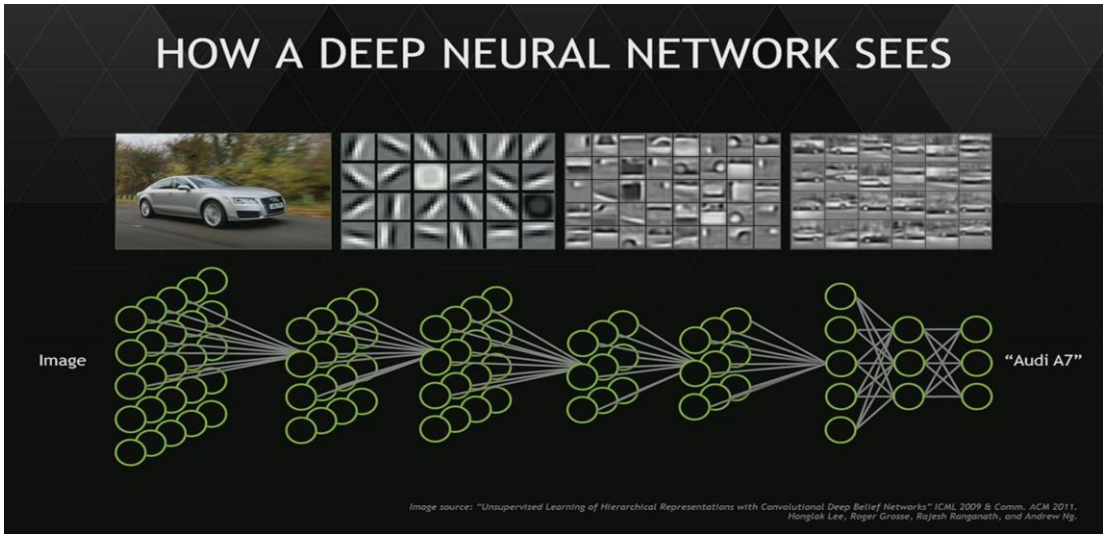


Figura 8. Mapa de características de una CNN [18]

Si se habla de las ventajas de esta arquitectura, obviamente hay que centrarse en la alta capacidad de procesado de imágenes, ya que este modelo se ha creado por y para este fin. Con el modelo convencional de ANN si se quería hacer estas mismas funciones se necesitaría mucha más computación y tiempo para obtener el mismo resultado o incluso un peor resultado. Si se quiere nombrar alguna desventaja de este tipo de redes, el rendimiento que necesitan es muy alto, es decir, son redes que trabajan muy bien con las tarjetas gráficas dedicadas como las de NVIDIA o AMD, además de que es bastante común que sea necesario implementar múltiples técnicas de optimización para alcanzar el rendimiento buscado. [11] [12] [13] [17]

2.3 Procesamiento de los datos

Una de las partes más importantes en el mundo del Deep Learning es el preprocesamiento de los datos. Esta técnica no es más que aplicar diferentes técnicas a cada uno de los datos para que así la red no aprenda de forma lineal y se produzca el denominado efecto de "overfitting", esto se produce debido a que la red es capaz de "memorizar" los resultados, pero no es capaz de aprender a realizar la tarea propuesta. Uno de los grandes problemas es encontrar el punto intermedio entre encontrar unos buenos resultados de entrenamiento y no estar en situación de overfitting o underfitting como se muestra en la figura 9.

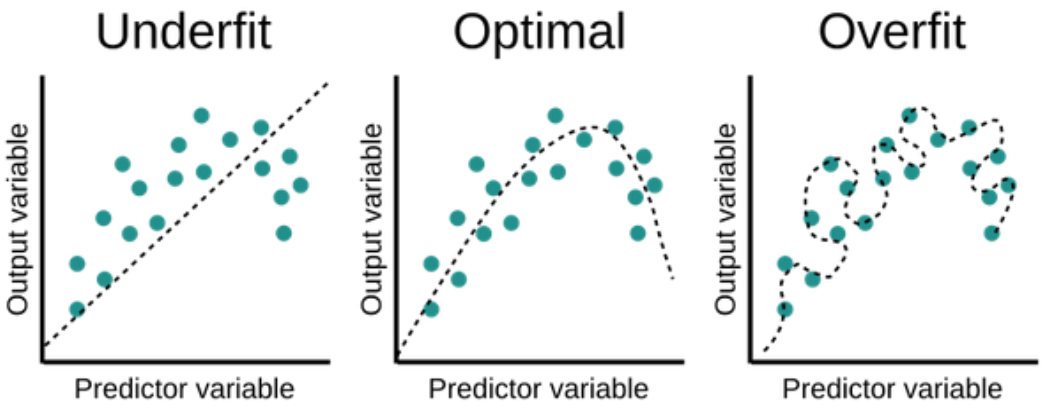


Figura 9. Etapas durante el entrenamiento [19]

Como se ha adelantado anteriormente, esto se consigue con el preprocesamiento de los datos. Cada tipo de problema, cada autor o cada tipo de red con la cual se quiera trabajar puede hacer uso de un preprocesamiento de datos distinto, por lo cual en este apartado se van a detallar los diferentes preprocesados que se le pueden realizar a una base de datos con la cual se va a entrenar a una CNN para clasificación de imágenes, pero destacar que existen muchas otras técnicas que se pueden aplicar a otro tipo de problemas.

1. La técnica más común y que se suele aplicar en todo tipo de base de datos es la normalización y centrado en cero (figura 10). La normalización es usada para intentar que el rango dinámico sea lo más constante posible, consiguiendo así menor computación y una mayor exactitud en los resultados. El centrar en cero los datos se usa para conseguir una mayor optimización mediante el descenso de gradiente, ya que lo normal es que existan datos tanto positivos como negativos y unos pueden estar muy alejados de otros por lo cual la actualización del gradiente puede ser muy grande y este efecto dificulta enormemente la optimización. [20]

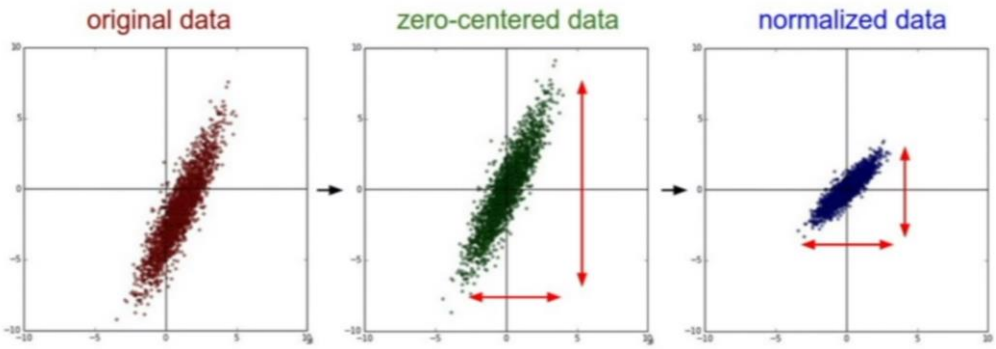


Figura 10. Centrado en cero y normalización de los datos [20]

2. Técnica denominada “Data augmentation”, con esta técnica se consiguen principalmente dos objetivos, el primero de ellos es el que ya se ha comentado, conseguir que la red no aprenda linealmente al darle diferentes puntos de vista como se verá a continuación y, en segundo lugar, se consigue aumentar la BBDD (Base de datos), que en el caso de tener una BBDD relativamente pequeña es de vital importancia. Esta técnica se puede implementar con múltiples librerías como son Keras, Scikit-Learn entre las más famosas, pero ambas librerías permiten realizar las mismas modificaciones a las imágenes, a continuación, veremos las más importantes y usuales: [21] [22]

    - **Voltear.** Esta técnica como bien su nombre indica, no consiste más que en darle la vuelta a la imagen tanto verticalmente como horizontalmente.

Figura 11. Flip vertical y horizontal [21]

    - **Rotación.** Permite rotar la imagen 180 grados en cualquier dirección, por lo cual hay hasta cuatro posibilidades y si se realizan dos rotaciones en la misma dirección se tiene una operación de voltear. Además, se puede especificar un porcentaje de inclinación en concreto en vez de dejar el de por defecto (180º).

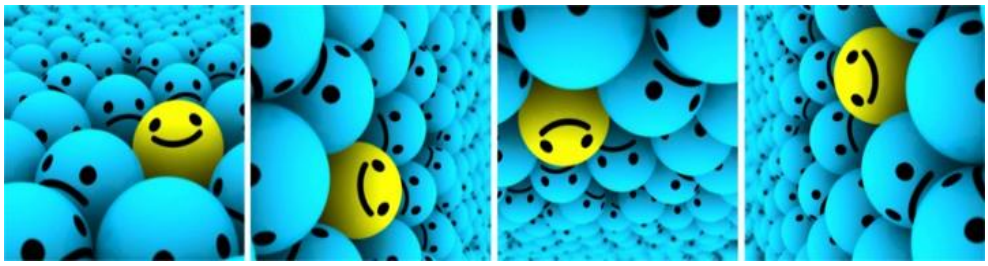


Figura 12. Rotación 180º en todas direcciones [21]

    - **Reescalado.** Permite reescalar la imagen tanto positivamente como negativamente, es decir, hacerla más pequeña o hacerla más grande.

Figura 13. Reescalado positivo [21]

- **Zoom.** Esta característica realiza zoom a una zona aleatoria de la imagen, teniendo que posteriormente reescalar al tamaño original de la imagen. La parte negativa de esta técnica es que se pierde calidad debido al reescalado.

Figura 14. Zoom aleatorio [21]

Estos solo son algunos ejemplos de todas las características que se pueden aplicar. En el caso de este trabajo se ha implementado mediante la librería Keras con la clase ImageDataGenerator en el caso de la CNN y con la librería de TensorFlow para la segunda arquitectura. Todo ello se puede ver con detalle en el apartado 4.2.2. En la figura 15 se puede ver una representación visual de como se hace más grande la BBDD.

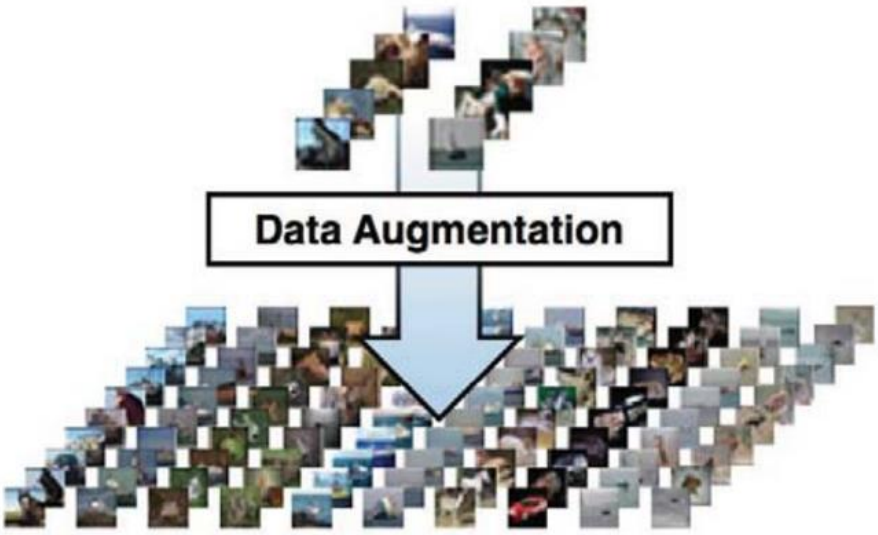


Figura 15. Representación de Data Augmentation [22]

3 ARQUITECTURA TRANSFORMER

En el año 2017 fue presentado el paper “Attention Is All You Need” publicado por el autor Ashish Vaswani y su equipo de Google Brain. [29] En este documento se detallaba una nueva estructura basada en una técnica ya existente anteriormente denominada “Attention” y presentaba datos muy prometedores comparando este modelo con las redes neuronales convencionales y con las redes neuronales recurrentes. Al estar basada en la técnica “Attention” antes de explicar cómo funciona este nuevo modelo se verá detalladamente dicha técnica. Esta arquitectura está centrada en el NLP (Natural Language Processing), pero es necesario verla para posteriormente ver la arquitectura Transformer para reconocimiento de imágenes (ViT).

3.1 Técnica Attention

La técnica “Attention” fue creada para el ámbito de la traducción mediante Deep Learning y posteriormente fue adaptada al mundo del reconocimiento de imágenes con las CNN. Cuando se comenzó su desarrollo se hizo respondiendo a la siguiente pregunta ¿cómo traduce la mente humana? Cuando se realiza una traducción hay partes de las oraciones que son más importantes que otras y oraciones que necesitan el contexto anterior para tener un sentido, por ejemplo, si se quiere traducir la frase: Juan es un buen chico, al idioma Ingles, las palabras Juan, buen y chico son las más importantes y hay que tenerlas en cuenta conforme se avance durante la traducción si se mantiene le contexto. Posteriormente esta idea se extrapoló al reconocimiento de imágenes y es que, cuando se quiere reconocer una fotografía de un retrato de una persona, los elementos característicos son los ojos, la boca, la nariz, etc. A su vez, es importante tener todas estas características presentes mientras se analiza dicha fotografía para terminar clasificándola como una persona. En la figura 16 se puede apreciar como esta técnica se centra en diferentes zonas de la imagen conforme va pasando por las diferentes capas.

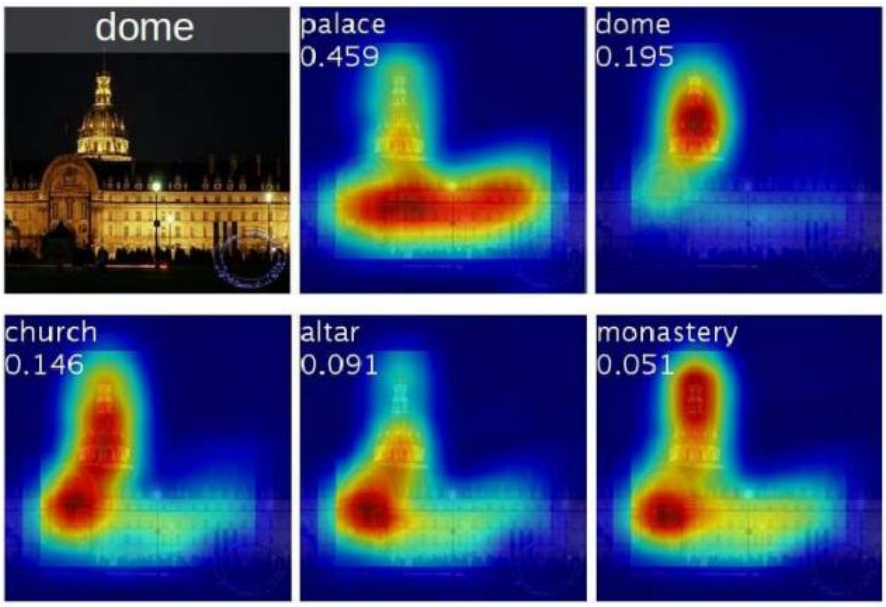


Figura 16. Representación por mapa del calor de la técnica Attention [26]

Como se puede apreciar en la figura anterior conforme se avanza de época la técnica se va focalizando las partes más importantes de la fotografía de entrada y le va asignando diferentes probabilidades para cada una de las características predefinidas (esquina superior izquierda en cada uno de los mapas de calor). Al contrario que con una ANN convencional, con la cual la entrada serían los valores de cada píxel de la imagen como si de una variable independiente se tratara, por lo cual, no se tendría en cuenta la posición de estos. Como es obvio, el concepto de la posición espacial de los elementos es muy importante en el reconocimiento de imágenes.

A continuación, se va a detallar tanto matemáticamente como visualmente el algoritmo empleado detrás de esta técnica. Para realizar dicha explicación se va a usar un ejemplo de NLP (Procesamiento del Lenguaje Natural), debido a que en un primer momento fue modelado para este fin y es más visual a la hora de realizar los esquemas. Se va a usar un ejemplo básico con la frase: *Me gusta ir de vacaciones*.

	Me	gusta	ir	de	vacaciones
Me	α				
gusta		α			
ir			α		
de				α	
vacaciones					α

Tabla 1. Representación de los valores asignados por Attention

En la tabla 1 anterior se puede apreciar un ejemplo de los valores que asignaría esta técnica para cada una de las relaciones entre las diferentes palabras, cabe destacar que cada uno de los colores representa un valor diferente asignado cuanto más oscuro sea un valor más bajo tendrá y por el contrario cuanto más vivo sea tendrá un valor más alto, la diagonal tiene asignado un valor α ya que dependerá del resultado final, lo ideal sería que este fuese el valor máximo de la tabla. Para realizar estas asignaciones se emplea un símil con la técnica de proyección SQL, ya que es empleada en dicho lenguaje, esta técnica consiste en una tabla Key-Value, vectores K y V respectivamente. Una vez obtenida la llave se busca en la tabla y te devuelve el valor Value (véase la tabla 2), solo quedaría definir el vector Q (Query) que serán cada una de las palabras de la oración anterior. [24] [25] [27]

Key	**Value**
K_1	V_1
K_2	V_2
.	.
.	.
.	.
K_N	V_N

Tabla 2. Proyección SQL

Para calcular el valor del vector V_i que se está buscando, es necesario buscar la similitud más grande entre el vector Q y el vector K, para ello se usa la herramienta matemática del producto escalar entre ambos vectores, dicho producto calcula el ángulo entre ambos y dará un valor próximo a 1 si la similitud es alta y 0 si no. Para normalizar los valores se hace uso de la función Softmax la cual devuelve los valores normalizados entre los valores 0 y 1, consiguiendo así un valor de 1 o muy próximo a 1 para el valor de la Key correcta y un valor de 0 o muy cercano a 0 para el resto de las posibilidades. Una vez obtenida la probabilidad normalizada solo quedaría multiplicarla por el valor de V_i correspondiente para la K_i obtenida. La ecuación que define todo lo comentado anteriormente es la ecuación 3.1 y la figura 17 representa la distribución de los diferentes valores de la técnica Attention una vez encontrada la K_i correspondiente para un valor de Q_i analizada.

$$Attention\ (Q,K,V) = Softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{3.1}$$

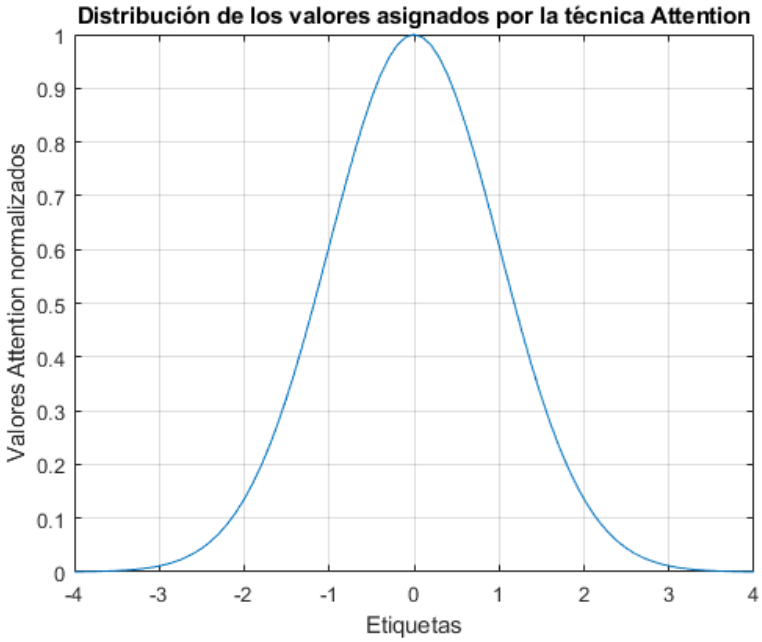


Figura 17. Distribución de los valores asignados por la técnica Attention

Cabe destacar que la ecuación 3.1 se encuentra normalizada por d_k, dicho valor representa la dimensión del vector Key, Value o Query (tabla 2). A continuación, se muestra un esquemático que muestra el funcionamiento del algoritmo descrito en los párrafos anteriores.

Para entender la notación, los valores K_N representan los valores correspondientes al vector Key. Los valores de Query son cada uno de los valores que se quieran comparar con el vector Key, para comparar la dicha similitud entre ambos se emplea la fórmula descrita anteriormente en 3.1, esta operación devuelve el valor a_N, el cual se encuentra entre los valores 0 y 1 debido a la función Softmax, definición de la operación realizada en la ecuación 3.2. Por último, estos valores a_N son multiplicados uno a uno por cada uno de los valores del vector Value, al existir uno de los valores de a_N de valor 1 o muy próximo al mismo y el resto valdrá 0 o prácticamente 0 y el valor de Attention final será el que corresponda al valor del vector Value en la posición donde se encuentra el 1. [28]

$$a_i = \frac{e^{S_i}}{\sum_j e^{S_j}} \qquad \forall i,j \leftrightarrow i \neq j \mid a_N \in [0,1] \tag{3.2}$$

Para terminar este apartado, quedaría hacer una última aclaración, la fórmula matemática formulada para calcular la similitud entre el Query y cada uno de los parámetros del vector Key (fórmula 3.1) puede ser sustituida por la expresión 3.3, en la cual se realiza una proyección de los vectores en un mismo subespacio vectorial gracias a la matriz de proyección dominada "W", posteriormente se le denominará embedding.

$$Attention\ (Q, K, V) = Softmax\ (QWK^T)V \tag{3.3}$$

Una vez realizada la explicación teórica se realizará una explicación cuantitativa de lo visto hasta el momento sobre la técnica Attention con el ejemplo propuesto anteriormente. El vector Query será cada palabra de la frase una vez esté tokenizada (se le haya realizado el embedding), es decir, en una primera iteración será la palabra "Me", en segundo lugar, será la palabra "gusta", en tercer lugar "ir", en cuarto lugar, será la palabra "de" y por último será la palabra "vacaciones". El vector de Key corresponde a la otra parte de la tabla, es decir, al embedding total que tenga la red que ha ido almacenando durante los entrenamientos como si de una BBDD se tratase, de ahí la analogía con la proyección SQL. En cada iteración cada uno de los valores de Query correspondiente es comparado con el vector Key mediante el producto escalar como se muestra en la ecuación 3.1, este producto devuelve un 1 en la posición correcto o en la que más se parezca y al ser multiplicado por el Vector Value, esto devolverá el valor de Key correspondiente a dicha posición ya que el resto será 0 o muy próximo a 0 gracias a la función Softmax.

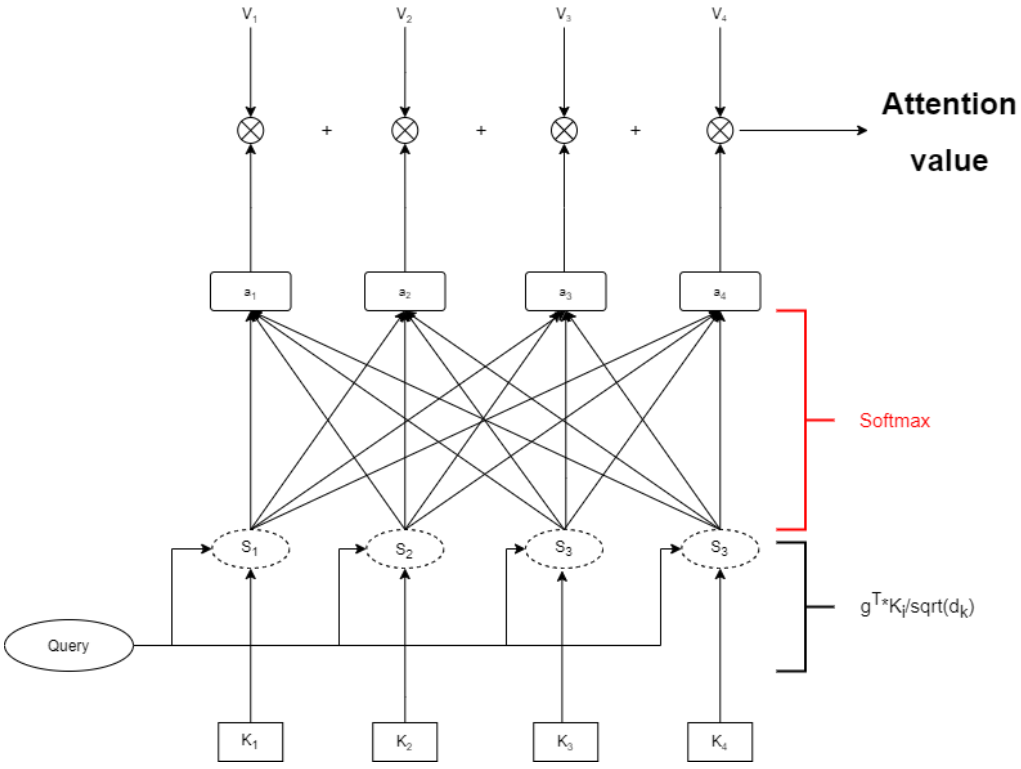


Figura 18. Esquemático de la técnica Attention

3.2 Arquitectura Transformer

A continuación, una vez visto cómo funciona la técnica Attention, se puede entrar en detalle en cómo funciona esta nueva arquitectura de Deep Learning. Como se comentó en la introducción del apartado 3 esta técnica promete suplir a las antiguas redes neuronales recurrentes para el procesado de texto (NLP).

En primer lugar, en la figura 19 se puede apreciar la arquitectura con cada parte que conforma a los Transformers. En la parte izquierda se tiene el codificador y en la derecha se tiene el descodificador, esta división en dos partes de la arquitectura hacer que sea tan eficiente. A continuación, se van a detallar las características de cada una de ellas por separado. Antes de comenzar con la descripción vamos a suponer el ejemplo en el que se quiere traducir del inglés a francés la oración: The big red dog.

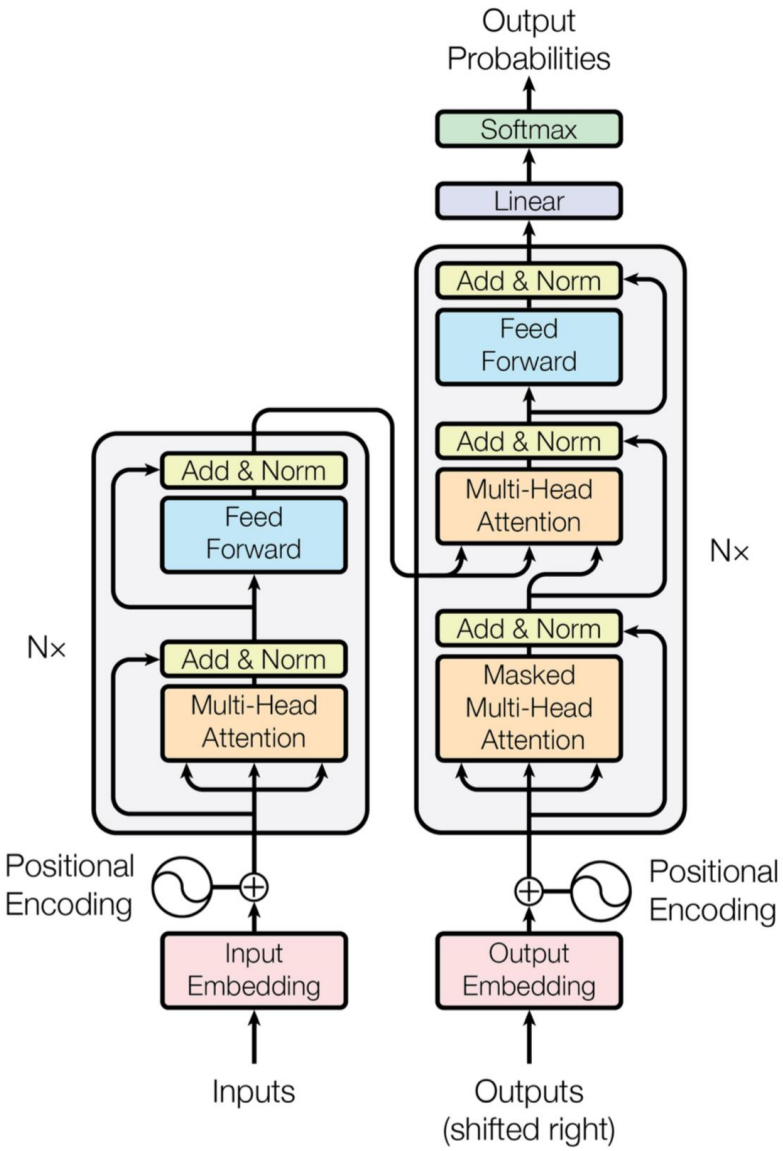


Figura 19. Arquitectura Transformer

- **Codificador**

En cuanto a la entrada se puede encontrar la primera gran diferencia con las redes convencionales, en esta ocasión la entrada es toda la oración a la vez en, al contrario que con las redes recurrentes con las cuales la entrada era palabra a palabra, de forma secuencial. Dicha entrada es convertida a "Input Embedding", es decir, se crea un espacio vectorial (un diccionario) con las palabras aprendidas por la red, las palabras que tengan un significado en común estarán más cercanas que las que no, esto permite poder mapear las diferentes palabras y poder encontrar el valor de similitud más fácilmente.

La segunda diferencia que nos encontramos es la denominada "Positional Encoding", esta herramienta permite tener un control de la posición de las palabras en todo momento. Esto se debe a que, si solo se tuviese la técnica de Attention sería como tener una bolsa con palabras (embedding) y se buscaría la más parecida, por lo que el resultado sería el mismo. Pero ¿y en el caso de que importe el orden? En ese caso al entrar toda la frase a la vez y no de forma secuencial (de palabra en palabra) es importante saber en qué posición iban cada una de las palabras antes de entrar en la capa embedding, es decir, si se busca la forma matemática, esto no es más que un vector el cual indica la distancia entre las palabras.

A continuación, el resultado de la capa anterior (una vez realizado el Encoding) se introduce en las capas Multi-Head Attention, esto no son más que múltiples capas que implementan la técnica Attention vista en el apartado 3.1. La estructura de esta capa está representada en la figura 20, donde "h" indica el número de heads deseado. Con cada una de estas capas se realiza una proyección entre el Query de entrada y el vector Key de dicha capa, esta operación devolverá el valor correspondiente que Value (fórmulas 3.5, 3.6, 3.7).

$$multihead\ (Q,K,V) = W^0 concat(head_1, head_2, \dots, head_h) \tag{3.5}$$

$$head_i = attention(W_i^Q Q, W_i^K K, W_i^V V) \tag{3.6}$$

$$attention(Q,K,V) = Softmax\ (QWK^T)V \tag{3.7}$$

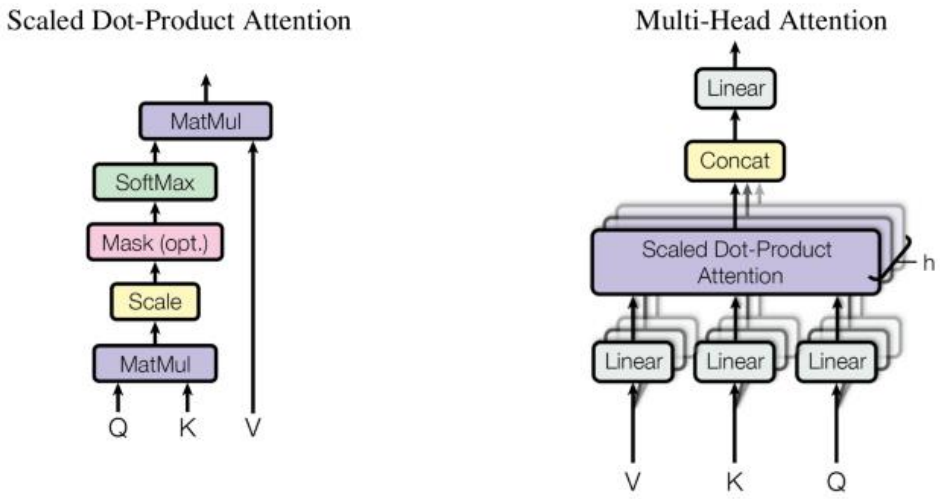


Figura 20. Parte izquierda: Representación de las fórmulas 3.5, 3.6 y 3.7
Parte derecha: Esquemático de Multi-Head Attention del codificador [29]

En concatenación con las capas Multi-Head Attention se realiza una concatenación de los valores de entrada y una normalización de los valores de los vectores Query, Key y Value obtenidos para cada uno de los tokens de la entrada. Una vez normalizados estos valores son introducidos a una red neuronal MLP, esto se hace para pasarle unos valores más sencillos al decodificador. Para esta red, es usada la función de activación ReLu y la salida vendrá dada por la ecuación 3.8 la cual indica que la salida no será más que una especie de regresión lineal, la cual dependerá de las matrices de proyección. Una representación de todo lo anterior es lo mostrado en la figura 21.

$$FFN(x) = \max(0, xW_1 + b_1)\, W_2 + b_2 \tag{3.8}$$

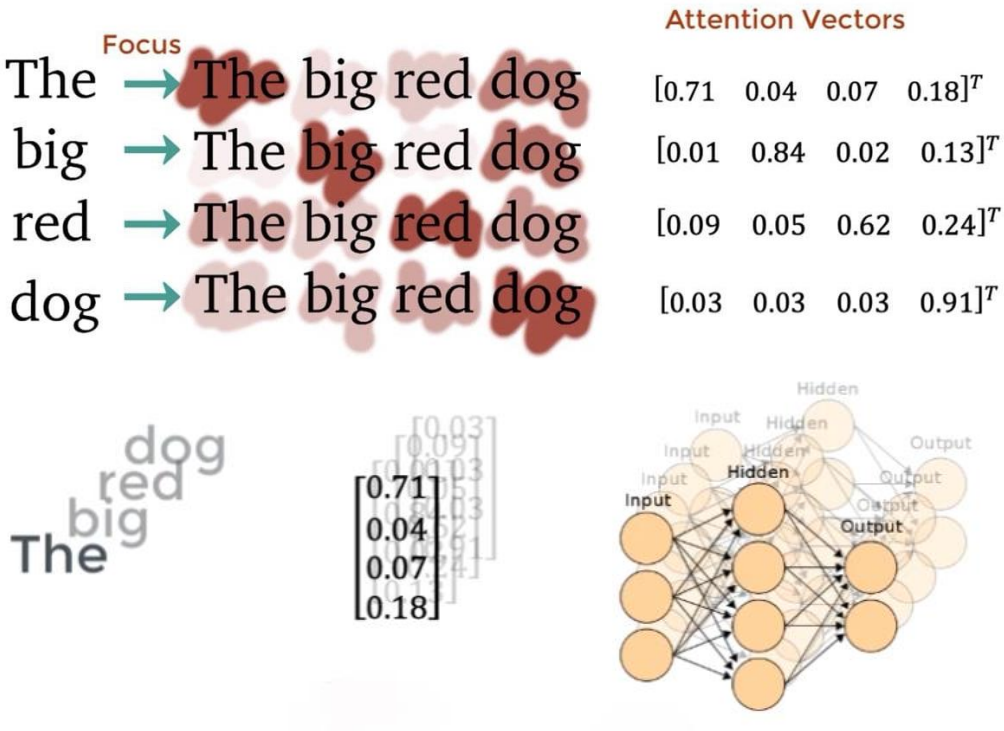


Figura 21. Ejemplo de implementación de las capas Multi-Head Attention y la MLP en el codificador

Para terminar la explicación del codificador quedaría por añadir al igual que a la salida de las capas Multi-Head Attention, una capa de normalización de los valores obtenidos y una capa full connected la cual generará los valores de Value a partir de los valores de Attention provenientes de la capa Multi-Head Attention correspondiente que a su vez, ambos valores, tanto los valores Attention como los generados por la MLP serán la entrada de una de las partes de decodificador que se verá a continuación.

- **Descodificador**

Al igual que en el codificador, se realizará el análisis de abajo hacia arriba, ya que este es el mismo proceso que seguiría la información. En primer lugar, durante el entrenamiento se tiene la entrada de información (del descodificador) que correspondería a la salida perfecta que tendría que devolver la red en un futuro. Es decir, al descodificador hay que darle como entrada la frase: Le gros chien rouge. Al contrario que en el codificador en el cual se introducía toda la información a la vez en todo momento, en esta ocasión se hace igual pero solo hasta un cierto punto en el tiempo, esto quiere

decir que en el inicio del entrenamiento si se realiza de la misma forma, pero conforme se va realizando el entrenamiento y se va creando un embedding mayor esto se cambia, en un inicio se introduce únicamente la palabra Le, a continuación, se introduce Le gros, después, Le gros chien y así sucesivamente hasta terminar, de esta forma hacemos que la red en cada iteración vaya terminando la frase con las diferentes palabras y corrigiendo los errores.

Como en el codificador cada una de estas palabras es proyectada (Output Embedding) en un subespacio vectorial para así pasar de palabras a vectores con números. Y se mantiene el positional Encoding para no perder la referencia con el codificador a la hora de relacionar palabras con la técnica Attention en la siguiente capa.

En la siguiente capa se encuentra la gran primera característica de esta arquitectura, la salida de la proyección embedding es introducida en las capas Masked Multi-Head Attention, estas capas tienen la característica de "enmascarar" la información no producida todavía, es decir, en cada una de las capas solo se tiene en cuenta la información recibida hasta ese instante temporal, poniendo a un valor de null el resto, este efecto se consigue mediante la fórmula 3.9 en la cual la matriz M incrementa la posibilidad de obtener un 0 con la función Softmax, o lo que es lo mismo, es como si se quitasen conexiones futuras. Como ya se ha comentado con anterioridad, a cada capa le entra una parte de la sentencia, por lo cual esta capa convierte en null las palabras futuras (del codificador) de la secuencia temporal por la que se encuentra. Se puede ver una representación de esta explicación en la figura 22.

Figura 22. Representación de la implementación de la capa Mask Multi-Head Attention en el decodificador

$$MaskedAttention\ (Q,K,V) = Softmax\left(\frac{QK^T + M}{\sqrt{d_k}}\right)V$$

$$M\ es\ una\ matriz\ de\ 0's\ y\ -\infty \qquad (3.9)$$

Una vez obtenidos los vectores de la técnica Attention enmascarada para cada una de las secuencias temporales se pasan a otra capa de Multi-Head Attention como la que se tenía en el codificador, esta capa posiblemente sea la más importante de toda la arquitectura y esto es debido a que es el momento en el cual se van a juntar las dos partes que conforman el Transformer. Aunque sea muy importante no deja de ser una capa con la técnica Attention y es por ello por lo que se necesitan los tres vectores característicos de la misma.

- **Vector Query:** Se corresponderá a la salida de la capa anterior Masked Multi-Head Attention.
- **Vector Key:** Será una de las dos salidas devuelta por el codificador.
- **Vector Value:** Es el otro vector que se recibe del codificador.

De forma general lo que se consigue con esta capa es ser capaz de relacionar en cada uno de los vectores las palabras que representan. Es decir, es el momento que realiza la proyección de las diferentes palabras en el mismo subespacio vectorial para ser capaz de relacionarlas a la hora de traducir. En la figura 23 se puede apreciar los vectores Query y Key recibidos, viéndose visualmente lo importante que son las capas anteriores para facilitarle y minimizar los errores de esta capa.

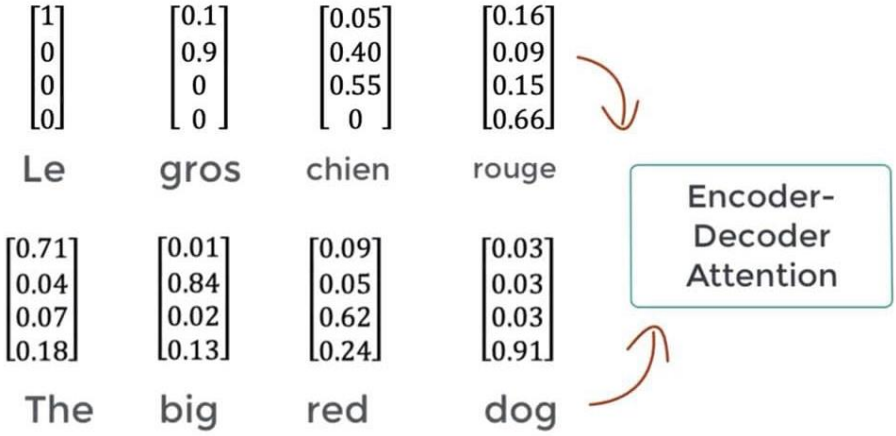


Figura 23. Representación de la implementación de la capa Multi-Head Attention en el decodificador

De forma general lo que se consigue con esta capa es ser capaz de relacionar en cada uno de los vectores las palabras de ambos idiomas. Es decir, es el momento en el cual se realiza la proyección de todas las palabras en un mismo subespacio vectorial para así ser capaz de relacionarlas a la hora de traducir. En la figura 23 se puede apreciar los vectores Query y Key recibidos, viéndose visualmente lo importante que son las capas anteriores para facilitarle el saber el punto temporal por el que se encuentra, encontrando así las estructuras sintácticas características de cada uno de los idiomas.

Una vez obtenidos los valores del vector Value de estas últimas capas de Attention se realiza de nuevo una normalización de los parámetros y se pasan a una capa en la cual se concatenan todos y cada uno de los vectores obtenidos anteriormente para su análisis mediante una MLP y son pasados por la función Softmax para una vez más tenerlos normalizados entre 0 y 1, pudiendo así distinguir fácilmente cual es el valor correcto devuelto por el Transformer. Se puede apreciar un ejemplo de esta última explicación en la figura 24. [29] [30] [31] [32]

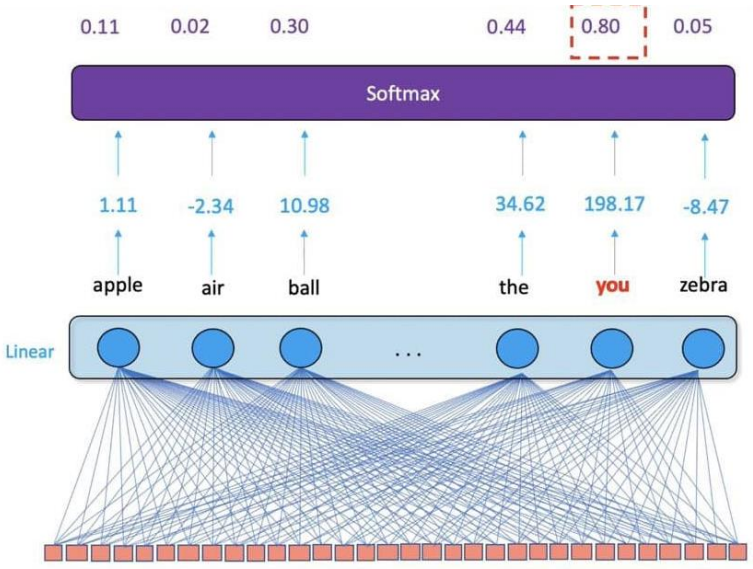


Figura 24. Representación de la salida devuelta por la arquitectura

3.3 Arquitectura Vision Transformer (ViT)

Una vez la arquitectura Transformer se había convertido en el estándar de arquitecturas a aplicar en el mundo del NPL siendo capaces de entrenar hasta 100 billones de parámetros un 400% más eficientemente que con las MLP, se comenzaron a buscar diferentes aplicaciones a otros campos del Deep Learning. Como se vio en el apartado anterior, una de las grandes características de esta nueva arquitectura era la implementación de la técnica Attention, entonces, ¿por qué no aplicarlo a reconocimiento de imágenes? Hasta este momento, las CNN habían sido las predominantes en esta rama del Deep Learning, pero como ya es sabido, esta arquitectura consume muchos recursos tanto computacionales como temporales durante el entrenamiento. En el 22 de octubre de 2020 Alexey Dosovitskiy y su equipo, presentaron el paper "An image is worth 16x16 words" [34]. En dicho paper mostraban lo que sería una evolución de la técnica Transformer con la que prometían destronar a las actuales CNN, necesitando menos recursos y obteniendo los mismos o mejores resultados que estas.

Como se puede intuir por el propio nombre del paper, para poder implementar el codificador, el cual es muy parecido al visto anteriormente (apartado 3.2), cada una de las imágenes a tratar se segmentará en N partes de 16x16, por lo tanto la estructura en esta ocasión se subdivide en dos partes un primer preprocesado de imagen y posteriormente el codificador anteriormente comentado.

- **Preprocesado de imagen**

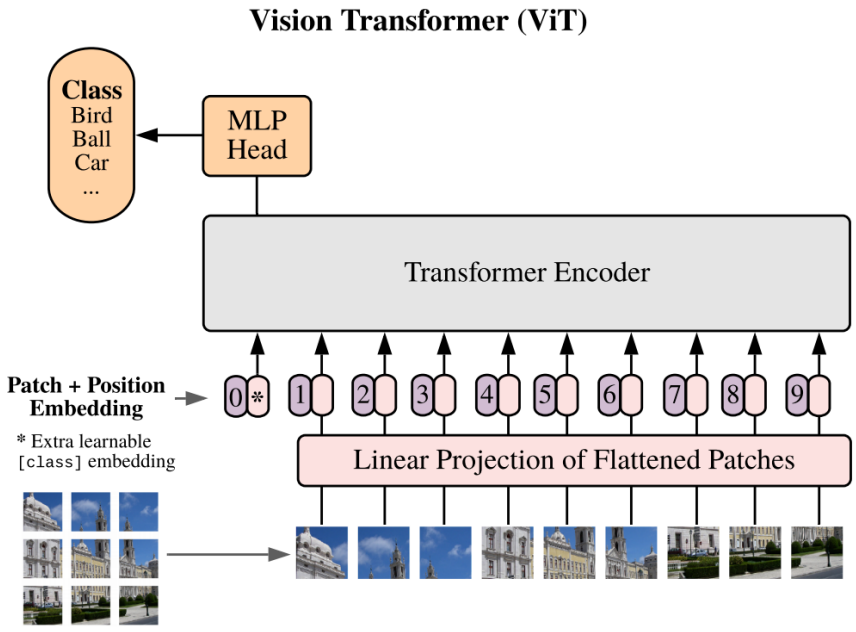


Figura 25. Preprocesado para ViT [34]

Como se ha adelantado en la introducción de este apartado, el primer paso es segmentar la imagen en 16x16 pixeles (figura 25). Esto se hace debido a que, al igual que en la anterior arquitectura, se empleará la técnica Attention, por lo cual, si la imagen es introducida como un registro de bits (valor de cada uno de los pixeles) con una codificación one-hot encoding para la tokenización de cada una de ellas, por ejemplo, la técnica calcularía la similitud con cada uno de los bits introducidos, sin tener en cuenta posición que guarda cada uno de los mismos dentro de la imagen, lo cual es posiblemente la parte más importante, además, de necesitar una dimensión de embedding del mismo tamaño que el número de clases que se tenga, ya que en este tipo de codificaciones, la distancia entre clases (vectores) ha de ser la misma y la única forma de hacer esto es haciendo que todos y cada uno de los vectores sean ortogonales a sus vecinos.

Una vez segmentada como es lógico la arquitectura solo entiende binario por lo cual, cada uno de los pixeles de cada una de las partes es pasado a vector y proyectado a un subespacio (embedding) siguiendo la ecuación 3.10. Esto es parecido a lo que se realizaba con la arquitectura Transformer en el punto anterior, al final, el resultado es tener un subespacio vectorial (diccionario) donde los patches que más se parezcan o que guarden relación entre ellos estarán más cercanos en el espacio.

$$Z_0 = \left[x_{class}; x_p^1 E; x_p^2 E; x_p^3 E; \dots; x_p^N E\right] + E_{possition}$$
$$E \in \mathbb{R}^{(P^2 \times C) \times D}, E_{possition} \in \mathbb{R}^{(N+1) \times D} \quad (3.10)$$

Donde x_p^i corresponde al i-ésimo vector de la i-ésima partición de la imagen correspondiente. E proviene de Embedding, es decir, la proyección subvectorial para proyectar cada uno de los vectores. $E_{possition}$ es la proyección de la posición a la cual corresponde cada una de las partes como ya se verá más adelante. P corresponde con los píxeles totales de los fragmentos de la imagen (resolución). C es el número de canales usados para la representación de los píxeles (igual a 3 si se usa RGB). D es la dimensión en la cual se esté trabajando, puede ser desde 1D hasta 3D.

Como se ha adelantado en el párrafo anterior, es muy importante mantener el orden de cada una de las secciones que se han sacado de la imagen ya que gracias a este orden las capas multi-head Attention son capaces de encontrar la relación entre las diferentes partes. Además, en la primera parte de la ecuación 3.10 se puede apreciar que se ha definido x_{class}, esta entrada no es más que la clase a la cual pertenece la imagen con la cual se está realizando el entrenamiento y también se le suele concatenar el número de patches creados, también se puede apreciar dicha entrada en la posición 0 en la figura 25.

- **Codificador**

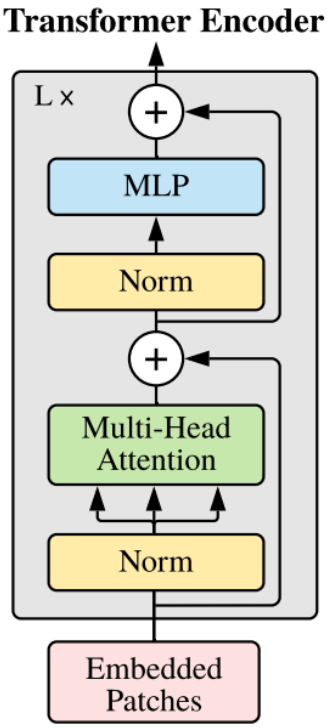


Figura 26. Codificador ViT [34]

Como se puede apreciar en la figura 26, el codificador de la arquitectura ViT guarda una cercana relación con el mostrado en la figura 19 proveniente del modelo de Vaswani planteado en 2017 para la arquitectura Transformer (apartado 3.2).

Una vez se ha realizado el embedding de cada uno de los fragmentos y de la clase a la cual pertenece la imagen completa, se pasa por una capa de normalización, para posteriormente pasar por las capas de muli-head Attention, al igual que en la arquitectura Transformer original, con estas capas se consigue establecer vectores de similitud entre cada uno de los fragmentos. Si se analiza desde el principio se puede apreciar que con este paso tenemos una de las cosas más importantes en el procesado de imágenes, es decir, por un lado se tiene la posición a la que pertenecen cada uno de los segmentos que conforman la imagen original y por otro lado, mediante la técnica Attention se están buscando los elementos de interés y estableciendo vectores de similitud entre los mismos, de esta forma, cada uno de los pixeles que conforma la imagen original tienen un sentido, estructuralmente hablando, al contrario que si se hubiera realizado con una ANN convencional que como ya se comentó anteriormente, la capa de entrada contendría cada uno de los pixeles sin importar el orden espacial.

Por último, se realiza una nueva normalización y la salida, es la entrada de una MLP que se encargará de clasificar el resultado, obteniendo así al final una categorización final. Si se piensa en una CNN convencional, se puede realizar un símil con esta última parte, al final de la CNN se obtiene una capa final con el resultado de todas las convoluciones realizadas por el kernel a lo largo de las épocas, y este resultado es introducido como entrada en una ANN convencional la cual es la encargada de realizar la clasificación al igual que pasa con esta nueva arquitectura. Todo el decodificador viene definido matemáticamente por las ecuaciones 3.11, 3.12 y 3.13, las cuales son la continuación de la ecuación 3.10 vista en el subapartado anterior [33]

Pero, si prácticamente realizan el mismo procedimiento, ¿por qué la arquitectura ViT promete ser mucho más eficiente tanto temporalmente como computacionalmente? Esto es debido a que en el ViT cada uno de los fragmentos es introducido a la vez, teniendo así tantas capas como fragmentos trabajando en paralelo. Al contrario que en las CNN en las cuales la imagen se tiene que ir recorriendo por el kernel y haciendo más pequeña para ir sacando a su vez, características más importantes. Esto, más la implementación directa de Attention sobre cada uno de los fragmentos hace que sea mucho más eficiente y ver más características conforme se avanza en las capas ViT.

$$Z'_l = MSA\big(LN(Z_{l-1})\big) + Z_{l-1} \qquad l = 1 \dots L \tag{3.11}$$

$$Z_l = MLP\big(LN(Z'_l)\big) + Z'_l \qquad l = 1 \dots L \tag{3.12}$$

$$y = LN(Z_L^0) \tag{3.13}$$

4 PROPUESTA DE TRABAJO

Una vez vistos todos los conocimientos teóricos necesarios para la resolución de este proyecto fin de grado. Se va a presentar tanto el trabajo propuesto con la BBDD (base de datos) empleada, como el material hardware y software implementado para la resolución del mismo. Además, hay que destacar que los códigos serán facilitados tanto en formato físico al final del documento (Anexos) como en formato online en Google Colaboratory donde estarán junto a los resultados post-compilación.

4.1 Trabajo propuesto

Cuando se propuso este trabajo surgieron multitud de BBDD con diferentes temáticas. Finalmente se optó por una BBDD la cual presentaba fotografías de rayos-x. Estas presentaban dos tipos de situaciones, por un lado, se tenían imágenes con estado "normal" y por otro lado se tenían imágenes en estado "neumonía". Dicha BBDD fue obtenida de la página Kaggle de la cual se propuso un concurso con dicha finalidad.

La BBDD empleada presenta la siguiente estructura:

	Train	Value	Test
Normal	1341 archivos	8 archivos	234 archivos
Neumonía	3875 archivos	8 archivos	390 archivos

Tabla 3. Estructura de la BBDD

Para el tratamiento de la BBDD se realizó la siguiente asignación, normal → '0', neumonía → '1', el código de esta parte se verá en detenimiento en el apartado 4.3.1. en la figura 28 y en el apartado 4.4.1 en la figura 36. De esta forma se conseguía bajar la computación en la parte de las etiquetas el tratamiento de las imágenes también se detallará en dichos apartados. Cómo se puede apreciar en la tabla 3, dicha base de datos no es muy extensa, ya que para el entrenamiento se tendrían solo 5216 imágenes para el entrenamiento, 8 para validación de los resultados y 624 para el test durante el entrenamiento. En un primer momento se tenía la intención de buscar una base de datos más extensa para ver el potencial de la arquitectura Transformer, pero esto no fue posible debido a los tiempos de computación que necesitaba la red convolucional para obtener una solución. Aún con esta base de datos, se podrá ver dicho efecto, aunque sea en menor escala.

Uno de los grandes retos a la hora de trabajar con esta base de datos era el poder reescalar las imágenes a un tamaño determinado sin perder la información relevante de cada una de las imágenes, ya que, para trabajar con este tipo de material, la escala de grises es muy importante y cada tono de gris cuenta positivamente para el análisis.

Al tratarse de una red relativamente pequeña se hará uso de la técnica vista en el apartado 2.3 "Data Augmentation", pudiendo así conseguir una BBDD casi tres veces más grande. Como norma general es gracias a esta técnica se puede conseguir todavía aumentar más la BBDD, pero como ya se ha comentado el trabajo con imágenes médicas es muy delicado y no se quería perder información estropeando así los resultados experimentales.

4.2 Material empleado

4.2.1 Hardware

Las técnicas de Deep Learning, incluyendo la clasificación de imágenes necesita una alta computación, puede ser proveniente de la CPU (Unidad Central de Procesamiento) o de la GPU (Unidad de Procesamiento Gráfico). En técnicas de Deep Learning como técnicas de renderizado, trazado de rayos (RTX), procesamiento tanto de video como de imágenes en tiempo real o parcial, etc., es muy común hacer uso de una tarjeta gráfica dedicada (GPU).

Para este trabajo se ha empleado un ordenador de torre con las siguientes especificaciones:

CPU	Rayzen 7 5800X
GPU	Nvidia RTX 3060
RAM	32 GB
ROM	2 TB SDD

Tabla 4. Equipo hardware empleado

4.2.2 Software

Una de las partes más importante es el software empleado para el desarrollo de este TFG. En primer lugar, si hablamos de la implementación de la CNN se han empleado las siguientes librerías específicas de Deep Learning, TensorFlow, Keras y Scikit-learn, véase la figura 27.

Figura 27. Librería da de Deep Learning implementada

De cada una de las librerías enumeradas anteriormente se importaron diferentes paquetes según se fueron necesitando durante la realización del código, para poder verlo detenidamente ir al apartado Anexo. Además de usar Numpy para la creación, operación y tratamiento tanto de matrices como de vectores, Panda para el tratamiento de los datos, PIL para el tratamiento de las imágenes. Pydot, Pydotplus y Matplotlib para la realización de gráficos y análisis visual de los datos que se mostrará en el apartado 5.1 y en el apartado 5.2.

Todas las librerías y paquetes nombrados anteriormente fueron instalados en un entorno en la aplicación Anaconda con Python 3.8.8, lo que ha permitido trabajar con las últimas versiones de todas las librerías y paquetes. El sistema operativo empleado en esta ocasión ha sido Windows 10.

4.3 Red Convolucional

4.3.1 Tratamiento de los datos

Para el tratamiento de la BBDD se realizó la siguiente asignación: normal → '0', neumonía → '1' (figura 28). De esta forma se conseguía bajar la computación en la parte de las etiquetas.

```
y_train = np.concatenate(((np.zeros(len(train_data_normal))),(np.ones(len(train_data_neumonia)))))
y_val = np.concatenate(((np.zeros(len(val_normal))),np.ones(len(val_neumonia))))
y_test = np.concatenate(((np.zeros(len(test_data_normal))),np.ones(len(test_data_neumonia))))
```

Figura 28. Codificación de las etiquetas

Por último, para el tratamiento de la información se usó el paquete ImageDataGenerator importado desde la librería Keras, véase el apartado 4.2.2 para ver las librearías usadas y el Anexo para verlas en el propio código. Como se acaba de comentar, el tratamiento de la información se realizó con la clase ImageDataGenerator y se le realizó el preprocesamiento que se aprecia en la figura 29, dicho tratamiento de los datos es la solución óptima, la evolución temporal que se ha seguido se puede ver en el apartado 5.1.

```
image_train = train_gen.flow_from_directory(
    train_data,
    target_size = (altura, long),
    batch_size = batch_size,
    color_mode = "rgb",
    class_mode='categorical')

image_val = val_gen.flow_from_directory(
    val_data,
    target_size=(altura, long),
    batch_size=batch_size,
    color_mode = "rgb",
    class_mode='categorical')

image_test = val_gen.flow_from_directory(
    test_data,
    target_size=(altura, long),
    batch_size=batch_size,
    color_mode = "rgb",
    class_mode='categorical')
```

Figura 29. Implementación de la clase ImageDataGenerator

Como se puede apreciar, a las imágenes de entrenamiento se les realizó cuatro tipos de procesados de imagen, en primer lugar, una normalización de los pixeles entre 0 y 1, una inclinación del 30%, un zoom relativo del 30% y se permitió el giro horizontal (apartado 2.3). Con esto se consigue que la red neuronal no aprenda de forma lineal, si no que sea capaz de aprender cuando las condiciones de las imágenes no sean las más apropiadas, además de aumentar la cantidad de datos de la BBDD. Tanto para los datos de validación como para los datos de test solo se le aplicó la normalización de los pixeles para asegurarse de que la red está aprendiendo de forma correcta.

A continuación, en la figura 30 se aprecia el siguiente paso en el tratamiento de las imágenes, a todas las imágenes, independientemente del dataset de procedencia, se les aplica una normalización tanto en altura como en anchura (130 x 130), se seleccionó un batch size (número de imágenes cogidas por época) de 64 imágenes y se le asignó una clasificación categórica ya que solo existen dos tipos de clases.

```
train_gen = ImageDataGenerator(
    rescale=1./255,
    shear_range=0.3,
    zoom_range=0.3,
    horizontal_flip=True)

val_gen = ImageDataGenerator(
    rescale=1./255)


test_gen = ImageDataGenerator(
    rescale=1./255)
```

Figura 30. Implementación del paquete ImageDataGenerator

4.3.2 Creación de la red

Para el primer modelo a implementar se implementaron las todas las librerías vistas en el apartado 4.2.2 y tanto en la figura 31 como en el apartado Anexo se pueden ver las implementaciones. Hay que destacar que los valores mostrados en ese subapartado son los óptimos y toda la evolución de cada uno de los parámetros se detalla en el apartado 5.1

```
#Imports
import os
import sys
import numpy as np
import pandas as pd
import keras
import tensorflow as tf
import pydot
import pydotplus
import sklearn
import matplotlib.pyplot as plt
from keras import layers
from keras import models

from keras.utils.vis_utils import plot_model
from keras.utils.vis_utils import plot_model
from keras.utils.vis_utils import model_to_dot
from keras.preprocessing.image import load_img
from keras.layers import Dropout, Flatten, Dense, Activation
from keras.preprocessing.image import ImageDataGenerator
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import normalize
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from tqdm import tqdm
from PIL import Image
```

Figura 31. Imports necesarios para la CNN

Como se vio anteriormente en la parte teórica (apartado 3.1) las CNN están formadas por dos partes principalmente:

1. **Parte convolucional**, donde se ha implementado un Kernel de 3x3 por ser esta dimensión óptima recomendada por los profesionales en este campo del Deep Learning [12]. Como se puede ver en la figura 32 se implementaron 3 capas, siendo cada una de ellas más ancha que la anterior, consiguiendo cada una de ellas más características que en la anterior. Además de capas "MaxPooling" entre cada una de las capas anteriores, esta capa permite reducir el tamaño de la matriz de cada una de las salidas (al tamaño indicado) y por lo tanto la computación de la capa siguiente, para conseguir este fin se divide la matriz de salida en tantos cuadrantes como se le indique y únicamente selecciona el valor máximo que por norma general será el que está relacionado con el hallazgo de un patrón.

```
CNN = models.Sequential()
CNN.add(layers.Conv2D(32,kernel_size=(3,3), padding = 'same',input_shape = (altura, long,3),
activation='relu'))
CNN.add(layers.MaxPooling2D(pool_size=(2,2)))
CNN.add(layers.Conv2D(64,kernel_size=(3), padding='same', activation='relu'))
CNN.add(layers.MaxPooling2D(pool_size=(2,2)))
CNN.add(layers.Conv2D(128, kernel_size=(3,3), padding='same', activation='relu'))
CNN.add(layers.MaxPooling2D(pool_size=(2,2)))
```

Figura 32. Creación de la parte convolucional

Cabe destacar varios aspectos más que se han seleccionado para este modelo, el primero de ellos es que se tiene el argumento input_shape, este argumento solo se indica en la primera capa ya que sirve para decirle a la red la altura como el ancho de las imágenes que va a recibir, es importante recordar que en el preprocesado se reescalaron todas las imágenes a esa dimensión, a continuación se le indica el número de canales (RGB), la primera idea sería poner el número de canales igual a dos ya que son imágenes en blanco y negro, pero esto es incorrecto debido a que el toque de los grises es muy importante y puede ser determinante en la detección de enfermedades en imágenes de este tipo, como se ha comentado en el apartado anterior, cuando se intentó implementar dicho cambio la red mostraba un error por la terminal y por lo tanto se volvió a los tres canales RGB. Por último, se indica la función de activación, la cual en este caso es la ReLu, pero ¿por qué ReLu y no otras? Cómo ya vimos existen funciones mejores que esta, como son la Leky-ReLu, ELU, etc. Pero en el caso de las CNN conviene anular los valores negativos y las características encontradas han de ser pasadas a la siguiente capa tal y como se detecten.

2. **Red neuronal densa**, esta red no es más que una ANN común, la cual se encarga de recibir la salida de la última capa de la parte convolucional, habiendo realizado un flatten (aplanamiento de la salida de la parte convolucional), y realizando una clasificación normal y corriente. En este caso se crearon 4 capas con 128, 64, 32 y 2 neuronas respectivamente. La más importante es la última de las capas, ya que esta tiene que tener el mismo número de neuronas que categorías ha de clasificar. A cada una de las capas se le aplicó la técnica denominada dropout, esta técnica tiene la finalidad de desactivar un porcentaje de neuronas (80%, 60% y 40% respectivamente) aleatorias en cada una de las iteraciones, de esta manera se consigue que la red no aprenda a memorizar caminos para realizar la predicción ya que esto empeoraría el resultado final, a este fenómeno se le denomina overfitting. En cuanto a la función de activación se mantuvo la implementación de la ReLu en las dos primeras capas y en la última se tomó la Softmax para normalizar el resultado final entre 0 y 1, eligiendo así cual es el resultado final correcto. Todo lo comentado en este párrafo se puede ver en la figura 33.

```
#Se aplana para pasarsela a la ANN
CNN.add(layers.Flatten())
#Creacion de la red densa
CNN.add(Dense(128,activation = 'relu'))
CNN.add(layers.Dropout(.8))
#el 80% de las neuronas son apagadas aleatoriamente
CNN.add(Dense(64,activation = 'relu'))
CNN.add(layers.Dropout(.6)) # 60%
CNN.add(Dense(32, activation='relu'))
CNN.add(layers.Dropout(.4)) #40%
CNN.add(Dense(2, activation='Softmax'))
```

Figura 33. Creación de la red densa

Para alcanzar el óptimo se seleccionaron un total de 75 épocas y dentro de cada época se seleccionaron un número en concreto de pasos, este número vino determinado por la línea "(len(x_train)//batch_size)//3", donde con "len(x_train)" se consigue saber el número de imágenes con el cual se va a entrenar, este es dividido entre el batch_size para saber con certeza así que en todas las etapas se van a tener el mismo número de imágenes para entrenar, por último se dividió este número entre 3 para conseguir bajar así la computación en cada una de las épocas y 66% (figura 34). Por último, para optimizar la red, se le pasó como parámetro de perdidas categorical_crossentropy ya que se tiene dos clases únicamente, como optimizador se usó Adam, ya que este es el optimizador por excelencia haciendo uso de la técnica del descenso del gradiente para alcanzar el valor óptimo y la métrica para saber cómo de bien va nuestra red fue accuracy definida en la ecuación 4.1 y la implementación puede verse en la figura 35.

```
CNN.fit(image_train,
        steps_per_epoch= ((len(x_train)//batch_size)//3),
        epochs = epochs,
        validation_data=image_val,
        validation_steps= ((len(x_test)//batch_size))//3)
```

Figura 34. Parámetros para el entrenamiento

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

Figura 35. Parámetros de optimización

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} \quad (4.1)$$

En cuanto a la notación que proporciona el resultado de accuracy definida anteriormente en la ecuación 4.1 y suponiendo un caso binario de 1-0 para el ejemplo, destacar que TP hace referencia a "True Negative" es decir el número de veces que la red ha predicho un 0 y se ha equivocado; TP hace referencia a "True Positive" es decir el número de veces que la red ha predicho un 0 y ha sido correcta dicha predicción; FP quiere decir "False Positive" que indica el número de veces que la red ha predicho un 1 y ha acertado; FN hace referencia a "False Negative" que es el número de veces que la red ha predicho un 1 y dicha predicción era incorrecta.

4.4 Arquitectura ViT

Para la realización de este apartado se ha usado el modelo ViT creado por Khalid Salama [38] como esqueleto. Tanto el tratamiento de los datos como la evaluación de los resultados fueron realizados de forma exclusiva para este trabajo y el modelo ViT fue adaptado para el mismo.

4.4.1 Tratamiento de los datos

El tratamiento de los datos que se realizó en esta ocasión fue un tanto distinto respecto al que se realizó en la anterior ocasión para la CNN (apartado 4.3.1). Para este apartado no se hizo uso de la clase "ImageDataGenerator" por incompatibilidad con el resto de las clases que formarán la estructura, los cuales serán visto en detalle en el siguiente apartado (apartado 4.4.2).

En un primer lugar, se creó la función give_set (figura 36) con la cual se extraerán las imágenes de cada una de las carpetas, extrayendo también sus etiquetas correspondientes (normal y neumonía), y guardándolas en las variables correspondientes ya se corresponda con las imágenes (x_carpeta) o con las etiquetas (y_carpeta), realizándose el mismo proceso que para la CNN, se asignó la etiqueta 1 si era normal y la etiqueta 0 si presentaba neumonía. Para el preprocesamiento de las imágenes si se le han dado las mismas características para estar en igualdad respecto a la red anterior, se han habilitado los tres canales RGB y se reescalaron a un tamaño de 130 x 130. Todo ello se puede ver en detalle en la figura que se muestra a continuación.

```
def give_set (path) :
    x = []
    y = []
    for etiqueta in os.listdir(path) :
        for img in os.listdir(path +'/' + etiqueta + '/') :
            img_path = path +'/'+ etiqueta + '/' + img
            image = keras.preprocessing.image.load_img(img_path,
                                color_mode = 'rgb', target_size = (130,130))
            x.append(img_to_array(image))
            y.append(1 if etiqueta == 'PNEUMONIA' else 0)

    x = np.array(x, dtype = float)
    y = np.array(y)


    return shuffle(x,y)

x_train, y_train = give_set(train_data)
x_test, y_test = give_set(test_data)
x_value, y_value = give_set(val_data)
```

Figura 36. Creación de las variables de train, test y value

A continuación, se hizo uso de "Data Augmentation" para conseguir una base de datos más extensa de la que ya se tenía como se vio en el apartado 2.3. Primero se normalizan los datos y posteriormente se realiza el reescalado de las imágenes al tamaño comentado anteriormente (130 x 130), posteriormente se habilita el giro horizontal, una rotación de un 2% y un zoom digital de un 20% tanto en altura como en anchura, si se compara con los valores aplicados en la CNN se puede ver que no son los mismos, esto se debe a que son los parámetros con los cuales mejores resultados se obtuvieron con esta red. Todo esto es aplicado a la variable x_train que es la que contiene las imágenes de entrenamiento. La implementación de todo esto se puede ver en la figura 37.

```
data_augmentation = keras.Sequential(
    [
        layers.experimental.preprocessing.Normalization(),
        layers.experimental.preprocessing.Resizing(image_size, image_size),
        layers.experimental.preprocessing.RandomFlip("horizontal"),
        layers.experimental.preprocessing.RandomRotation(factor=0.02),
        layers.experimental.preprocessing.RandomZoom(
            height_factor=0.2, width_factor=0.2
        ),
    ],
    name="data_augmentation",
)
data_augmentation.layers[0].adapt(x_train)
```

Figura 37. Implementación de "Data augmentation" manualmente a x_train

Por último, como se vio en el apartado 3.3 en el cual se explicó en detalle el funcionamiento de la arquitectura ViT, cada una de las imágenes ha de ser segmentada en particiones de 16x16 para posteriormente ser analizados por la red. En la figura 38, se puede apreciar el código de este paso ya que hace uso de diferentes funciones. En primer lugar, hace uso de la clase "Patches" la cual se encargará de devolver las particiones de cada imagen, si se hace uso del índice [1] devuelve el número de arrays que forman la imagen total, es decir, el número de particiones y si se hace uso del índice [-1] indica el número de elementos que hay dentro de cada array, devolviendo así el número de pixeles que forma cada imagen. Las particiones serán de 16x16 como así se indicó, este valor viene dado en la variable "patch_size" que se configura al inicio del código.

```
class Patches(layers.Layer):
    def __init__(self, patch_size):
        super(Patches, self).__init__()
        self.patch_size = patch_size

    def call(self, images):
        batch_size = tf.shape(images)[0]
        patches = tf.image.extract_patches(
            images=images,
            sizes=[1, self.patch_size, self.patch_size, 1],
            strides=[1, self.patch_size, self.patch_size, 1],
            rates=[1, 1, 1, 1],
            padding="VALID",
        )
        patch_dims = patches.shape[-1]
        patches = tf.reshape(patches, [batch_size, -1, patch_dims])
        return patches


"""
Let's display patches for a sample image
"""

plt.figure(figsize=(4, 4))
image = x_train[np.random.choice(range(x_train.shape[0]))]
plt.imshow(image.astype("uint8"))
plt.axis("off")

resized_image = tf.image.resize(
    tf.convert_to_tensor([image]), size=(image_size, image_size)
)
patches = Patches(patch_size)(resized_image)
print(f"Image size: {image_size} X {image_size}")
print(f"Patch size: {patch_size} X {patch_size}")
print(f"Patches per image: {patches.shape[1]}")
print(f"Elements per patch: {patches.shape[-1]}")

n = int(np.sqrt(patches.shape[1]))
plt.figure(figsize=(4, 4))
for i, patch in enumerate(patches[0]):
    ax = plt.subplot(n, n, i + 1)
    patch_img = tf.reshape(patch, (patch_size, patch_size, 3))
    plt.imshow(patch_img.numpy().astype("uint8"))
    plt.axis("off")
```

Figura 38. Creación de la segmentación de 16x16

Para mostrar cómo se vería estas particiones en una imagen cualquiera, se realizó un ejemplo de este código con una imagen random devuelta de “x_train”. Véase la figura 39.

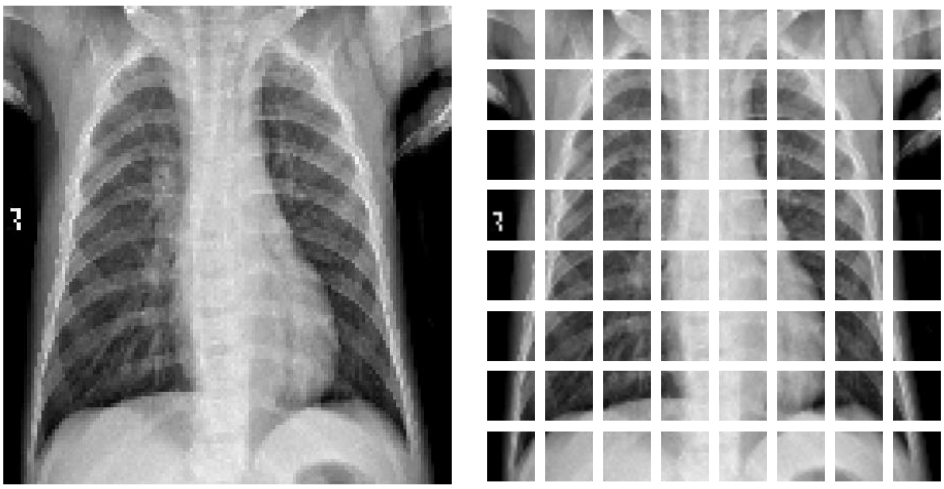


Figura 39. Ejemplo de la partición de 16x16

4.4.2 Creación del modelo

En este subapartado se va a implementar el desarrollo teórico que se desarrolló en el apartado 3.3 y se representó en la figura 25. Para ello, se va a comenzar por la parte izquierda de dicha figura, es decir, por crear los segmentos (patches) y el embedding de la posición. Antes de entrar con la arquitectura, es necesario crear la codificación de los segmentos y el embedding de la posición, con esto se consigue la entrada del denominado codificador. Para conseguir esta codificación se hace uso de una clase llamada “PatchEncoder”, al ser llamada esta clase proyecta cada uno de los segmentos y crea el embedding de la posición que ocupa del segmento en cuestión.

```
class PatchEncoder(layers.Layer):
    def __init__(self, num_patches, projection_dim):
        super(PatchEncoder, self).__init__()
        self.num_patches = num_patches
        self.projection = layers.Dense(units=projection_dim)
        self.position_embedding = layers.Embedding(
            input_dim=num_patches, output_dim=projection_dim
        )

    def call(self, patch):
        positions = tf.range(start=0, limit=self.num_patches, delta=1)
        encoded = self.projection(patch) + self.position_embedding(positions)
        return encoded
```

Figura 40. Patch + Position Embedding

Una vez creadas las clases y funciones características de este modelo, se puede empezar con la arquitectura Transformer en cuestión. Para los códigos que se presentarán a continuación es recomendable recordar la figura 26 ya que será el modelo por seguir a partir de este momento. La estructura ViT fue definida en la función “create_vit_classifier”, en dicha función en primer lugar se llama a las clases y funciones vistas anteriormente data_augmentation, Patches y PatchEncoder, respectivamente. Si se mira la figura 26, hasta ahora se ha conseguido la entrada del codificador Transformer. El número de capas que tendrá el codificador (“L” en la figura 26) viene determinado por la variable “transformer_layer”.

Siguiendo el esquema, el siguiente paso es realizar una normalización de la salida devuelta por las clases y funciones anteriores, esta normalización es denominada “x1”, importante recalcar que esta normalización se ha realizado con una épsilon=1e-6, este valor se le suma a la varianza para evitar así que al dividir se divida entre cero. Después, el resultado es la entrada de las capas Multi-Head Attention, el número de capas vendrá delimitado por “num_heads” (ver figura 20), la dimensión de esta capa ha de coincidir con la entrada, es decir, ha de tener el mismo tamaño que la proyección realizada en el apartado anterior y se aplica un dropout de un 10% para evitar overfitting, estas capas son denominadas attention_output. En el código “x2” únicamente representa la suma de la salida del párrafo anterior con estas Multi-Head Attention siguiendo la realimentación que se muestra en la figura. La salida de Multi-Head Attention es normalizada al igual que “x1” y en este caso es denominada “x3”, esta normalización es analizada por una MLP que tendrá el tamaño que se le indique mediante la variable “transformer_units” y con un dropout de un 10% de nuevo. Por último, hay volver a sumar estas dos últimas capas siguiendo de nuevo la realimentación que se indica. Toda la explicación detallada en este párrafo es la implementación de la figura 41.

Para terminar, quedaría “representar” la salida de alguna forma, esto en el paper original no se especifica como tal, por lo cual cada autor es libre de hacer lo que crea mejor, en este caso el creador de este modelo creó la siguiente estructura:

1. Normalización de la salida
2. Aplanamiento de los datos a un vector de 1D.
3. El resultado es analizado por una red MLP la cual tendrá tantas capas como se le indique en la variable “mlp_head_units”, esta red es la que se encargará de realiza la decisión final mediante una función Softmax, que en nuestro caso es perfecta ya que solo tenemos dos clases.
4. Creación el objeto “model” que será el devuelto por la función en cuestión y donde se guardará el modelo final.

Todo lo descrito anteriormente es la implementación mostrada en la figura 41.

```
def create_vit_classifier():
    inputs = layers.Input(shape= (130, 130, 3))
    # Augment data.
    augmented = data_augmentation(inputs)
    # Create patches.
    patches = Patches(patch_size)(augmented)
    # Encode patches.
    encoded_patches = PatchEncoder(num_patches, projection_dim)(patches)

    # Create multiple layers of the Transformer block.
    for _ in range(transformer_layers):
        # Layer normalization 1.
        x1 = layers.LayerNormalization(epsilon=1e-6)(encoded_patches)
        # Create a multi-head attention layer.
        attention_output = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=projection_dim, dropout=0.1
        )(x1, x1)
        # Skip connection 1.
        x2 = layers.Add()([attention_output, encoded_patches])
        # Layer normalization 2.
        x3 = layers.LayerNormalization(epsilon=1e-6)(x2)
        # MLP.
        x3 = mlp(x3, hidden_units=transformer_units, dropout_rate=0.1)
        # Skip connection 2.
        encoded_patches = layers.Add()([x3, x2])

    # Create a [batch_size, projection_dim] tensor.
    representation = layers.LayerNormalization(epsilon=1e-6)(encoded_patches)
    representation = layers.Flatten()(representation)
    representation = layers.Dropout(0.5)(representation)
    # Add MLP.
    features = mlp(representation, hidden_units=mlp_head_units, dropout_rate=0.5)
    # Classify outputs.
    logits = layers.Dense(num_classes)(features)
    # Create the Keras model.
    model = keras.Model(inputs=inputs, outputs=logits)
    return model
```

Figura 41. Implementación del codificador ViT

Una vez creado el modelo, solo queda entrenarlo y para este fin se creó la función "run_experiment", en dicha función se realizan todos los ajustes necesarios para la compilación, entrenamiento y monitorización. El optimizador elegido es AdamW, este optimizador es una subversión del optimizador Adam y puede recibir dos parámetros:

1. **Learning rate:** Este parámetro indica cada cuanto son actualizados los pesos de nuestra arquitectura. Para esta ocasión fue seleccionado un valor muy bajo (0.0005) ya que el parecido entre todas las fotos que conforman la base de datos es muy grande.
2. **Weight decay:** Este parámetro se encarga de penalizar los cambios grandes en los pesos con un determinado valor (menor que uno) si estos superan el umbral. Con esto se consigue el entrenamiento sea más lineal y no tendrá grandes picos durante el entrenamiento evitando entre otras cosas el overfitting. El valor usado fue 0.001.

Gracias a la manipulación de estos dos parámetros se consiguen unos resultados más precisos. Para la función de perdidas (loss) se implementó "SparseCategoricalCrossentropy" que gracias a tener solo dos clases las cuales son representadas mediante números enteros (0-1) es la más eficiente. Y para la monitorización se implementó el accuracy, como no podía ser de otra forma mediante "SparseCategoricalAccuracy".

La diferencia entre "CategoricalCrossentropy" y "SparseCategoricalCrossentropy" o "CategoricalAccuracy" y "SparseCategoricalAccuracy", es la versión de TensorFlow con la que se trabaje, sin embargo, ambas son válidas y ambas se pueden usar, la gran diferencia entre ambas funciones (dando igual la pareja que se elija), es que si no presenta el prefijo "Sparse" la función compara si el índice del valor real máximo y el máximo predicho son iguales y por el contrario si presenta el prefijo "Sparse" la función comprueba si el valor real máximo es igual al índice del valor máximo predicho.

A continuación, en la figura 42 se representa el código tanto de la compilación como del entrenamiento y de la monitorización durante el entrenamiento de la arquitectura ViT.

```
def run_experiment(model):
    optimizer = tfa.optimizers.AdamW(
        learning_rate=learning_rate, weight_decay=weight_decay
    )

    model.compile(
        optimizer=optimizer,
        loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
        metrics=[
            keras.metrics.SparseCategoricalAccuracy(name="accuracy"),
            keras.metrics.SparseTopKCategoricalAccuracy(5, name="top-5-accuracy")])

    checkpoint_filepath = "/tmp/checkpoint"
    checkpoint_callback = keras.callbacks.ModelCheckpoint(
        checkpoint_filepath,
        monitor="val_accuracy",
        save_best_only=True,
        save_weights_only=True,
    )

    history = model.fit(
        x=x_train,
        y=y_train,
        batch_size=batch_size,
        epochs=num_epochs,
        validation_data= (x_value, y_value),
        callbacks=[checkpoint_callback],
    )

    model.load_weights(checkpoint_filepath)
    _, accuracy, top_5_accuracy = model.evaluate(x_test, y_test)
    print(f"Test accuracy: {round(accuracy * 100, 2)}%")
    print(f"Test top 5 accuracy: {round(top_5_accuracy * 100, 2)}%")

    return history

vit_classifier = create_vit_classifier()
history = run_experiment(vit_classifier)
```

Figura 42. Compilación, entrenamiento y monitorización de ViT.

5 RESULTADOS OBTENIDOS

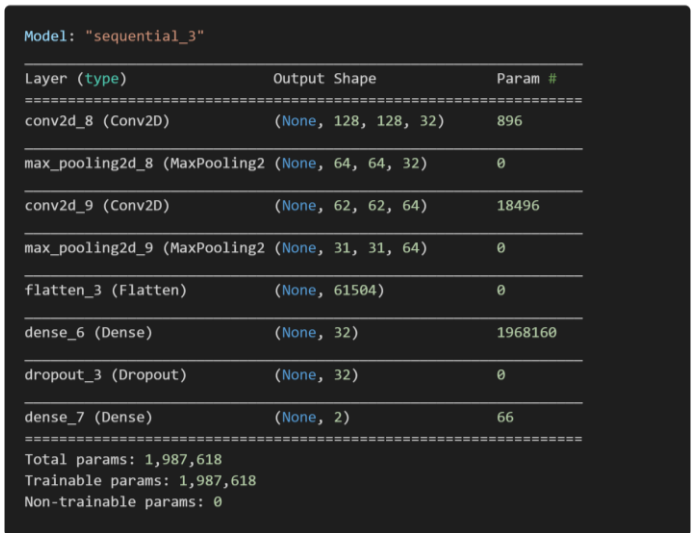
Como ya se comentó en el apartado 1, para la resolución de este problema se han propuesto dos modelos y en el apartado 4.3 y 4.4 se ha detallado su implementación en Python, en primer lugar se detalló la implementación de la CNN y en segundo lugar se implementó la arquitectura ViT. A continuación, se detallará tanto la evolución temporal que se ha seguido con cada una de las arquitecturas como los resultados finales obtenidos con cada uno de los modelos que se han ido creado para cada una de las arquitecturas y al final del apartado se hará una breve comparación de las dos arquitecturas, aunque su discusión se verá en detalle en el apartado 6.

5.1 Red Convolucional

En el aparado anterior se ha podido ver el código de la estructura final de esta red, pero como es obvio este resultado no fue el primero con el cual se probó si no que a lo largo de los entrenamientos conforme se iban analizando los resultados y se iban realizando los cambios pertinentes para intentar solventarlos. En primer lugar, no se sabía la dificultad que tenían las imágenes médicas en cuanto al tratamiento de los datos que como se verá a continuación es muy delicado ni tampoco la dificultad que tendría la red en un futuro para entrenar ya que al ser imágenes en escala de grises se tenía una concepción errónea.

5.1.1 Primer modelo

Para el primer tratamiento de los datos, no se realizó la técnica “Data Augmentation” que se explicó en el apartado 2.3 por simple desconocimiento de la existencia de la misma y la red que se creó fue la mostrada en la figura 43, que como se puede apreciar es demasiado pequeña ya que como se ha comentado en la primera parte del apartado el primer pensamiento fue que el análisis de este tipo de imágenes era muy sencillo.



```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_8 (Conv2D)            (None, 128, 128, 32)      896
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 64, 64, 32)        0
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 62, 62, 64)        18496
_________________________________________________________________
max_pooling2d_9 (MaxPooling2 (None, 31, 31, 64)        0
_________________________________________________________________
flatten_3 (Flatten)          (None, 61504)             0
_________________________________________________________________
dense_6 (Dense)              (None, 32)                1968160
_________________________________________________________________
dropout_3 (Dropout)          (None, 32)                0
_________________________________________________________________
dense_7 (Dense)              (None, 2)                 66
=================================================================
Total params: 1,987,618
Trainable params: 1,987,618
Non-trainable params: 0
```

Figura 43. Estructura de la primera CNN creada

Como se puede apreciar la red presenta únicamente seis capas en total, cuatro capas para la parte convolucional (dos capas convolucionales y dos capas Max Pooling) y únicamente dos capas para la parte densa de la CNN de 32 y 2 neuronas. Como ya se comentó en el apartado 4.3.2 en la parte densa de la red se especificaron dos capas “Max Pooling” para reducir el tamaño de las salidas

y a su vez cada una de las capas es más gruesa que la anterior para así poder detectar más características, destacar que esta filosofía se ha seguido hasta el final ya que es lo recomendable en todos los casos. Además, en la red densa se añadió la técnica de Dropout, con esta técnica lo que se pretende es evitar el famoso "overfitting" ya que su función es desactivar un determinado número aleatorio de neuronas en cada época que se realicen durante el entrenamiento, de esta forma cada neurona no aprende de forma lineal sabiendo que solo necesita aprender unas determinadas características ya que las neuronas adyacentes aprenden otras, si no, que tienen que hacer todo el trabajo por ellas mismas ya que no tienen el apoyo de sus neuronas vecinas, el dropout aplicado fue de un 40%. En la figura 44 se pueden apreciar los resultados obtenidos no son muy positivos sobre todo en la parte de "Validation" tanto en Accuracy como en Loss en las cuales se puede ver que el entrenamiento no es positivo. Esto puede deberte a diversos factores, pero los más comunes son "underfitting" u "overfitting", que el entrenamiento no se estaba realizando de forma correcta por la estructura de la red o que la base de datos era insuficiente (demasiado pequeña).

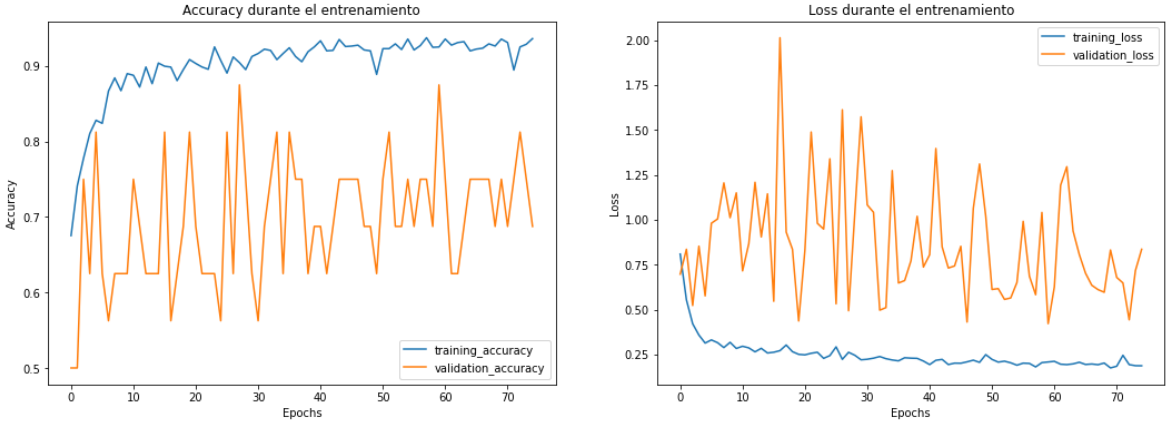


Figura 44. Valores de Accuracy y Loss del 1er modelo CNN durante el entrenamiento

5.1.2 Segundo modelo

Después de buscar información en diferentes fuentes sobre el tratamiento necesario para imágenes médicas se descubrió la gran dificultad que estas presentaban. Esto es debido a la escala de grises, aunque parezca un dato que no es muy importante a primera vista, en las imágenes de rayos-x una de las cosas más importantes son la escala de grises y el tamaño de estas, por lo tanto, habría que tener especial cuidado en el redimensionamiento de las imágenes y en el tratamiento que se les da a los pixeles para que estos no pierdan la información que tienen. En primer lugar, se hizo uso de la técnica "Data Augmentation" para poder así hacer la BBDD mayor para ello se realizó un zoom relativo y una inclinación de un 30% además de permitir el giro de 180º en todas las direcciones. En segundo lugar, se realizó una normalización de los pixeles dejando los valores entre [0,1] con el comando "rescale = 1./255". En tercer lugar, se optó por usar dos canales en vez de tres (RGB), pero instantáneamente la red descubría un error ya que necesitaba los tres canales para poder representar todas las tonalidades de grises que presentaban las imágenes. Por último, se hizo más grande la parte convolucional de la CNN añadiendo una última capa más de un ancho de 128 como se puede apreciar en la figura 45.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_3 (Conv2D)            (None, 128, 128, 32)      896
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 64, 64, 32)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 62, 62, 64)        18496
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 31, 31, 64)        0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 29, 29, 128)       73856
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 14, 14, 128)       0
_________________________________________________________________
flatten_1 (Flatten)          (None, 25088)             0
_________________________________________________________________
dense_2 (Dense)              (None, 32)                802848
_________________________________________________________________
dropout_1 (Dropout)          (None, 32)                0
_________________________________________________________________
dense_3 (Dense)              (None, 2)                 66
=================================================================
Total params: 896,162
Trainable params: 896,162
Non-trainable params: 0
```

Figura 45. Estructura de la segunda CNN creada

En la figura 46 se pueden ver los resultados que devolvió la red después de los cambios que se han comentado en el párrafo anterior, como se puede apreciar la respuesta parece algo mejor en los valores de "Validation" aunque el Accuracy sea algo peor, lo que es indicativo de que se iba por buen camino, pero aún se buscaban unos mejores resultados. A continuación, uno de los principales objetivos era suavizar los picos por si eran indicativos de "underfitting" u "overfitting".

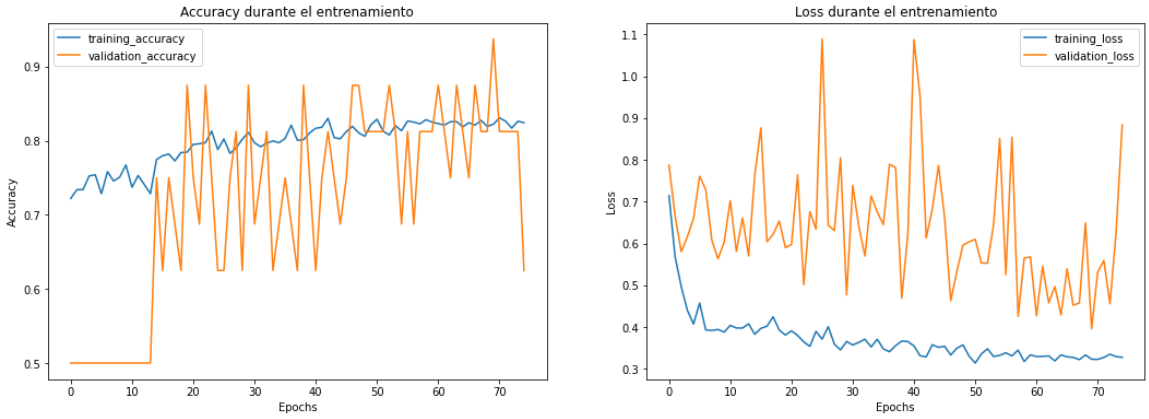


Figura 46. Valores de Accuracy y Loss del 2º modelo CNN durante el entrenamiento

5.1.3 Tercer modelo

Como se ha comentado anteriormente, el principal objetivo en este punto de la optimización de la CNN era eliminar esos picos que tan abundantes eran durante el entrenamiento y durante la validación, para ello el primer cambio que se hizo fue la modificación del valor del dropout para las capas, anteriormente el valor era del 40%, ahora se creó una nueva capa densa de 64 neuronas (figura 47) y se le aplicó un dropout del 60% con esto se consigue que cada capa tenga un dropout diferente y que las conexiones entre las neuronas sea todavía más aleatoria, no permitiendo así el aprendizaje automático (overfitting). Además, se realizó un cambio respecto a la BBDD, se realizó un reescalado diferentes a las imágenes, hasta el momento se había seleccionado un valor de 128x128 pensando en la arquitectura Transformer para que así estuviesen en igualdad de condiciones (ya se verá el por qué) y se probó con 144x144 a posteriori también, pero tras leer en varios foros que el tamaño de estas imágenes era muy importante se seleccionó un nuevo tamaño y este fue 130x130.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_17 (Conv2D)           (None, 128, 128, 32)      896
_________________________________________________________________
max_pooling2d_17 (MaxPooling (None, 64, 64, 32)        0
_________________________________________________________________
conv2d_18 (Conv2D)           (None, 62, 62, 64)        18496
_________________________________________________________________
max_pooling2d_18 (MaxPooling (None, 31, 31, 64)        0
_________________________________________________________________
conv2d_19 (Conv2D)           (None, 29, 29, 128)       73856
_________________________________________________________________
max_pooling2d_19 (MaxPooling (None, 14, 14, 128)       0
_________________________________________________________________
flatten_6 (Flatten)          (None, 25088)             0
_________________________________________________________________
dense_16 (Dense)             (None, 64)                1605696
_________________________________________________________________
dropout_11 (Dropout)         (None, 64)                0
_________________________________________________________________
dense_17 (Dense)             (None, 32)                2080
_________________________________________________________________
dropout_12 (Dropout)         (None, 32)                0
_________________________________________________________________
dense_18 (Dense)             (None, 2)                 66
=================================================================
Total params: 1,701,090
Trainable params: 1,701,090
Non-trainable params: 0
```

Figura 47. Estructura de la tercera CNN creada

Como se puede apreciar en la figura 48 los resultados no eran nada prometedores, estos seguían presentando múltiples picos durante todo el entrenamiento y los valores de “train” y “Validation” diferían de forma abrumadora.

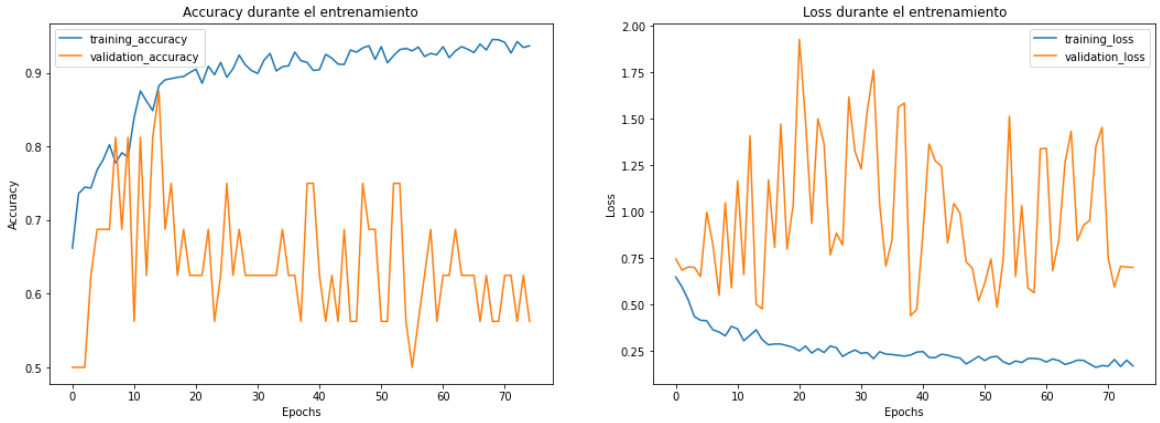


Figura 48. Valores de Accuracy y Loss del 3er modelo CNN durante el entrenamiento

5.1.4 Cuarto modelo

Hasta el momento no se habían conseguido unos resultados concluyentes ni se estaba conforme con los mismos ya que por más entrenamientos que se hiciesen no se conseguían mejorar. Se siguió buscando información para ver que parámetros mejorar en la red o en el tratamiento de los datos y tras leer diferentes papers [39] que hablaban del tema se tomó la determinación de bajar el learning rate a un valor muy pequeño (lr = 0.0005) con esto se consigue que los pesos de la red cambien muy poco en cada iteración, esto es debido a que solo existen dos clases finales y los cambios no son tan grandes como si se tuviese que diferenciar entre diferentes tipos de vehículos o entre diferentes tipos de animales… Y se añadió una capa más a la red densa de 128 neuronas con un dropout del 80% como se puede apreciar en la figura 49.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_26 (Conv2D)           (None, 128, 128, 32)      896
_________________________________________________________________
max_pooling2d_26 (MaxPooling (None, 64, 64, 32)        0
_________________________________________________________________
conv2d_27 (Conv2D)           (None, 62, 62, 64)        18496
_________________________________________________________________
max_pooling2d_27 (MaxPooling (None, 31, 31, 64)        0
_________________________________________________________________
conv2d_28 (Conv2D)           (None, 29, 29, 128)       73856
_________________________________________________________________
max_pooling2d_28 (MaxPooling (None, 14, 14, 128)       0
_________________________________________________________________
flatten_9 (Flatten)          (None, 25088)             0
_________________________________________________________________
dense_26 (Dense)             (None, 128)               3211392
_________________________________________________________________
dropout_18 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_27 (Dense)             (None, 64)                8256
_________________________________________________________________
dropout_19 (Dropout)         (None, 64)                0
_________________________________________________________________
dense_28 (Dense)             (None, 32)                2080
_________________________________________________________________
dropout_20 (Dropout)         (None, 32)                0
_________________________________________________________________
dense_29 (Dense)             (None, 2)                 66
=================================================================
Total params: 3,315,042
Trainable params: 3,315,042
Non-trainable params: 0
```

Figura 49. Estructura de la cuarta CNN creada

Una vez entrenada la red se obtuvieron los resultados que se muestran a continuación tanto en la figura 50, en ellas se muestran el accuracy a lo largo del entrenamiento durante todas las épocas y por otro lado se muestra la variable loss por época, tanto de los dataset de train como de validación. Como se puede apreciar ambas líneas en las dos gráficas son mucho más adyacentes que en el resto de los entrenamientos (figuras 44, 46, 48) vistos hasta el momento, pero siguen presentando los picos característicos vistos hasta el momento, después de continuar leyendo diferentes papers ([40] [41] [42]) sobre el tema de Deep Learning en imágenes médicas se concluyó que podría no ser un efecto negativo si no un efecto característico sobre el mismo. Además, se buscó en foros casos particulares y se vio que muchos presentaban casos similares y los calificaban como resultados positivos por lo cual, en este momento, se había obtenido el mejor de los resultados, el resultado más óptimo.

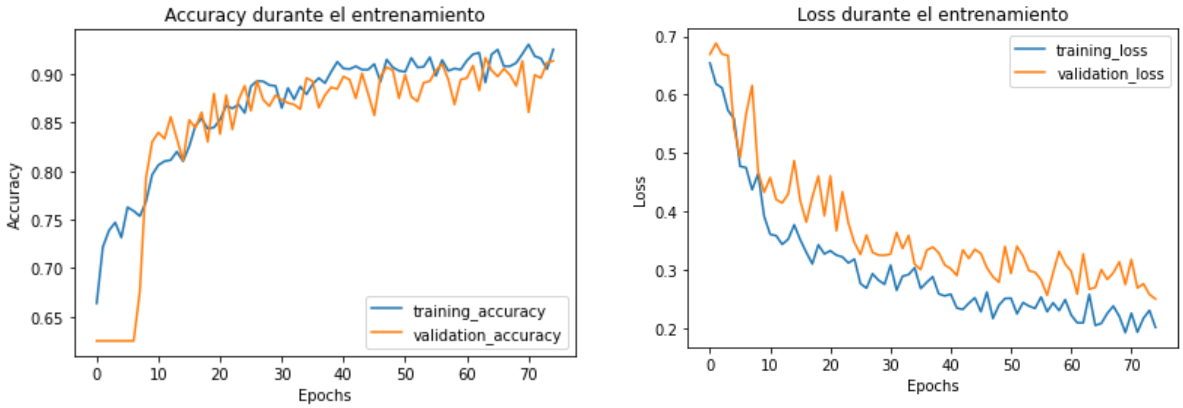


Figura 50. Accuracy y loss durante el entrenamiento óptimo CNN, respectivamente

Como se puede apreciar a simple vista en las figuras anteriores el resultado de esta red es bastante bueno, ya que rápidamente sube el accuracy tiene un valor en torno al 90% y las perdidas están alrededor de 20%. Durante el entrenamiento se pusieron diferentes cantidades de épocas para intentar mejorar el entrenamiento, pero únicamente se conseguía overfitting al pasar de las 100 épocas, esto puede ser debido a la BBDD. Si se mira en detalle la figura 50, se puede apreciar un pico bastante abundante en torno a la época 9 del valor de validation_loss, lo primero que se pensó es que podía ser indicativo de overfitting de nuevo, pero al bajar luego rápidamente, seguir buscando el resultado de training_loss y no presentar ningún otro tipo de pico de esa magnitud no se le dio mayor importancia. En la figura 51 se muestra la salida de las últimas 10 épocas devueltas por la terminal.

```
Epoch 65/75
27/27 [==============================] - 17s 619ms/step - loss: 0.2132 - accuracy: 0.9123 - val_loss: 0.2701 - val_accuracy: 0.9038
Epoch 66/75
27/27 [==============================] - 16s 592ms/step - loss: 0.1986 - accuracy: 0.9273 - val_loss: 0.3007 - val_accuracy: 0.8974
Epoch 67/75
27/27 [==============================] - 16s 595ms/step - loss: 0.2281 - accuracy: 0.9044 - val_loss: 0.2839 - val_accuracy: 0.9054
Epoch 68/75
27/27 [==============================] - 16s 599ms/step - loss: 0.2116 - accuracy: 0.9239 - val_loss: 0.2953 - val_accuracy: 0.8990
Epoch 69/75
27/27 [==============================] - 16s 599ms/step - loss: 0.2133 - accuracy: 0.9213 - val_loss: 0.3139 - val_accuracy: 0.8878
Epoch 70/75
27/27 [==============================] - 16s 594ms/step - loss: 0.1901 - accuracy: 0.9279 - val_loss: 0.2750 - val_accuracy: 0.9135
Epoch 71/75
27/27 [==============================] - 17s 617ms/step - loss: 0.2213 - accuracy: 0.9341 - val_loss: 0.3179 - val_accuracy: 0.8606
Epoch 72/75
27/27 [==============================] - 17s 617ms/step - loss: 0.1882 - accuracy: 0.9217 - val_loss: 0.2692 - val_accuracy: 0.8990
Epoch 73/75
27/27 [==============================] - 16s 594ms/step - loss: 0.2151 - accuracy: 0.9144 - val_loss: 0.2764 - val_accuracy: 0.8958
Epoch 74/75
27/27 [==============================] - 16s 589ms/step - loss: 0.2348 - accuracy: 0.8987 - val_loss: 0.2581 - val_accuracy: 0.9119
Epoch 75/75
27/27 [==============================] - 17s 616ms/step - loss: 0.1948 - accuracy: 0.9286 - val_loss: 0.2505 - val_accuracy: 0.9135
```

Figura 51. Salida por la terminal durante el entrenamiento con la CNN

Una vez entrenada la red, esta es capaz de diferenciar casi perfectamente si la imagen que está viendo es de una persona que presenta neumonía o que no. Un ejemplo de la salida podría ser la figura 52, en la cual se ha puesto de forma gráfica la salida de esta mediante el código adjuntado en el Anexo 3. Para tener una visión más objetiva de los resultados devueltos por la red, se realizó una evaluación del modelo con el dataset image_test:

- Accuracy: 89%
- Loss: 32%

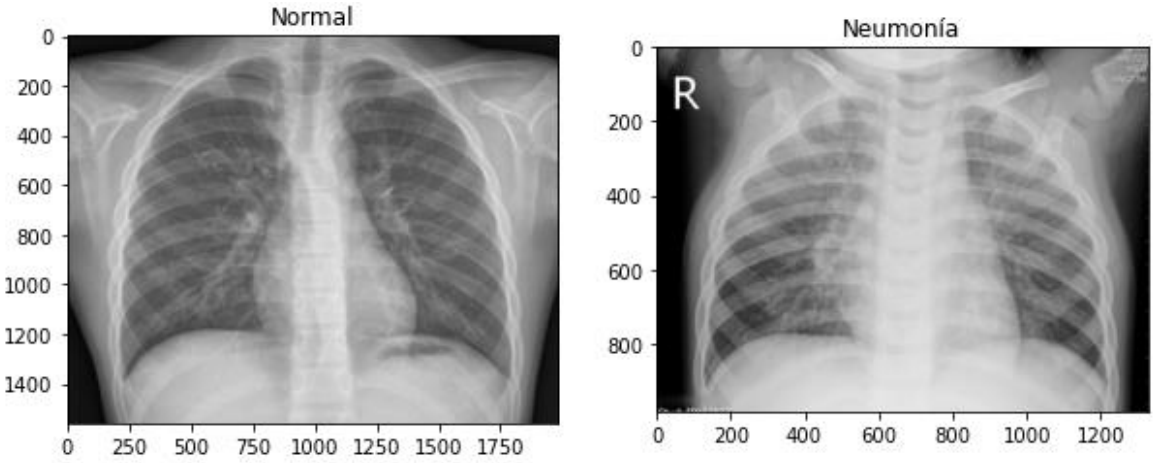


Figura 52. Representación de la salida de la CNN

5.2 Arquitectura ViT

Para la arquitectura ViT ya se tenía hecho una gran parte del trabajo que era la parte del tratamiento de los datos de la BBDD y los conocimientos sobre la importancia de ciertos parámetros dentro de las imágenes médicas, como es la importancia de la escala de grises y el tamaño a la hora de ser reescaladas, como ya se comentó en el apartado 5.1. En el aspecto del tratamiento de los datos el mayor reto fue la creación de la función give_set mostrada en el apartado 4.4.1 (figura 36), la gran diferencia respecto a la CNN es que para la anterior se hizo uso de la clase facilitada por Keras "ImageDataGenerator" (apartado 4.3.1) pero a partir de este momento no podía ser usada ya que el esqueleto de este ViT estaba pensado para recibir dos variables distintas, por un lado las imágenes y por otra sus correspondientes etiquetas, además, de que al no hacer uso de dicha clase no se podría usar "Data Augmentation" de una manera tan clara y sencilla, por el contrario habría que aplicarlo de forma manual. Para la parte de giro, zoom e inclinación no hubo demasiados problemas ya que el creador de la estructura lo implementó dentro del código facilitado y únicamente se tuvo que adaptar a la nueva terminología e ir modificando las diferentes líneas para conseguir el efecto deseado, pero por otro lado para la parte del reescalado si surgieron problemas, ya que implementar la normalización los valores de los pixeles la imagen de salida era completamente negra. Se intentó realizar de múltiples formas, pero ninguna de ellas exitosas por lo que se decidió no realizar este paso y aunque esto supusiera un poco más de computación se sabría con certeza que las imágenes eran las correctas.

Como se ha adelantado anteriormente en un principio el tamaño seleccionado para el reescalar las imágenes había sido 128x128, esto fue así para favorecer a esta arquitectura a la hora de realizar el tratamiento de las imágenes, como ya se comentó en el aparatado 4.4.1 es necesario crear segmentos de 16x16 de cada una de las imágenes por lo que la idea principal fue reescalar a un múltiplo exacto de dicho número saliendo así un número total de 8x8 segmentos pero la CNN no trabajaba bien con este tamaño de imagen, por lo cual se optó por el siguiente valor exacto que era 144x144 consiguiendo así 9x9 particiones exactas por pero la CNN seguía devolviendo valores realmente malos. Tras buscar por diferentes foros se descubrió que se recomendaba el tamaño de 130x130, este tamaño no era del todo positivo ya que si se divide 130 entre 16 el número resultante no es exacto, por lo cual existirán 9 segmentos que tendrán que ser rellenados con 0's para conseguir que todos sean de igual tamaño. Lo primero que se hizo fue comprobar el funcionamiento de la CNN con este tamaño de reescalado y los resultados eran muy prometedores (apartado 5.1.3) por lo cual solo quedaba probarlo en este nuevo modelo, tras revisar detenidamente las imágenes se vio que todas las imágenes en ambos extremos laterales presentaban un mayor porcentaje de negro ya que este era el fondo (figura 39), por lo cual, existía una posibilidad de que esto no fuese un problema para el entrenamiento de esta nueva arquitectura.

En cuanto a la arquitectura, posiblemente el trabajo más duro de este apartado y posiblemente de este trabajo, este fue en general entender y ser capaz de comprender el funcionamiento de la arquitectura creada por Khalid Salama [38] para así poder tocar cada uno de los parámetros necesarios para ajustar la red a la BBDD de este trabajo sin estropear el funcionamiento de la arquitectura. Una vez realizado este trabajo, se eligió una red ViT lo más básica posible pero obviamente siguiendo las recomendaciones de los papers originales [29] [34], desde un principio la red estuvo compuesta por 8 capas Transformer (figura 26 L = 8) y por 4 capas Multi-Head Attention (figura 20 h = 4). Otro parámetro muy importante es la matriz de proyección y viene definido por la variable "projection_dim", dicho parámetro también fue bastante difícil de calcular ya que es bastante difícil de calcular las dimensiones de cada una de las variables con las cuales se trabaja y si dicho parámetro no está correctamente asignado inmediatamente salta un error avisando de que las dimensiones no son correctas, en resumen se puede llegar a la conclusión de que este parámetro tiene que ser del mismo tamaño que el batch_size y como para la CNN fue seleccionado un tamaño de 64 para este apartado se mantuvo para que ambas estuviesen en igualdad de condiciones a la hora de realizar la futura comparación. Además, el learning rate fue seleccionado con el mismo valor que en la CNN también.

5.2.1 Primer modelo

En un primer lugar se pecó un poco de avaricioso y aunque ya se ha comentado anteriormente que la estructura sería lo más básica posible dentro de unos límites, pero en la parte de las MLP que serán las encargadas de generar la clasificación mediante la entrada de los valores Attention (a_N) al inicio sí que se tomó una red demasiado extensa, formada por tres capas de 64, 32, 2 neuronas, respectivamente. Además, como ya se explicó en el apartado 4.4.2 se hace uso de un nuevo parámetro que no fue usado en la CNN denominado "weight_decay" (apartado 4.4.2.), este parámetro era desconocido hasta el momento y por lo cual hasta llegar al valor óptimo se fueron probando diferentes valores y ajustándolo, en un primer lugar se seleccionó un valor de 0.01.

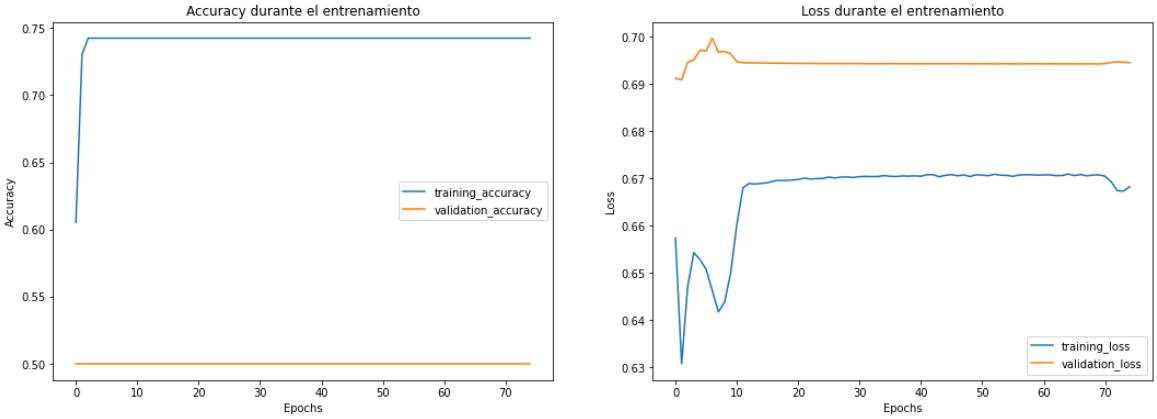


Figura 53. Valores de Accuracy y Loss del 1er modelo ViT durante el entrenamiento

Como se puede apreciar en la figura 53 los resultados no eran nada prometedores. Los valores de validación eran muy malos en comparación al primer resultado obtenido para la CNN (figura 44) no tenían nada que ver, además, no se sabía muy bien como analizar los resultados ya que nunca se habían visto unos resultados parecidos en ningún otro experimento. Se optó por seguir modificando los parámetros un poco a ciegas hasta conseguir unos resultados con los cuales poder avanzar.

5.2.2 Segundo modelo

Para este segundo modelo se cambiaron los parámetros "weight_decay" (apartado) a un valor de 0.001 y la red MLP se modificó a 128, 32, 2 neuronas por capa, respectivamente, con esto se pretendía penalizar menos los cambios bruscos de los pesos, que en principio no deberían existir porque los cambios entre imágenes no serían muy grandes y conseguir una mayor diversificación de los valores devueltos por la técnica Attention gracias a las 128 neuronas intentando así que cada neurona "cargue con menos peso durante el entrenamiento".

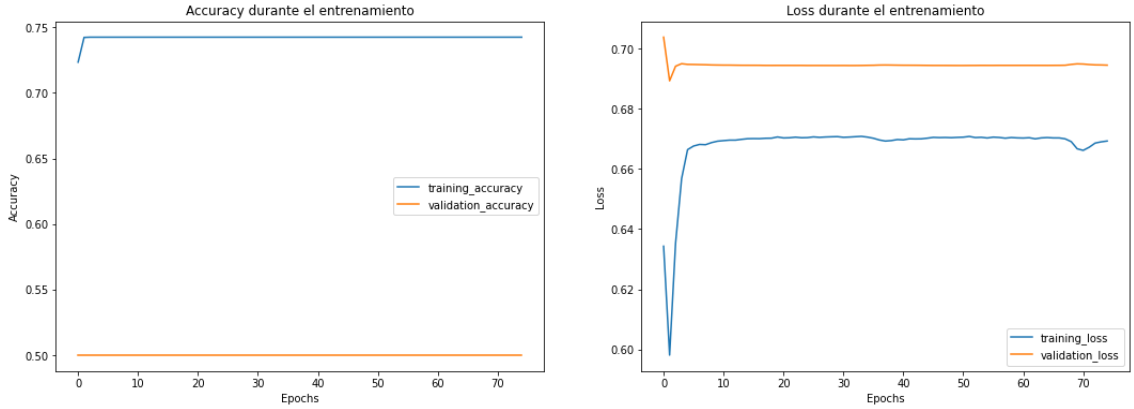


Figura 54. Valores de Accuracy y Loss del 2º modelo ViT durante entrenamiento

Como se puede apreciar en la figura 54 los valores no mejoraban con los cambios realizados, los valores del Accuracy no respondía, se mantenía constante en un 75% aproximadamente y en caso del Loss también se mantenía constante durante prácticamente todo el entrenamiento en torno al 69%, todo esto en la gráfica de "training", en la gráfica de "Validation" los resultados son peores si cabe, en resumen, todos los resultados eran demasiado lineales y esto puede ser caso de overfitting ya que la red puede haber aprendido de memoria y no a clasificar.

5.2.3 Tercer modelo

Con intención de solventar el overfitting la MLP se redujo drásticamente y se cambiaron los valores de dropout que traía por defecto el modelo, fueron seleccionadas tres capas de 32, 16 y 2 neuronas cada una de ellas respectivamente y se seleccionó un dropout de un 50% para todas las capas ya que no era posible seleccionar distintos valores para cada una de las capas. Y, el valor de "weight_decay" se mantuvo en su valor anterior ya que parecía lo suficientemente bajo.

El resultado de estos cambios está representado en la figura 55. Como se puede apreciar los valores de "Validation" fueron capaces de responder ante el entrenamiento por lo cual, se sabe con certeza que los resultados obtenidos anteriormente estaban mal y los cambios efectuados han sido beneficiosos para la experimentación, aunque como se puede ver todavía no era una gráfica como la obtenida en la experimentación de la CNN. Además, El accuracy en la gráfica de training alcanzaba más del 90% de precisión demasiado deprisa por lo cual se le realizó una evaluación con el set "Value" y el resultado confirmó dicha sospecha:

- Accuracy = 28%
- Loss = 87%

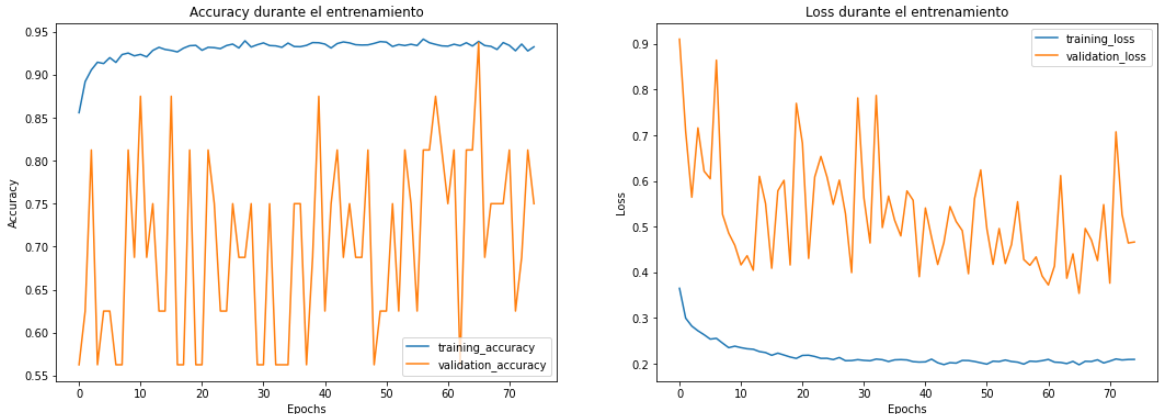


Figura 55. Valores de Accuracy y Loss del 3er modelo ViT durante entrenamiento

5.2.4 Cuarto modelo

Hasta el momento, no se habían obtenido los resultados deseados y no se sabía muy bien el por qué ya todo lo que se modificaba no funcionaba. Uno de los cambios que se realizaron en este apartado fue en el apartado del tratamiento de los datos y es en la función give_set, hasta el momento se tenía "return x, y" y al mirar la documentación de Keras se descubrió que la función empleada con la CNN automáticamente barajaba los dataset para volverlos así aleatorios y por lo tanto se cambió dicha salida por la salida mostrada en la figura 36. En segundo lugar, se pensó que si estaba aplicando un dropout del 50% de las neuronas en cada capa incluida la del final, cabía la posibilidad de que al solo tener dos neuronas y aplicar el dropout no realizase bien la clasificación, en principio esto no debería de ser un problema ya que como se explicó en el apartado 2.2.1 una única neurona puede realizar una clasificación de dos clases pero la mayoría de autores siempre recomienda tener tantas neuronas activas en la capa final como clases se tengan que clasificar, es decir, dos neuronas

en la capa de salida. Y por último quedaba saber él por qué de los resultados tan raros en la gráfica de "Validation", por lo que se tomó la determinación de analizar con detenimiento los resultados obtenidos por Khalid Salama durante su experimentación y ahí se obtuvo una grata sorpresa, sus resultados en la parte de "Validation" tampoco eran nada buenos y sin embargo él en sus comentarios mostraba su alegría con los resultados obtenidos siendo realista de que no es para nada un modelo final, pero no era un parámetro que le importarse, por lo cual, a partir de este momento, ese parámetro pasó a un segundo plano, solo se tendría en cuenta para ver que no hubiese overfitting como pasaba en los apartados 5.2.1 y 5.2.2.

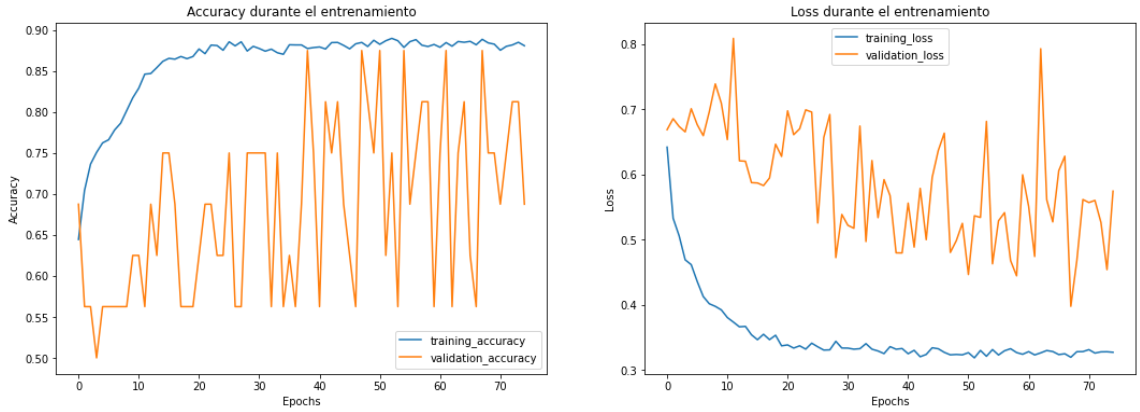


Figura 56. Valores de Accuracy y Loss del 4º modelo ViT durante entrenamiento

Como se puede apreciar en la figura 56, los resultados son mucho más prometedores, por un lado, el accuracy no parte directamente del 80-90% lo cual es bastante positivo, el entrenamiento no es lineal como si se mostraba anteriormente y los valores de "Validation" no son planos por lo cual no presenta overfitting en principio. Una vez visto los resultados devueltos por la arquitectura quedaba verificar los resultados mediante una evaluación con el set "Value".

- Accuracy = 85'96%
- Loss = 37'47%

Ahora sí, los resultados eran mucho mejores y eran más o menos los esperados al iniciar este proyecto, pero… ¿Cuál de los dos cambios ha sido el causante de la mejora?

5.2.5 Quinto modelo

La creación de este modelo no era del todo necesaria ya que los resultados obtenidos anteriormente eran más que satisfactorios, pero se realizó por simple curiosidad de cuál de los dos cambios había sido el que causó la mejora. Por lo cual, para este apartado únicamente se modificó la última capa de la MLP y se volvió a poner únicamente dos neuronas ya que el cambio de realizar un shuffle de los datos antes de entrenar a la red parecía mucho más coherente.

En la figura 57 se pueden apreciar que los resultados no son muy dispares de los mostrados por el anterior modelo, por lo cual se puede llegar a la conclusión de que el error principal, entre otros muchos, era la falta de la inclusión shuffle a la hora de pasarle los datos a la red para ser entrenado. Ambos modelos, tanto este quinto modelo como el anterior, cumplen con las expectativas y son capaces de ofrecer resultados muy interesantes.

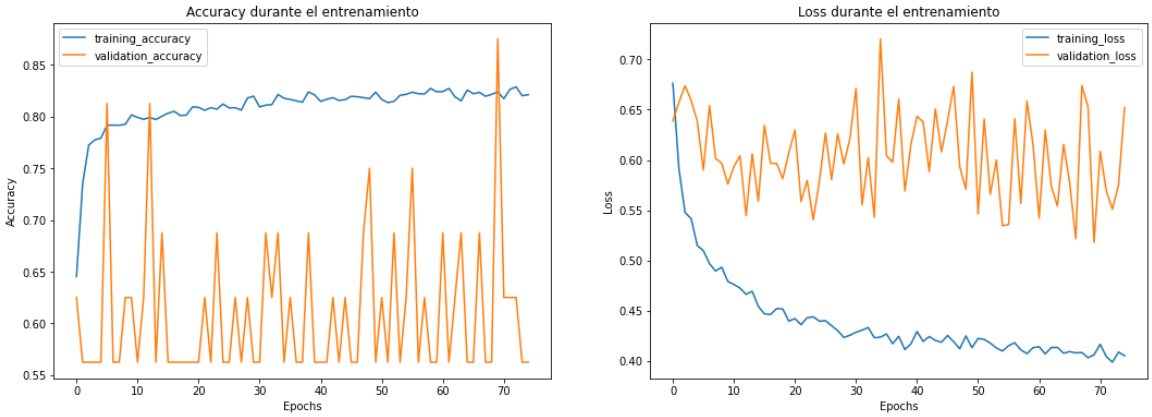


Figura 57. Resultados durante el entrenamiento con ViT. Accuracy y Loss, respectivamente

En la figura 57, se puede apreciar este último modelo con la diferencia respecto al anterior de presentar únicamente dos neuronas en la capa final. Como se puede apreciar los resultados no son muy dispares unos de otros por lo cual se llega a la conclusión de que el cambio que hizo que la red comenzase a funcionar correctamente fue el shuffle de los datos y no esta modificación. Entre estos dos últimos modelos no existe una gran diferencia y ambos son totalmente válidos como resultado final de este proyecto fin de carrera, aunque como es obvio se tomará la que ha dado un mejor porcentaje de acierto como es el cuarto modelo para la comparación con la CNN en el siguiente apartado.

```
Epoch 65/75
82/82 [==============================] - 5s 58ms/step - loss: 0.4126 - accuracy: 0.8183 - top-5-accuracy: 1.0000 - val_loss: 0.7068 - val_accuracy: 0.5625 - val_top-5-accuracy: 1.0000
Epoch 66/75
82/82 [==============================] - 5s 58ms/step - loss: 0.4145 - accuracy: 0.8147 - top-5-accuracy: 1.0000 - val_loss: 0.5375 - val_accuracy: 0.6875 - val_top-5-accuracy: 1.0000
Epoch 67/75
82/82 [==============================] - 5s 58ms/step - loss: 0.4190 - accuracy: 0.8244 - top-5-accuracy: 1.0000 - val_loss: 0.5565 - val_accuracy: 0.6875 - val_top-5-accuracy: 1.0000
Epoch 68/75
82/82 [==============================] - 5s 58ms/step - loss: 0.4166 - accuracy: 0.8195 - top-5-accuracy: 1.0000 - val_loss: 0.5422 - val_accuracy: 0.6250 - val_top-5-accuracy: 1.0000
Epoch 69/75
82/82 [==============================] - 5s 57ms/step - loss: 0.4143 - accuracy: 0.8140 - top-5-accuracy: 1.0000 - val_loss: 0.5499 - val_accuracy: 0.6250 - val_top-5-accuracy: 1.0000
Epoch 70/75
82/82 [==============================] - 5s 58ms/step - loss: 0.4157 - accuracy: 0.8155 - top-5-accuracy: 1.0000 - val_loss: 0.5475 - val_accuracy: 0.6250 - val_top-5-accuracy: 1.0000
Epoch 71/75
82/82 [==============================] - 5s 58ms/step - loss: 0.4140 - accuracy: 0.8180 - top-5-accuracy: 1.0000 - val_loss: 0.5210 - val_accuracy: 0.8125 - val_top-5-accuracy: 1.0000
Epoch 72/75
82/82 [==============================] - 5s 59ms/step - loss: 0.4167 - accuracy: 0.8189 - top-5-accuracy: 1.0000 - val_loss: 0.5089 - val_accuracy: 0.7500 - val_top-5-accuracy: 1.0000
Epoch 73/75
82/82 [==============================] - 5s 58ms/step - loss: 0.4081 - accuracy: 0.8199 - top-5-accuracy: 1.0000 - val_loss: 0.5961 - val_accuracy: 0.5625 - val_top-5-accuracy: 1.0000
Epoch 74/75
82/82 [==============================] - 5s 58ms/step - loss: 0.4108 - accuracy: 0.8181 - top-5-accuracy: 1.0000 - val_loss: 0.4817 - val_accuracy: 0.8750 - val_top-5-accuracy: 1.0000
Epoch 75/75
82/82 [==============================] - 5s 58ms/step - loss: 0.4168 - accuracy: 0.8168 - top-5-accuracy: 1.0000 - val_loss: 0.4968 - val_accuracy: 0.8125 - val_top-5-accuracy: 1.0000
```

Figura 58. Salida por la terminal durante el entrenamiento con ViT

Al igual que con el resto de las experimentaciones se realizó una evaluación del modelo para comprobar el resultado final del modelo que como se ha adelantado anteriormente es ligeramente peor que cuarto modelo.

- Accuracy: 85’65%
- Loss: 40’94%

Por último, para tener una visión de lo que la red estaba devolviendo, se adaptó el código creado para el apartado 5.1.4 que devolvió la figura 52 para esa arquitectura y el resultado, como es obvio, es muy parecido y se muestra en la figura 59.

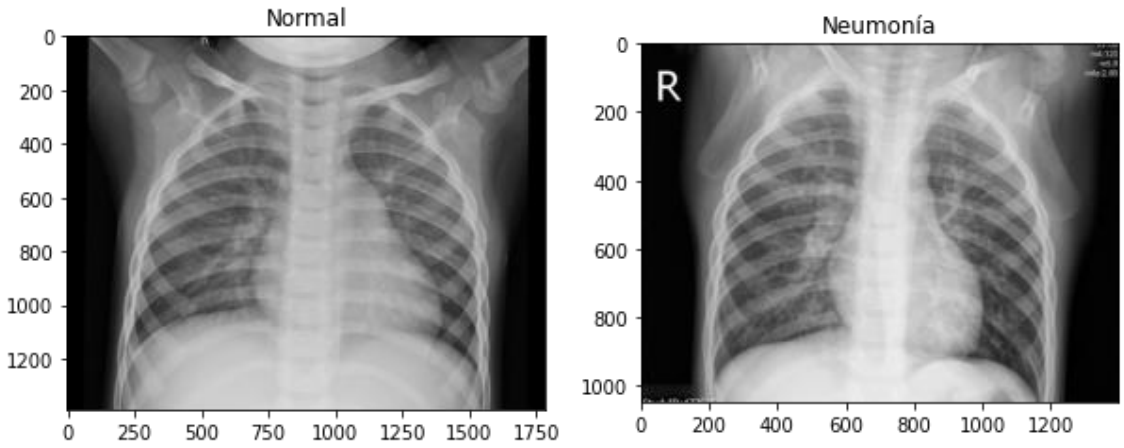


Figura 59. Representación de la salida del ViT

5.3 Resumen

Como resumen se presenta la tabla 5, en dicha tabla se muestran tanto los resultados finales de accuracy y loss de las dos arquitecturas como tiempos de computación. Hay que destacar que, aunque a primera vista numéricamente pueda parecer que la CNN es mucho mejor en cuanto a resultados, en el siguiente apartado de Conclusiones y líneas futuras se verán en detalle las conclusiones y las líneas futuras en las cuales se detalla que este tipo de arquitectura aún está en una fase muy temprana del desarrollo por lo cual hay que tomar los datos con cierta cautela.

En primer lugar, se puede apreciar que los valores de accuracy no difieren demasiado, pero en esta ocasión la CNN es mejor que la arquitectura ViT. En segundo lugar, se pueden apreciar la comparación de los valores de loss en los cuales se puede ver que la CNN sí que es mucho mejor que la nueva arquitectura ViT. Por último, se tienen los tiempos de computación durante el entrenamiento de ambas arquitecturas y en esta ocasión la arquitectura ViT es la clara vencedora consiguiendo unos tiempos muchos menores.

	CCNN	ViT
Accuracy	88’9%	85’96%
Loss	32%	37’47%
Tiempo computación	~ 21 min	~ 4 min

Tabla 5. Comparación de las dos arquitecturas

6 CONCLUSIONES Y LÍNEAS FUTURAS

Aunque a primera vista (tabla 5) pueda parecer que la CNN es mucho mejor que el ViT en términos de resultados, como ya se comentó las figuras 39 y 49 han sido realizadas con un pequeño código, el cual, una vez entrenada la red en cuestión, recibía una imagen y mediante la clase predict del paquete de Keras intentaba predecir de que tipo era la imagen. Cuando se realizaba dicho experimento en bloques de 12 tiempos con cada una de las redes, la red ViT devolvía muchos mejores resultados que la red CNN, por ello, se subieron las iteraciones a 24 y el resultado era similar, la red ViT devolvía mejores resultados que la CNN, aproximadamente de cada 24 intentos la red ViT fallaba 3 o 4 y la red CNN hasta 9 en sus peores resultados.

Aun así, con este trabajo únicamente se tenía el objetivo de introducir al mundo de las arquitecturas Transformer ya que son el nuevo presente dentro de este apasionante mundo del Deep Learning. Es tanto así que, Google, en su conferencia anual Google I/O 2021 en la cual presentan todas las novedades en las que han estado trabajando en los últimos años y en las que siguen trabajando, su mayor novedad fue la implementación de la arquitectura Transformer para la gran mayoría de servicios que ofrecen hoy en día como es el motor de búsqueda de Google, Google Translate, Google Lens para el reconocimiento de texto en tiempo real, etc. Además, de presentar LaMDA que hace competencia directa a la actual GPT-3, dicho modelo también implementa una arquitectura Transformer para NLP en tiempo real.

No cabe duda de que en cuanto la arquitectura ViT esté tan madura como lo está a día de hoy las redes Transformer, las CNN lo tendrán muy difícil para no ser desplazadas de este campo del Deep Learning, ya que como se puede ver en la tabla 5, los tiempos de computación son mucho más pequeños y los resultados pese a ser un modelo en estado “Beta” o incluso “Alpha” ha conseguido ponérselo muy difícil.

Como trabajos futuros respecto a este proyecto existen varias posibilidades, una de ellas, es continuar con el desarrolle e implantación de las redes ViT con paquetes optimizados por bibliotecas como PyTorch la cual, recientemente ha lanzado uno para las redes ViT con el cual es mucho más fácil su implementación además de no tener que optimizar capa a capa manualmente. También se puede intentar optimizar aún más la red de Khalid Salama concatenando de forma distinta las capas o usando otro algoritmo para la creación de las múltiples capas Transformer en vez de hacer uso del for o cualquier otro cambio que permita mejorar el rendimiento de la red sin perjudicar los resultados. Por otro lado, al estar el mundo del Deep Learning en constante desarrollo se pueden intentar implementar otras nuevas arquitecturas como pueden ser un Perciber de la compañía DeepMind o una MLP Mixer, arquitectura que ha sido desarrollada a partir del ViT y hace uso de su procesado de imagen en patches de 16x16 pero en vez de hacer uso de un codificador únicamente hace uso de múltiples MLPs.

Como uno se puede imaginar este campo tiene multitud de opciones, para acotar un tanto por ciento los resultados, únicamente se nombrarán los proyectos más grandes hoy en día en análisis de imágenes médicas en los cuales puede ser usada la red ViT y de NPL con el uso del Transformer. Por ejemplo, Watson, la IA de IBM para el reconocimiento de Melanomas que actualmente está siendo entrenada con aproximadamente 30.000 millones de imágenes trabaja con CNNs, por lo cual la computación que ha de soportar la infraestructura que rodea dicha tecnología ha de ser muy elevada. Una vez la arquitectura ViT sea lo suficientemente estable y sea capaz de devolver unos resultados tan buenos como las CNN, ya se ha visto que no está muy lejos, con un 400% menos de computación como aseguraba Alexey Dosovitskiy es más que probable que sea reemplazada. Si se quiere buscar una referencia de los resultados que obtendría esta arquitectura con una base de datos tan grande, es recomendable irse al paper original [34] y buscar la tabla 5 en el apartado C “Additional Results”, en dicha tabla podemos ver los resultados obtenidos para la BBDD JFT-300M la cual presenta como se puede intuir 300 millones de imágenes que son menos de las cuales tiene IBM pero aun así es un número enorme, y presenta unos resultados con accuracies en torno al 90%, 84% en el peor resultado obtenido. Por otro lado, si nos centramos en el NPL con la arquitectura Transformer la

presentación más reciente e innovadora ha sido “GitHub Copilot”, esta herramienta ha sido desarrollada por GitHub, OpenAI y Azure (Microsoft) las cuales posiblemente sean 3 de las empresas tecnológicas más fuertes en este campo. Esta herramienta está formada en su interior por una arquitectura Transformer que en vez de ser entrenada con texto normal y corriente ya sea en un idioma en concreto para crear un modelo como GPT-3 o en varios idiomas para generar un modelo de traducción, ha sido entrenada con una base de datos de código principalmente en Python ofrecida por GitHub como no podría ser de otra forma, de esta forma, esta herramienta permite que mediante una entrada de texto natural en forma de comentario o incluso únicamente poniendo el nombre de la función que se quiera desarrollar, ella automáticamente es capaz de crear todo el código de forma automática incluso si ya se ha creado un código intentará mejorarlo en la medida de lo posible haciendo el código más legible para futuros desarrolladores que necesiten leer dicho código o haciéndolo más óptimo. Aunque, su presentación se ha centrado principalmente en Python debido a su gran potencial en el mundo del desarrollo de software, pero también puede ser usado para otro con otros lenguajes como son JavaScript, TypeScript, Ruby and Go, han sido seleccionados estos lenguajes debido a la cantidad de gente que los usa. Actualmente, según el propio paper presentado por GitHub, este modelo presenta un acierto del 43% para la creación de código en una primera iteración, es decir, que es capaz de crear el código perfecto que se le ha indicado de una solo vez sin necesidad de repetir la iteración, en caso de hacerlo, el porcentaje de acierto está en torno al 55%. Como se puede ver, el campo del desarrollo de estas dos tecnologías no acaba nada más que empezar.

REFERENCIAS

[1] A. García Serrano, INTELIGENCIA ARTIFICIAL: FUNDAMENTOS, PRACTICA Y APLICACIONES, 2nd ed. Madrid, 2016, p. 296.

[2] A. Pant, "Introduction to Machine Learning for Beginners", Medium, 2019. [Online]. Available: https://towardsdatascience.com/introduction-to-machine-learning-for-beginners-eed6024fdb08. [Accessed: 17- Apr- 2021].

[3] F. Chollet, Deep learning with Python. 2018, p. Chapter 1: What is Deep Learning?

[4] M. Gorini, "¿Cuál es la diferencia entre el machine learning y el deep learning?", Blog.bismart.com, 2020. [Online]. Available: https://blog.bismart.com/es/diferencia-machine-learning-deep-learning. [Accessed: 17 - Apr - 2021].

[5] "Types of Neural Networks and Definition of Neural Network", GreatLearning Blog: Free Resources what Matters to shape your Career!, 2020. [Online]. Available: https://www.mygreatlearning.com/blog/types-of-neural-networks/#Perceptron. [Accessed: 20 – Apr - 2021].

[6] L. Guesmi, "Researchgate", Researchgate, 2018. [Online]. Available: https://www.researchgate.net/figure/McCulloch-Pitts-computational-model-of-a-neuron_fig2_323465059 [Accessed: 20- Apr- 2021].

[7] "Types of Neural Networks and Definition of Neural Network", GreatLearning Blog: Free Resources what Matters to shape your Career!, 2020. [Online]. Available: https://www.mygreatlearning.com/blog/types-of-neural-networks/#Perceptron. [Accessed: 20- Apr- 2021].

[8] "Redes Neuronales," *Github.io*, 2012. https://ml4a.github.io/ml4a/es/neural_networks/ [Accessed 24 – May – 2021].

[9] P. Jain, "Laptrinhx", *Laptrinhx*, 2019. [Online]. Available: https://laptrinhx.com/complete-guide-of-activation-functions-574622854/. [Accessed: 20- Apr- 2021].

[10] Hamed Rahimi Nohooji, "Dynamic analysis and intelligent control techniques for flexible manipulators: a review," ResearchGate, Aug. 2018. https://www.researchgate.net/publication/326735709_Dynamic_analysis_and_intelligent_control_techniques_for_flexible_manipulators_a_review (accessed Jun. 30, 2021).

[Accessed: 20- Apr- 2021].

[11] Y. Manolopoulos, B. Hammer, L. Iliadis and E. Maglogiannis, "Artificial Neural Networks and Machine Learning", in *27th International Conference on Artificial Neural Networks*, Rhodes (Greece), 2018, p. Chapter: A Convolutional Neural Network Approach for Modeling Semantic Trajectories and Predicting Future Locations.

[12] F. Chollet, Deep learning with Python. 2018, p. Chapter 5: Deep Learning for computing vision.

[13] R. Venkatesan and B. Li, Convolutional Neural Network in visual computing. Arizona State University, Phoenix,(USA): CRC Press, 2018, pp. Chapters 4 and 5.

[14] A. Géron, Hands on machine learning with scikit-learn and TensorFlow. O'Reilly Media, Inc., 2017, pp. Pages: 358 - 360.

[15] E. Decena, “Entendiendo las redes neuronales PART 1 - Eddy Decena - Medium,” Medium, Jun. 25, 2019. https://medium.com/@eddydecena/entendiendo-las-redes-neuronales-part-1-fca3adf78c5b (accessed Jul. 11, 2021).

[16] Y. Sai Palaghat, "What can convolutional neural network do? - Quora", *Quora.com*, 2019. [Online]. Available: https://www.quora.com/What-can-convolutional-neural-network-do. [Accessed: 21- Apr-2021].

[17] "Understanding of Convolutional Neural Network (CNN) — Deep Learning", *Medium*, 2018. Available: https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148. [Accessed: 21- Apr- 2021].

[18] "(1/3) El rol del Open Source en el desarrollo de la Inteligencia Artificial y el Deep Learning | GUTL", *GUTL*, 2019. [Online]. Available: https://gutl.jovenclub.cu/rol-open-source-inteligencia-artificial-deep-learning/. [Accessed: 21- Apr- 2021].

[19] “Is overfitting okay if the test accuracy is high enough? - Quora,” Quora.com, 2021. https://www.quora.com/Is-overfitting-okay-if-the-test-accuracy-is-high-enough [Accessed 23-May-2021).

[20] “CS231n Convolutional Neural Networks for Visual Recognition,” Github.io, 2014. https://cs231n.github.io/neural-networks-2. [Accessed 23-May-2021].

[21] Arun Gandhi, “set Augmentation | How to use Deep Learning when you have Limited Data,” AI & Machine Learning Blog, May 19, 2021. https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/ [Accessed 23-May-2021].

[22] Data augmentation, “Aumento de datos | TensorFlow Core,” *TensorFlow*, 2021. https://www.tensorflow.org/tutorials/images/data_augmentation [Accessed 23-May-2021].

[23] L. Taylor and G. Nitschke, "Improving Deep Learning with Generic Data Augmentation", Bangalore (India), 2018.

[24] N. Adaloglou, "How Attention works in Deep Learning: understanding the attention mechanism in sequence models", *Theaisummer.com*, 2019. [Online]. Available: https://theaisummer.com/attention/. [Accessed: 25- Apr- 2021].

[25] W. Wang and J. Shen, "Deep Visual Attention Prediction," in IEEE Transactions on Image Processing, vol. 27, no. 5, pp. 2368-2378, May 2018, doi: 10.1109/TIP.2017.2787612.

[26] P. Rajpurkar, "Visualizing A Convolutional Neural Network's Predictions", *mlx*, 2017. [Online]. Available: https://rajpurkar.github.io/mlx/visualizing-cnns/. [Accessed: 25- Apr- 2021].

[27] A. Everyone, "Attention Mechanism In Deep Learning | Attention Model Keras", *Analytics Vidhya*, 2021. [Online]. Available: https://www.analyticsvidhya.com/blog/2019/11/comprehensive-guide-attention-mechanism-deep-learning/. [Accessed: 03- May- 2021].

[28] M. Luong, H. Pham and C. D. Manning, "Effective Approaches to Attention-based Neural Machine Translation", Computer Science Department (Stanford University), Stanford, 2015.

[29] A. Vaswani et al., "Attention is all you need", no. 1706037625, 2017. Available: https://arxiv.org/abs/1706.03762.

[30] Maxime. "What is a Transformer?" Medium. https://medium.com/inside-machine-learning/what-is-a-transformer-d07dd1fbec04 [Accessed 28 – May - 2021].

[31] J. Alammar. "The Illustrated Transformer". Jay Alammar – Visualizing machine learning one concept at a time. https://jalammar.github.io/illustrated-transformer/ [Accessed 28 - May - 2021].

[32] "Transformers from scratch | peterbloem.nl", *Peterbloem.nl*, 2019. [Online]. Available: http://peterbloem.nl/blog/transformers. [Accessed 29-May-2021].

[33] "Transformers for Image Recognition at Scale," Google AI Blog, Dec. 03, 2020. https://ai.googleblog.com/2020/12/transformers-for-image-recognition-at.html [Accessed 4-Jun-2021].

[34] A. Dosovitskiy et al., " An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale", no. 2010.11929, 2020. Available: https://arxiv.org/abs/2010.11929v2.

[35] "Data-efficient image Transformers: A promising new technique for image classification," Facebook AI Blog, Dec. 23, 2020. https://ai.facebook.com/blog/data-efficient-image-transformers-a-promising-new-technique-for-image-classification/ [Accessed 4-Jun-2021].

[36] Keras Team, "Keras documentation: Keras API reference," Keras.io, 2021. https://keras.io/api/ [Accessed all the time].

[37] "TensorFlow Core," TensorFlow, 2019. https://www.tensorflow.org/tutorials [Accessed all the time]

[38] Keras Team, "Keras documentation: Image classification with Vision Transformer," Keras.io (Khalid Salama), 2021. https://keras.io/examples/vision/image_classification_with_vision_transformer/ [Accessed 18-Jun-2021].d

[39] Shen, D., Wu, G., & Suk, H. I. (2017). Deep learning in medical image analysis. *Annual review of biomedical engineering*, *19*, 221-248.

[40] J. Ker, L. Wang, J. Rao and T. Lim, "Deep Learning Applications in Medical Image Analysis," in IEEE Access, vol. 6, pp. 9375-9389, 2018, doi: 10.1109/ACCESS.2017.2788044.

[41] Lundervold, A. S., & Lundervold, A. (2019). An overview of deep learning in medical imaging focusing on MRI. *Zeitschrift für Medizinische Physik*, *29*(2), 102-127.

[42] Lundervold, A. S., & Lundervold, A. (2019). An overview of deep learning in medical imaging focusing on MRI. *Zeitschrift für Medizinische Physik*, *29*(2), 102-127.

ANEXO A

Código correspondiente a la CNN, hay que recordar que a este código se puede acceder de forma online mediante el siguiente enlace:

https://colab.research.google.com/drive/11KauBs8Wkcm0aXvG9dGcgCgZDes7JrNQ?usp=sharing.

```
# -*- coding: utf-8 -*-
"""
Created on Sun May 16 6:15:57 2021

@author: imata
"""

#Imports
import os
import sys
import numpy as np
import random
import keras
import tensorflow as tf
import pydot
import pydotplus
import sklearn
import matplotlib.pyplot as plt
from keras import layers
from keras import models
from keras.utils.vis_utils import plot_model
from keras.utils.vis_utils import plot_model
from keras.utils.vis_utils import model_to_dot
from keras.preprocessing.image import load_img
from keras.layers import Dropout, Flatten, Dense, Activation
from keras.preprocessing.image import ImageDataGenerator
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import normalize
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from tqdm import tqdm
from PIL import Image


print("Cargando datos")
train_data_normal = os.listdir('train/NORMAL')
train_data_neumonia = os.listdir('train/PNEUMONIA')
test_data_normal = os.listdir('test/NORMAL')
test_data_neumonia = os.listdir('test/PNEUMONIA')
val_normal = os.listdir('val/NORMAL')
val_neumonia = os.listdir('val/PNEUMONIA')
train_data = 'train'
```

```
test_data = 'test'
val_data = 'val'
print("Datos cargados correctamente")

x_train = train_data_normal + train_data_neumonia
x_val = val_normal + val_neumonia
x_test = test_data_normal + test_data_neumonia

#asignaremos 1 --> neumonía
#asignaremos 0 --> normal

y_train =
np.concatenate(((np.zeros(len(train_data_normal))),(np.ones(len(train_data_neumonia)
y_val = np.concatenate(((np.zeros(len(val_normal))),np.ones(len(val_neumonia))))
y_test =
np.concatenate(((np.zeros(len(test_data_normal))),np.ones(len(test_data_neumonia))))


#dado que las resoluciones de las imagenes son de diferente tamaño y hay
#algunas de alta resolucion vamos a usar el paquete ImageDataGenerator
epochs = 50
altura, long = 130,130 #escalado del tamaño de las imagenes  130
batch_size = 64 #imagenes por cada iteracion 64

train_gen = ImageDataGenerator(
    rescale=1./255, #Reescalado entre 0-1 para los pixeles
    shear_range=0.3, #inclinacion
    zoom_range=0.3, #zoom
    horizontal_flip=True #giro admitido
    )

val_gen = ImageDataGenerator(
    rescale=1./255 #solo queremos reescalado para la validacion
    )              #para no tener overfitting pero tampoco estropear el
                   #train dde todas las epocas

test_gen = ImageDataGenerator(
    rescale=1./255
    )

image_train = train_gen.flow_from_directory( #sacamos todas las imagenes
    train_data,                              #redimensionadas, de 64 en 64
    target_size = (altura, long),            #categoricamente (2 clases)
    batch_size = batch_size,
    color_mode = "rgb",
    class_mode='categorical'  #se podría poner 'binary'
    )

image_val = val_gen.flow_from_directory(
    val_data,
    target_size=(altura, long),
    batch_size=batch_size,
    color_mode = "rgb",
    class_mode='categorical'
    )

image_test = val_gen.flow_from_directory(
    test_data,
```

```
    target_size=(altura, long),
    batch_size=batch_size,
    color_mode = "rgb",
    class_mode='categorical'
    )

print("Creando el modelo")
#Creación de la parte convolucional
CNN = models.Sequential()
CNN.add(layers.Conv2D(32,kernel_size=(3,3),input_shape = (altura, long,
3), activation='relu'))
CNN.add(layers.MaxPooling2D(pool_size=(2,2)))
CNN.add(layers.Conv2D(64,kernel_size=(3,3), activation='relu'))
CNN.add(layers.MaxPooling2D(pool_size=(2,2)))
CNN.add(layers.Conv2D(128, kernel_size=(3,3), activation='relu'))
CNN.add(layers.MaxPooling2D(pool_size=(2,2)))

#Se aplana para pasarsela a la ANN
CNN.add(layers.Flatten())
#Creacion de la red densa
CNN.add(Dense(128,activation = 'relu'))
CNN.add(layers.Dropout(.8)) #el 80% de las neuronas son apagadas aleatoriamente
CNN.add(Dense(64,activation = 'relu'))
CNN.add(layers.Dropout(.6)) # 60%
CNN.add(Dense(32, activation='relu'))
CNN.add(layers.Dropout(.4)) #40%
CNN.add(Dense(2, activation='Softmax'))

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

history = CNN.fit_generator(image_train,
                  steps_per_epoch= ((len(x_train)//batch_size)//3),
                  epochs = epochs,
                  validation_data = image_test,
                  validation_steps= None )


target_dir = './modelo/'
if not os.path.exists(target_dir):
    os.mkdir(target_dir)
CNN.save('./modelo/modelo.h5')
CNN.save_weights('./modelo/pesos.h5')

loss, accuracy = CNN.evaluate_generator(image_val)
print(f'Despues del entrenamiento mi resultado del test es loss = {loss} y
acurracy = {accuracy}')



plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Accuracy durante el entrenamiento')
plt.xlabel('Epochs')
```

```
plt.ylabel('Accuracy')
plt.legend(['training_accuracy', 'validation_accuracy'])
plt.show()


plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Loss durante el entrenamiento')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend(['training_loss', 'validation_loss'])
plt.show()
```

ANEXO B

Código correspondiente a la arquitectura ViT. Hay que recordar que a este código se puede acceder de forma online mediante el siguiente enlace:

https://colab.research.google.com/drive/1arn2pp6_eMT2sLjsf6EcEfgBk8F2mxO3?usp=sharing

```
# -*- coding: utf-8 -*-
"""
Date created: 2021/01/18
Last modified: 2021/06/11
@author: Khalid Salama
@author: imata
"""
import os
import sys
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import tensorflow_addons as tfa
from keras.preprocessing.image import img_to_array
from keras.preprocessing.image import ImageDataGenerator
from sklearn.utils import shuffle
import matplotlib.pyplot as plt
import random
from keras.preprocessing.image import load_img, img_to_array

"""
## Prepare the data
"""


#input_shape = (32, 32, 3)

#(x_train, y_train), (x_test, y_test) =
keras.datasets.cifar100.load_data()

#print(f"x_train shape: {x_train.shape} - y_train shape: {y_train.shape}")
#print(f"x_test shape: {x_test.shape} - y_test shape: {y_test.shape}")

print("Cargando datos")
train_data_normal = os.listdir('train/NORMAL')
train_data_neumonia = os.listdir('train/PNEUMONIA')
test_data_normal = os.listdir('test/NORMAL')
test_data_neumonia = os.listdir('test/PNEUMONIA')
val_normal = os.listdir('Val/NORMAL')
val_neumonia = os.listdir('val/PNEUMONIA')
train_data = 'train'
test_data = 'test'
val_data = 'val'

```

```
46
47
48 def give_set (path) :
49     x = []
50     y = []
51     for etiqueta in os.listdir(path) :
52         for img in os.listdir(path +'/' + etiqueta + '/') :
53             #contador = 0
54             img_path = path +'/'+ etiqueta + '/' + img
55             image = keras.preprocessing.image.load_img(img_path,
56                                 color_mode = 'rgb', target_size =
57 (130,130))
58             x.append(img_to_array(image))
59             #if etiqueta == 'NORMAL':
60              #    contador += 1
61               #  y.append(contador)
62            # else:
63             #    y.append(contador)
64             y.append(1 if etiqueta == 'NORMAL' else 0)
65
66     x = np.array(x, dtype = float)
67     y = np.array(y)
68
69     return shuffle(x,y)
70
71
72 x_train, y_train = give_set(train_data)
73 #x_train /= 255.0
74 x_test, y_test = give_set(test_data)
75 #x_test /= 255.0
76 x_value, y_value = give_set(val_data)
77 #x_value /= 255.0
78 print(y_train)
79 print("Datos cargados correctamente")
80
81
82 num_classes = 2
83
84 print(f"x_train shape: {x_train.shape} - y_train shape: {y_train.shape}")
85 print(f"x_test shape: {x_test.shape} - y_test shape: {y_test.shape}")
86
87
88 print ('Configure the hyperparameters')
89
90
91 learning_rate = 0.0005
92 weight_decay = 0.001
93 batch_size = 64
94 num_epochs = 50 #cambiado
95 image_size = 130  # We'll resize input images to this size #cambiado
96 patch_size = 16  # Size of the patches to be extract from the input images
97 num_patches = (image_size // patch_size) ** 2
98 projection_dim = 64
99 num_heads = 4
100 transformer_units = [ projection_dim * 2,
101                       projection_dim]  # Size of the transformer layers
102 transformer_layers = 8
103 mlp_head_units = [32, 16, 2]  # Size of the dense layers of the final
```

```
104 classifier #cambiado
105
106
107 """
108 ## Use data augmentation
109 """
110
111 data_augmentation = keras.Sequential(
112     [
113         layers.experimental.preprocessing.Normalization(),
114         layers.experimental.preprocessing.Resizing(image_size,
115 image_size),
116         layers.experimental.preprocessing.RandomFlip("horizontal"),
117         layers.experimental.preprocessing.RandomRotation(factor=0.02),
118         layers.experimental.preprocessing.RandomZoom(
119             height_factor=0.2, width_factor=0.2
120         ),
121     ],
122     name="data_augmentation",
123 )
124 # Compute the mean and the variance of the training data for
125 normalization.
126 data_augmentation.layers[0].adapt(x_train)
127
128
129 """
130 ## Implement multilayer perceptron (MLP)
131 """
132
133
134 def mlp(x, hidden_units, dropout_rate):
135     for units in hidden_units:
136         x = layers.Dense(units, activation=tf.nn.gelu)(x)
137         x = layers.Dropout(dropout_rate)(x)
138     return x
139
140
141 """
142 ## Implement patch creation as a layer
143 """
144
145 class Patches(layers.Layer):
146     def __init__(self, patch_size):
147         super(Patches, self).__init__()
148         self.patch_size = patch_size
149
150     def call(self, images):
151         batch_size = tf.shape(images)[0]
152         patches = tf.image.extract_patches(
153             images=images,
154             sizes=[1, self.patch_size, self.patch_size, 1],
155             strides=[1, self.patch_size, self.patch_size, 1],
156             rates=[1, 1, 1, 1],
157             padding="VALID",)
158         patch_dims = patches.shape[-1]
159         patches = tf.reshape(patches, [batch_size, -1, patch_dims])
```

```
        return patches


plt.figure(figsize=(8, 8))
image = x_train[np.random.choice(range(x_train.shape[0]))]
plt.imshow(image.astype("uint8"))
plt.axis("off")

resized_image = tf.image.resize(
    tf.convert_to_tensor([image]), size=(image_size, image_size))

patches = Patches(patch_size)(resized_image)
print(f"Image size: {image_size} X {image_size}")
print(f"Patch size: {patch_size} X {patch_size}")
print(f"Patches per image: {patches.shape[1]}")
print(f"Elements per patch: {patches.shape[-1]}")

n = int(np.sqrt(patches.shape[1]))
plt.figure(figsize=(8, 8)) #4, 4
for i, patch in enumerate(patches[0]):
    ax = plt.subplot(n, n, i + 1)
    patch_img = tf.reshape(patch, (patch_size, patch_size, 3))
    plt.imshow(patch_img.numpy().astype("uint8"))
    plt.axis("off")


print ('Creando particiones')
class PatchEncoder(layers.Layer):
    def __init__(self, num_patches, projection_dim):
        super(PatchEncoder, self).__init__()
        self.num_patches = num_patches
        self.projection = layers.Dense(units=projection_dim)
        self.position_embedding = layers.Embedding(
            input_dim=num_patches, output_dim=projection_dim
        )

    def call(self, patch):
        positions = tf.range(start=0, limit=self.num_patches, delta=1)
        encoded = self.projection(patch) +
self.position_embedding(positions)
        return encoded

print ('Particiones creadas')


def create_vit_classifier():
    inputs = layers.Input(shape= (130, 130, 3))
    # Augment data.
    augmented = data_augmentation(inputs)
    # Create patches.
    patches = Patches(patch_size)(augmented)
    # Encode patches.
    encoded_patches = PatchEncoder(num_patches, projection_dim)(patches)

    # Create multiple layers of the Transformer block.
    for _ in range(transformer_layers):
        # Layer normalization 1.
        x1 = layers.LayerNormalization(epsilon=1e-6)(encoded_patches)
```

```
218         # Create a multi-head attention layer.
219         attention_output = layers.MultiHeadAttention(
220         num_heads=num_heads, key_dim=projection_dim, dropout=0.1)(x1, x1)
221 
222         # Skip connection 1.
223         x2 = layers.Add()([attention_output, encoded_patches])
224         # Layer normalization 2.
225         x3 = layers.LayerNormalization(epsilon=1e-6)(x2)
226         # MLP.
227         x3 = mlp(x3, hidden_units=transformer_units, dropout_rate=0.1)
228         # Skip connection 2.
229         encoded_patches = layers.Add()([x3, x2])
230 
231     # Create a [batch_size, projection_dim] tensor.
232     representation = layers.LayerNormalization
233 (epsilon=1e-6)(encoded_patches)
234     representation = layers.Flatten()(representation)
235     representation = layers.Dropout(0.5)(representation)
236     # Add MLP.
237     features = mlp(representation, hidden_units=mlp_head_units,
238 dropout_rate=0.5)
239     # Classify outputs.
240     logits = layers.Dense(num_classes)(features)
241     # Create the Keras model.
242     model = keras.Model(inputs=inputs, outputs=logits)
243     return model
244 
245 
246 """
247 ## Compile, train, and evaluate the mode
248 """
249 
250 def run_experiment(model):
251     optimizer = tfa.optimizers.AdamW(
252         learning_rate=learning_rate, weight_decay=weight_decay
253     )
254 
255     model.compile(
256         optimizer=optimizer,
257         loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
258         metrics=[
259             keras.metrics.SparseCategoricalAccuracy(name="accuracy"),
260             keras.metrics.SparseTopKCategoricalAccuracy(5, name="top-5-
261 accuracy")])
262 
263     checkpoint_filepath = "/tmp/checkpoint"
264     checkpoint_callback = keras.callbacks.ModelCheckpoint(
265         checkpoint_filepath,
266         monitor="val_accuracy",
267         save_best_only=True,
268         save_weights_only=True,
269     )
270 
271     history = model.fit(
272         x=x_train,
273         y=y_train,
```

```
274         batch_size=batch_size,
275         epochs=num_epochs,
276         validation_data= (x_value, y_value),
277         callbacks=[checkpoint_callback],
278     )

279     model.load_weights(checkpoint_filepath)
280     _, accuracy, top_5_accuracy = model.evaluate(x_test, y_test)
281     print(f"Test accuracy: {round(accuracy * 100, 2)}%")
282     print(f"Test top 5 accuracy: {round(top_5_accuracy * 100, 2)}%")
283
284     return history
285
286 vit_classifier = create_vit_classifier()
287 history = run_experiment(vit_classifier)
```

ANEXO C

Parte del código correspondiente al código para la comprobación visual de los resultados. Este código se puede encontrar al final de los dos ficheros anteriores en su formato online, tanto en el Anexo 1 y Anexo 2.

```
"""
Created on Tue May 18 07:44:51 2021

@author: imata
"""
import numpy as np
import tensorflow as tf
from keras.preprocessing.image import load_img, img_to_array
from keras.models import load_model
import matplotlib as plt
import os
import sys
import numpy as np
import random
import pandas as pd
import keras
import tensorflow as tf
import sklearn
import matplotlib.pyplot as plt
from keras import layers
from keras import models
from keras.utils.vis_utils import plot_model
from keras.utils.vis_utils import plot_model
from keras.utils.vis_utils import model_to_dot
from keras.preprocessing.image import load_img
from keras.layers import Dropout, Flatten, Dense, Activation
from keras.preprocessing.image import ImageDataGenerator
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import normalize
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from tqdm import tqdm
from PIL import Image

long, altura = 130, 130
modelo ='./modelo4/modelo.h5'
pesos = './modelo4/pesos.h5'

train_data_normal = os.listdir('train/NORMAL')
train_data_neumonia = os.listdir('train/PNEUMONIA')
test_data_normal = os.listdir('test/NORMAL')
test_data_neumonia = os.listdir('test/PNEUMONIA')
val_normal = os.listdir('val/NORMAL')
```

```
val_neumonia = os.listdir('val/PNEUMONIA')

CNN = load_model(modelo)
CNN.load_weights(pesos)

def predictor(archivo):
    # image1 = tf.keras.preprocessing.image.load_img(archivo,
target_size=(altura, long))
    # image = img_to_array(image1)
     image =
np.expand_dims(img_to_array(tf.keras.preprocessing.image.load_img(archivo,
target_size=(altura, long))), axis = 0)
     solucion = CNN.predict(image)
     resultado = solucion[0]
     indice = np.argmax(resultado)
     print(solucion)
     if indice == 0:
         etiqueta = "Normal"
         print("Normal")
     elif indice == 1:
         etiqueta = "Neumonía"
         print("Neumonía")
     return etiqueta


img_normal = 'test/NORMAL/' + test_data_normal[random.randrange(0,
len(test_data_normal))]
img_neumonia = 'test/PNEUMONIA/' + test_data_neumonia[random.randrange(0,
len(test_data_neumonia))]
img_normal1 = load_img('test/NORMAL/' +
test_data_normal[random.randrange(0, len(test_data_normal))])
img_neumonia1 = load_img('test/PNEUMONIA/' +
test_data_neumonia[random.randrange(0, len(test_data_neumonia))])
etiqueta1 = predictor(img_normal)
etiqueta =  predictor(img_neumonia)

fig, axs = plt.subplots(1,2,figsize=(10,6))
axs[0].imshow(img_normal1)
axs[0].set_title(etiqueta1)
axs[1].imshow(img_neumonia1)
axs[1].set_title(etiqueta)
```