Trabajo de Fin de Máster

Máster Universitario en Ciberseguridad e Inteligencia de Datos

Clasificación de imágenes pulmonares con redes neuronales convolucionales

Classification of Chest X-Rays using Convolutional Neural Networks

Ainhoa Aranda Elvira

La Laguna, 20 de mayo de 2024

D. **Carlos Pérez González**, con N.I.F. 45452719G, Profesor Titular de Universidad adscrito al Departamento de Matemáticas, Estadística e Investigación Operativa de la Universidad de La Laguna, como tutor.

C E R T I F I C A

Que la presente memoria titulada:

”*Clasificación de imágenes pulmonares con redes neuronales convolucionales*”

ha sido realizada bajo su dirección por D. **Ainhoa Aranda Elvira**, con N.I.F. 75576830A.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 20 de mayo de 2024

Agradecimientos

A mis amigos y familia, que siempre están ahí.

Licencia

Resumen

El objetivo de este Trabajo de Fin de Máster (TFM) es desarrollar una herramienta que mejore la precisión y rapidez en el diagnóstico de condiciones médicas mediante el uso de imágenes biomédicas, específicamente rayos X de tórax. El enfoque principal es utilizar técnicas de aprendizaje profundo, concretamente redes neuronales convolucionales (CNN), aprovechando el método de transfer learning. Esta herramienta se centrará en la clasificación de imágenes de rayos X en tres categorías: COVID-19, neumonía y normal. Al implementar modelos de CNN preentrenados como VGG16 y LeNet-5, y utilizando la biblioteca PyTorch Lightning para facilitar el entrenamiento y la gestión de experimentos, se busca proporcionar una solución eficaz que pueda ser integrada en entornos clínicos para ayudar a los profesionales de la salud a tomar decisiones más informadas y rápidas.

Palabras clave: Clasificación de imágenes, redes neuronales convolucionales, Python, Pytorch Lightning

Abstract

The objective of this Master's Thesis (TFM) is to develop a tool that enhances the accuracy and speed of diagnosing medical conditions using biomedical images, specifically chest X-rays. The main focus is to use deep learning techniques, particularly convolutional neural networks (CNN), leveraging the method of transfer learning. This tool will focus on classifying X-ray images into three categories: COVID-19, pneumonia, and normal. By implementing pre-trained CNN models such as VGG16 and LeNet-5, and using the PyTorch Lightning library to facilitate training and experiment management, the goal is to provide an effective solution that can be integrated into clinical settings to help healthcare professionals make more informed and rapid decisions.

Keywords: Image classification, convolutional neural networks, Python, Pytorch Lightning

Índice general

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

El objetivo de este Trabajo de Fin de Máster (TFM) es desarrollar una herramienta que mejore la precisión y rapidez en el diagnóstico de condiciones médicas mediante el uso de imágenes biomédicas, específicamente rayos X de tórax. El enfoque principal es utilizar técnicas de aprendizaje profundo, concretamente redes neuronales convolucionales (CNN), aprovechando el método de transfer learning. Esta herramienta se centrará en la clasificación de imágenes de rayos X en tres categorías: COVID-19, neumonía y normal. Al implementar modelos de CNN preentrenados como VGG16 y LeNet-5, y utilizando la biblioteca PyTorch Lightning para facilitar el entrenamiento y la gestión de experimentos, se busca proporcionar una solución eficaz que pueda ser integrada en entornos clínicos para ayudar a los profesionales de la salud a tomar decisiones más informadas y rápidas.

1.1. Contexto y motivación

El diagnóstico temprano y preciso de enfermedades pulmonares como el COVID-19 y la neumonía es crucial para el tratamiento efectivo y la gestión de pacientes. La pandemia de COVID-19 ha resaltado la necesidad urgente de herramientas que puedan asistir a los médicos en la identificación rápida y precisa de infecciones pulmonares mediante el análisis de imágenes de rayos X.

- Según la Organización Mundial de la Salud (OMS), las enfermedades respiratorias como la neumonía son una de las principales causas de muerte a nivel mundial, especialmente en niños menores de cinco años y en personas mayores.
- La pandemia de COVID-19 ha causado millones de infecciones y muertes, subrayando la importancia de una detección rápida y precisa para controlar la propagación del virus y tratar a los pacientes adecuadamente.
- Estudios han mostrado que la precisión de los diagnósticos de rayos X realizados por radiólogos puede variar significativamente, lo que resalta la necesidad de herramientas automatizadas que puedan asistir en el diagnóstico.
- Un diagnóstico tardío puede llevar a complicaciones graves y un aumento en la mortalidad. En el caso de la neumonía, un tratamiento tardío puede resultar en sepsis y fallo multiorgánico.
- En el contexto de COVID-19, un diagnóstico tardío no solo afecta al paciente individual, sino que también contribuye a la propagación del virus, afectando a la comunidad en general.

La herramienta propuesta pretende mitigar estos problemas al proporcionar un sistema automatizado y preciso para la clasificación de imágenes de rayos X de tórax, mejorando así el proceso de diagnóstico y tratamiento.

1.1.1. Estado del arte

El uso de CNNs en el diagnóstico médico ha sido ampliamente estudiado y aplicado en los últimos años. Las CNNs han demostrado ser altamente efectivas en tareas de clasificación de imágenes debido a su capacidad para aprender características jerárquicas a partir de los datos de entrada.

1.1.2. Modelos y métodos utilizados

- **VGG16**: Este modelo, desarrollado por el Visual Geometry Group de la Universidad de Oxford, ha sido ampliamente utilizado en tareas de clasificación de imágenes debido a su arquitectura profunda y su capacidad para aprender representaciones ricas de las imágenes. VGG16 ha demostrado una alta precisión en la clasificación de imágenes médicas.
- **LeNet-5**: Este modelo más simple, desarrollado por Yann LeCun, es conocido por su eficacia en tareas de reconocimiento de dígitos manuscritos y ha sido adaptado para tareas de clasificación de imágenes biomédicas debido a su simplicidad y eficiencia computacional.

1.1.3. Transfer learning

El transfer learning se ha convertido en una técnica fundamental en el aprendizaje profundo, especialmente cuando se dispone de conjuntos de datos limitados. Al utilizar modelos preentrenados en grandes conjuntos de datos (e.g., ImageNet), es posible adaptar estos modelos a nuevas tareas con una cantidad menor de datos etiquetados.

Estudios recientes han demostrado que el transfer learning puede mejorar significativamente la precisión de los modelos en tareas de diagnóstico médico, reduciendo el tiempo de entrenamiento y los recursos computacionales necesarios.

1.1.4. Comparación de métodos

- **VGG16 vs. LeNet-5**: Mientras que VGG16 ofrece una profundidad y una capacidad de representación superiores, LeNet-5 es más ligero y rápido de entrenar. La elección del modelo depende del equilibrio entre la precisión deseada y los recursos computacionales disponibles.

1.2. Implementación con PyTorch Lightning

PyTorch Lightning es una biblioteca que facilita la organización y gestión de experimentos de aprendizaje profundo. Permite una implementación más estructurada y reproducible de modelos, lo cual es crucial para la investigación y desarrollo de modelos de CNN.

Capítulo 2

Orígenes y conceptos básicos de las redes neuronales

La Inteligencia Artificial (IA), y dentro de ella, el Aprendizaje Profundo (DL por sus siglas en inglés), es un subconjunto del Aprendizaje Automático (ML), inspirado en los patrones de procesamiento de información encontrados en el cerebro humano. A diferencia de los métodos tradicionales de ML, DL no requiere reglas diseñadas por humanos para operar; en cambio, utiliza una gran cantidad de datos para mapear la entrada dada a etiquetas específicas. DL está diseñado utilizando numerosas capas de algoritmos (redes neuronales artificiales, o ANNs por sus siglas en inglés), cada una de las cuales proporciona una interpretación diferente de los datos que se les ha alimentado.

Lograr la tarea de clasificación utilizando técnicas de ML convencionales requiere varios pasos secuenciales, específicamente preprocesamiento, extracción de características, selección de características sabias, aprendizaje y clasificación. Además, la selección de características tiene un gran impacto en el rendimiento de las técnicas de ML. Una selección de características sesgada puede llevar a una discriminación incorrecta entre clases. Por el contrario, DL tiene la capacidad de automatizar el aprendizaje de conjuntos de características para varias tareas, a diferencia de los métodos de ML convencionales. DL permite que el aprendizaje y la clasificación se logren en una sola pasada. En los últimos años, DL se ha convertido en un tipo de algoritmo de ML increíblemente popular debido al enorme crecimiento y evolución del campo de los grandes datos.

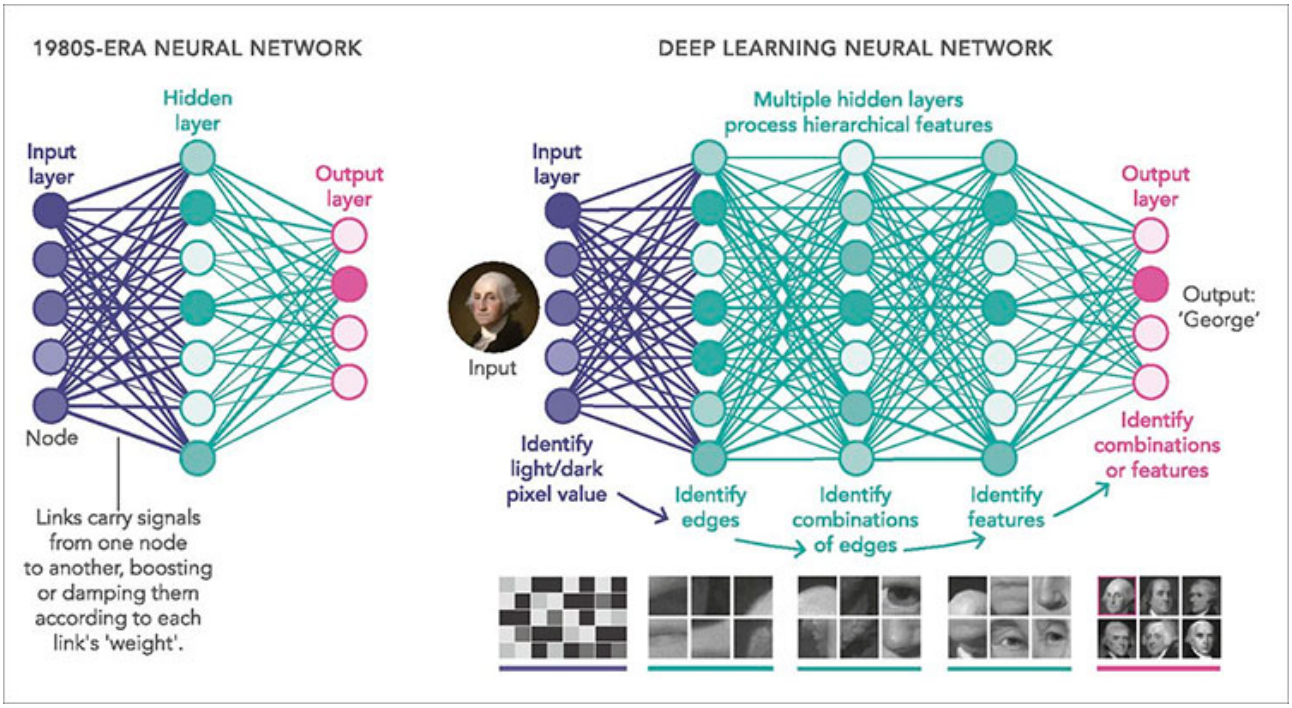


Figura 2.1: Estructura de la evolución de las redes neuronales. Fuente: Google Imágenes.

2.1. Orígenes Históricos

La historia del aprendizaje automático se remonta al 300 a.C., con Aristóteles, quien propuso el asociacionismo como una teoría que explica la mente como un conjunto de elementos conceptuales organizados en asociaciones. El asociacionismo se basa en cuatro leyes: contigüidad, frecuencia, similitud y contraste. Estas leyes aún sirven como fundamentos de los métodos de aprendizaje automático en la actualidad. Filósofos posteriores, como Hobbes, Locke, Hume y Stewart, ampliaron y consolidaron esta teoría.

Aunque la IA como campo de estudio no existía en la antigüedad, Aristóteles sentó algunas de las bases teóricas de la lógica y el razonamiento que más tarde influirían en el desarrollo de la IA. Su obra sobre silogismos y lógica formal proporcionó las primeras herramientas para la sistematización del pensamiento. En los siglos XVII y XVIII, filósofos y matemáticos como René Descartes y Gottfried Wilhelm Leibniz desarrollaron ideas sobre la mecanización del pensamiento. Leibniz, en particular, imaginó un lenguaje universal de razonamiento, un precursor conceptual de los lenguajes formales utilizados en la IA moderna.

En el siglo XIX y principios del XX, Ada Lovelace y Charles Babbage trabajaron en la creación de la Máquina Analítica, un diseño temprano de un ordenador que podría realizar cálculos complejos. Lovelace especuló sobre la capacidad de las máquinas para seguir algoritmos y posiblemente crear música, ideas que anticipan aspectos de la IA. La evolución del aprendizaje profundo comenzó en la década de 1940 cuando Warren McCulloch y Walter Pitts propusieron el concepto de neuronas artificiales. Desarrollaron un modelo matemático basado en el funcionamiento de las neuronas biológicas básicas, dando origen a la primera neurona artificial llamada McCulloch-Pitts (MCP-neurona), sentando así las bases para el aprendizaje profundo.

En 1957, Frank Rosenblatt introdujo el perceptrón, un algoritmo basado en neuronas artificiales capaz de aprender de la experiencia y ajustar sus pesos para realizar predicciones precisas. Sin embargo, los perceptrones estaban limitados a resolver problemas linealmente separables, lo que llevó a una disminución del interés en las redes neuronales. Después del fracaso del perceptrón, se llevó a cabo mucha investigación y finalmente, en 1986, Geoffrey Hinton, junto con David Rumelhart y Ronald Williams, hicieron un gran avance al introducir el algoritmo de retropropagación. Esto permitió el entrenamiento de redes neuronales multicapa al calcular eficientemente los gradientes de los pesos de la red, reavivando el interés en las redes neuronales y allanando el camino para el desarrollo del aprendizaje profundo.

El punto de partida formal de la IA como campo de estudio se considera la Dartmouth Conference en 1956. John McCarthy, Marvin Minsky, Nathaniel Rochester y Claude Shannon organizaron esta conferencia, donde acuñaron el término “inteligencia artificial” y propusieron que “cada aspecto del aprendizaje o cualquier otra característica de la inteligencia puede ser descrito de tal manera que una máquina puede ser construida para simularlo”.

2.1.1. Evolución de la IA

Desde la década de 1950 hasta la de 1970, se desarrollaron los primeros programas de IA, como el Logic Theorist de Allen Newell y Herbert A. Simon, y el ELIZA de Joseph Weizenbaum. En este período también surgió el concepto de redes neuronales artificiales con los primeros modelos como el perceptrón de Frank Rosenblatt. Durante la década de 1980, la IA se expandió con el desarrollo de sistemas expertos, programas que imitan el conocimiento y el proceso de toma de decisiones de expertos humanos en campos específicos. Además, se introdujo el concepto de redes neuronales multicapa y el algoritmo de retropropagación de errores.

Entre las décadas de 1990 y 2000, se produjeron avances significativos en el aprendizaje automático y la minería de datos. Algoritmos como las máquinas de soporte vectorial (SVM) y los bosques aleatorios (random forests) se volvieron populares. La capacidad de procesamiento de datos y el almacenamiento mejoraron, permitiendo la manipulación de grandes conjuntos de datos. Desde 2010 en adelante, la IA experimentó un renacimiento con la explosión de datos (big data), mejoras en el hardware de computación (GPUs) y avances en algoritmos de aprendizaje profundo (deep learning). Empresas tecnológicas y académicas desarrollaron modelos avanzados como las redes neuronales convolucionales (CNNs) y redes neuronales recurrentes (RNNs), aplicándolas en reconocimiento de imágenes, procesamiento de lenguaje natural y más.

2.2. Tipos de Redes Neuronales

Las redes neuronales son una herramienta fundamental en el campo de la inteligencia artificial, y existen diversos tipos que se adaptan a diferentes tipos de problemas y datos. A continuación, se presentan algunos de los tipos más importantes de redes neuronales y sus características distintivas.

El **perceptrón** es la unidad más básica de una red neuronal, introducida por Frank Rosenblatt en 1958. Consiste en una neurona artificial con una sola capa que puede resolver problemas de clasificación lineal. Su funcionamiento se basa en el ajuste de pesos para minimizar el error en las predicciones, permitiendo así aprender de los datos de entrenamiento. Sin embargo, el perceptrón está limitado a problemas que son linealmente separables.

Las **redes neuronales multicapa** (MLP, por sus siglas en inglés) superan esta limitación al consistir en una capa de entrada, una o más capas ocultas y una capa de salida. Estas redes utilizan la retropropagación para ajustar los pesos durante el entrenamiento, lo que les permite aprender representaciones más complejas de los datos. La inclusión de múltiples capas ocultas permite a las MLPs modelar relaciones no lineales entre las variables de entrada y salida.

Las **redes neuronales convolucionales** (CNNs, por sus siglas en inglés) están especializadas en el procesamiento de datos con estructura de cuadrícula, como imágenes. Utilizan operaciones convolucionales para extraer características espaciales jerárquicas, lo que las hace particularmente efectivas para tareas de visión por computadora. Las operaciones convolucionales involucran el uso de filtros (kernels) que se desplazan sobre la imagen de entrada para detectar patrones locales. Estas redes también utilizan funciones de activación, como ReLU (Rectified Linear Unit), para introducir no linealidades en el modelo, lo que mejora su capacidad de aprendizaje.

Las **redes neuronales recurrentes** (RNNs, por sus siglas en inglés) están diseñadas para procesar datos secuenciales. A diferencia de las redes neuronales tradicionales, las RNNs mantienen un estado interno que captura la dependencia temporal entre los elementos de la secuencia, lo que las hace útiles para tareas como el procesamiento de lenguaje natural y el reconocimiento de voz. Sin embargo, las RNNs convencionales pueden tener dificultades para aprender dependencias a largo plazo en secuencias de datos.

Para abordar estas limitaciones, se desarrollaron las **redes neuronales de memoria a largo plazo** (LSTM, por sus siglas en inglés). Las LSTM son una variante de las RNNs que introducen mecanismos de memoria para manejar dependencias a largo plazo. Estos mecanismos permiten a las LSTM almacenar y recuperar información relevante durante periodos prolongados, mejorando significativamente su rendimiento en tareas secuenciales complejas.

Finalmente, las **redes generativas adversariales** (GANs, por sus siglas en inglés) representan un enfoque innovador en el campo de las redes neuronales. Las GANs consisten en dos redes neuronales, un generador y un discriminador, que compiten entre sí. El generador intenta producir datos sintéticos que imitan la distribución de los datos reales, mientras que el discriminador intenta distinguir entre datos reales y sintéticos. Este proceso competitivo lleva a la mejora continua de ambos modelos, resultando en la generación de datos sintéticos de alta calidad que pueden ser utilizados en diversas aplicaciones, desde la creación de imágenes hasta la síntesis de datos para entrenamiento de modelos.

Cada uno de estos tipos de redes neuronales tiene sus propias ventajas y limitaciones, y la elección del tipo adecuado depende de la naturaleza del problema y los datos disponibles. En conjunto, estos avances han expandido enormemente las capacidades de la inteligencia artificial, permitiendo resolver problemas complejos que antes eran inaccesibles.

Capítulo 3

Redes Neuronales Convolucionales (CNN)

A finales de la década de 1990, principalmente debido al trabajo innovador de Yann LeCun, se desarrolló la arquitectura LeNet-5, que utilizaba capas convolucionales, capas de agrupación y capas completamente conectadas para lograr un rendimiento notable en tareas de reconocimiento de dígitos escritos a mano.

Las ConvNets, o redes neuronales convolucionales, están diseñadas para procesar datos que se presentan en forma de múltiples matrices, como una imagen a color compuesta por tres matrices 2D que contienen las intensidades de píxeles en los tres canales de color. Muchas modalidades de datos tienen esta estructura: 1D para señales y secuencias, como el lenguaje; 2D para imágenes o espectrogramas de audio; y 3D para video o imágenes volumétricas. Hay cuatro ideas clave detrás de las ConvNets que aprovechan las propiedades de las señales naturales: conexiones locales, pesos compartidos, agrupación y el uso de muchas capas.

La arquitectura típica de una ConvNet se estructura como una serie de etapas. Las primeras etapas están compuestas por dos tipos de capas: capas convolucionales y capas de agrupación. Aunque el papel de la capa convolucional es detectar conjunciones locales de características de la capa anterior, el papel de la capa de agrupación es fusionar características semánticamente similares en una sola.

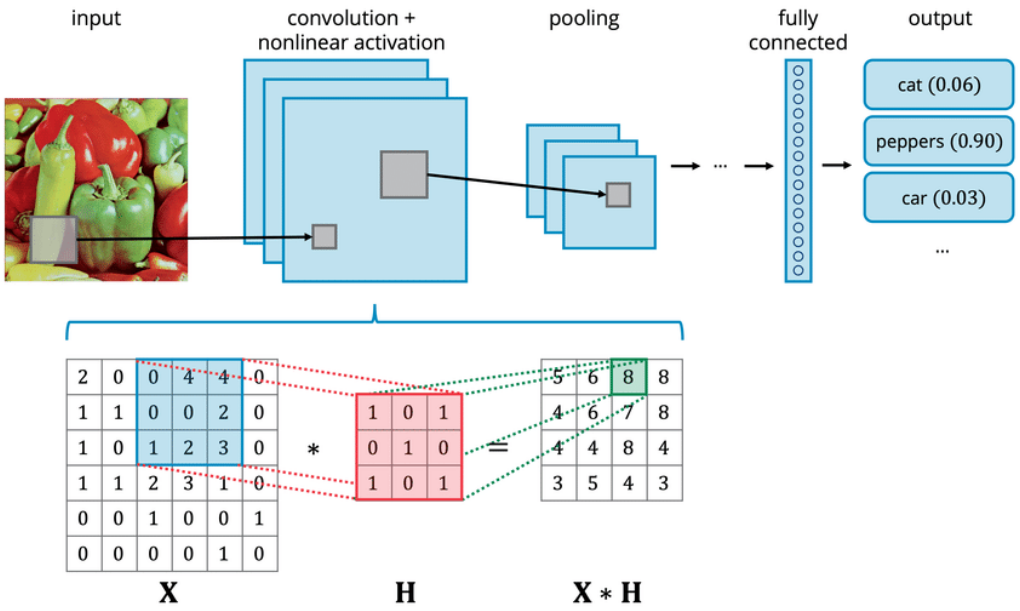


Figura 3.1: Estructura de una red neuronal convolucional (CNN). Fuente: Google Imágenes.

3.1. Conceptos matemáticos utilizados por las CNNs

A continuación, se presentan una serie de conceptos matemáticos sobre los que se apoyan las redes neuronales convolucionales.

3.1.1. Operación de Convolución

En el contexto de imágenes, la convolución implica desplazar una pequeña ventana (filtro o kernel) sobre la imagen de entrada y computar la suma ponderada de los elementos de la imagen que están cubiertos por el filtro en cada posición.

Notación y Definición Matemáticamente, la convolución 2D de una imagen I con un kernel K se define como:

$$(I * K)(i, j) = \sum_m \sum_n I(i - m, j - n) \cdot K(m, n)$$

En esta expresión:

- I representa la imagen de entrada.
- K representa el kernel o filtro.
- i y j son las coordenadas espaciales en la imagen de salida.
- m y n son las coordenadas en el kernel.

Figura 3.2: Esquema del proceso de convolución. Fuente: Medium.

Proceso de la Convolución

- **Desplazamiento del Kernel**: El kernel K se desplaza sobre la imagen de entrada I. En cada posición del kernel, se toma una región de la imagen que tiene el mismo tamaño que el kernel.
- **Multiplicación Elemento a Elemento**: Se multiplican los valores correspondientes de la imagen y el kernel. Es decir, cada elemento del kernel se multiplica por el elemento de la imagen que está cubierto por el kernel en esa posición.
- **Suma de Productos**: Se suman todos los productos obtenidos en el paso anterior. Este valor resultante se convierte en el valor del píxel en la posición (i, j) de la imagen de salida.

3.1.2. Función de Activación

Después de cada operación de convolución, se aplica una función de activación no lineal.

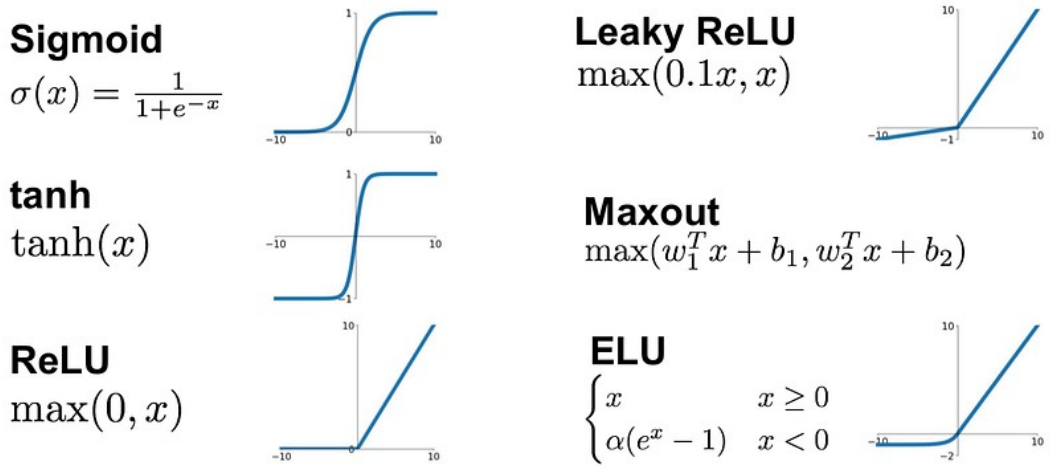


Figura 3.3: Esquema de las diferentes funciones de activación. Fuente: Medium.

Una de las más comunes es la ReLU (Rectified Linear Unit), que introduce no linealidad en la red, permitiendo que pueda aprender funciones más complejas.

La función ReLU se define matemáticamente como:

$$\text{ReLU}(x) = \text{máx}(0, x)$$

Esto significa que todos los valores negativos se convierten en cero, mientras que los valores positivos permanecen iguales.

3.1.3. Agrupación (Pooling)

La capa de agrupación, o *pooling*, se utiliza para reducir la dimensionalidad espacial de la representación, disminuyendo así el número de parámetros y la carga computacional de la red.

Una de las técnicas más comunes es el *Max Pooling*. Esta técnica divide la imagen en regiones no superpuestas y toma el valor máximo de cada región. Esto preserva las características más importantes y proporciona invarianza a pequeñas traslaciones en la imagen.

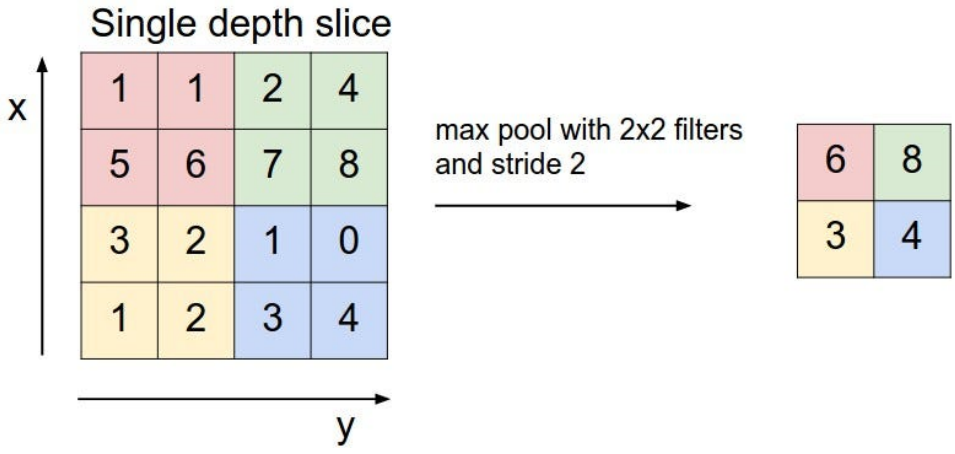


Figura 3.4: Esquema de MaxPooling. Fuente: Medium.

Matemáticamente, si X es la entrada a la capa de pooling y Y es la salida, la operación de Max Pooling se define como:

$$Y(i,j) = \max_{m,n} X(i \cdot s + m, j \cdot s + n)$$

donde s es el tamaño del kernel de pooling.

3.2. Capas más usuales

Tras revisar los conceptos matemáticos, vamos a pasar a hablar acerca de las capas más usuales con las que cuentas las redes neuronales convolucionales.

3.2.1. Conv2d (Capa de Convolución 2D)

Esta capa aplica operaciones de convolución en la entrada usando filtros que se aprenden durante el entrenamiento. Los parámetros importantes incluyen:

- Número de canales de entrada y salida.
- Tamaño del kernel.
- Paso (*stride*) del desplazamiento del kernel.

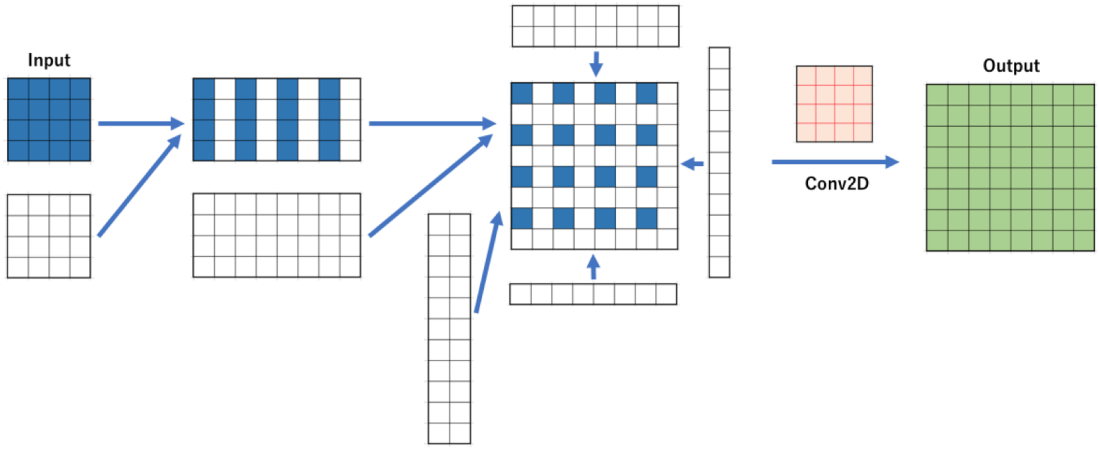


Figura 3.5: Esquema de la capa Conv2d. Fuente: GitHub.

La operación de convolución se realiza mediante la multiplicación del kernel con la porción correspondiente de la entrada y luego sumando los resultados.

3.2.2. MaxPool2d (Capa de Max Pooling 2D)

Esta capa se usa comúnmente después de las capas de convolución. Su función es reducir la dimensionalidad espacial tomando el valor máximo dentro de regiones definidas (por ejemplo, 2×2) y descartando el resto. Esto ayuda a:

- Disminuir la cantidad de parámetros y computaciones en la red.
- Proporcionar invarianza a pequeñas traslaciones y deformaciones en las características detectadas.

3.3. Algoritmos de optimización y función de pérdida

Los algoritmos de entrenamiento y optimización son esenciales en el aprendizaje profundo, ya que guían el proceso de ajuste de los parámetros de la red neuronal para minimizar la función de pérdida y mejorar el rendimiento del modelo.

Entre los algoritmos de optimización más destacados se encuentra **Adam (Adaptive Moment Estimation**), una combinación ingeniosa de momentum y RMSProp. Adam es especialmente poderoso porque adapta las tasas de aprendizaje para cada parámetro en función de la magnitud de los gradientes y de los momentos de primer y segundo orden. Esto permite que el modelo se ajuste de manera más rápida y precisa, adaptándose dinámicamente a la topología del espacio de búsqueda de la función de pérdida. Además, Adam es conocido por su eficiencia y capacidad para evitar los mínimos locales, lo que lo convierte en una elección popular para el entrenamiento de redes neuronales.

La elección de la función de pérdida es crucial, ya que determina cómo el modelo evalúa su propio rendimiento durante el entrenamiento. En la clasificación, la función de pérdida más comúnmente utilizada es la entropía cruzada. Esta función mide la discrepancia entre la distribución de probabilidad predicha por el modelo y la distribución real de las etiquetas. La entropía cruzada es preferida porque penaliza de manera efectiva las predicciones incorrectas con una alta confianza, lo que impulsa al modelo a mejorar su precisión y confianza en sus predicciones.

3.4. Técnicas de Mejora del Modelo

Para mejorar el rendimiento y la generalización de los modelos de CNN, se han empleado diversas estrategias y técnicas durante el entrenamiento.

El aumento de datos, que es una técnica fundamental que busca enriquecer el conjunto de entrenamiento generando nuevas muestras a partir de las existentes mediante transformaciones aleatorias. Estas transformaciones incluyen rotaciones, traslaciones, reflejos, cambios de contraste y zoom. Al aumentar la diversidad de los datos de entrenamiento, se reduce el riesgo de sobreajuste y se mejora la capacidad del modelo para generalizar a datos no vistos. Además, el aumento de datos puede ayudar a mitigar problemas de desequilibrio de clases y mejorar la robustez del modelo ante variaciones en los datos de entrada.

La regularización es una técnica diseñada para reducir el sobreajuste al penalizar modelos más complejos. Una de las técnicas de regularización más efectivas en las CNN es el dropout. Esta técnica consiste en aleatoriamente desactivar una fracción de las unidades (neuronas) de la red durante el entrenamiento, lo que previene la coadaptación de las unidades y obliga al modelo a aprender características más robustas y distribuidas. El dropout es especialmente útil en redes profundas para prevenir el sobreajuste y mejorar la generalización del modelo.

Capítulo 4

Estudio de un caso práctico: clasificación de radiografías de tórax

El dataset utilizado constituye una colección integral de imágenes de radiografías de tórax, meticulosamente organizadas en tres categorías principales: covid, pneumonía y normal. Cada categoría refleja distintas condiciones clínicas observadas en los pacientes, proporcionando así un amplio espectro de casos para el análisis y la investigación.

En total, el dataset consta de 6939 muestras, distribuidas equitativamente entre las categorías mencionadas. Con 2313 muestras dedicadas a cada caso, se garantiza un equilibrio representativo en la exploración de las diversas condiciones médicas.

4.1. Proceso de transformación y carga de los datos (ETL)

Esta clase está diseñada para crear un conjunto de datos personalizado para usarlo posteriormente en el entrenamiento de los modelos de multiclasificación.

El propósito principal de esta clase es cargar imágenes de un directorio y sus subdirectorios, donde cada subdirectorio representa una clase diferente. Estas imágenes se utilizan para crear un conjunto de datos que PyTorch puede entender y procesar.

Al inicializar CustomDataset, se especifica la ruta al directorio raíz donde residen las imágenes. Además, se identifican las clases del conjunto de datos explorando los subdirectorios del directorio raíz. Durante la inicialización, se recorre cada clase y se carga cada imagen junto con su etiqueta (nombre de la clase). Estos pares de imagen-etiqueta se almacenan en una lista llamada data. Finalmente, la imagen transformada y su etiqueta se devuelven como una tupla.

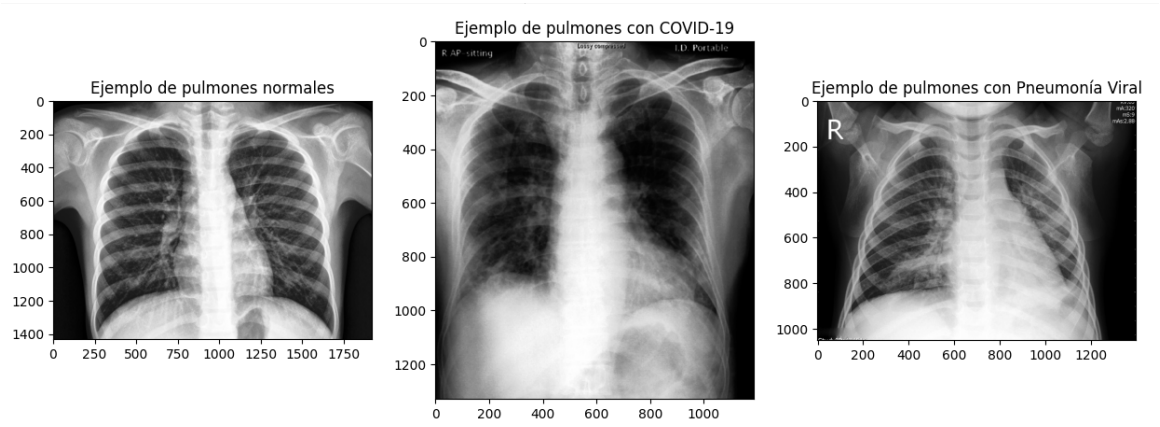


Figura 4.1: Ejemplos de los tres tipos de imágenes del Dataset

4.2. La librería Pytorch Lightning

PyTorch Lightning es una librería de código abierto que proporciona una abstracción sobre PyTorch para simplificar el proceso de entrenamiento y desarrollo de modelos de aprendizaje profundo. Está diseñada para mejorar la legibilidad, modularidad y escalabilidad del código, al mismo tiempo que ofrece una serie de funcionalidades adicionales para facilitar tareas comunes en el entrenamiento de modelos.

4.2.1. Características principales

La librería PyTorch Lightning destaca por varias características clave que facilitan y optimizan el proceso de entrenamiento de modelos. Entre las más importantes se encuentran:

- **Abstracción del ciclo de entrenamiento**: PyTorch Lightning maneja automáticamente el ciclo de entrenamiento, incluyendo la iteración sobre lotes de datos, el cálculo de gradientes, la actualización de parámetros y el registro de métricas, lo que permite que el código de entrenamiento sea más limpio y legible.
- **Módulos reutilizables**: Permite definir modelos, optimizadores y esquemas de entrenamiento en forma de módulos reutilizables y configurables, lo que facilita la experimentación con diferentes arquitecturas de modelos y estrategias de entrenamiento.
- **Escalabilidad**: PyTorch Lightning está diseñado para admitir fácilmente la distribución de entrenamientos en paralelo en múltiples GPUs o incluso en clusters de GPUs, lo que permite entrenar modelos en grandes conjuntos de datos de manera eficiente.
- **Registro de métricas y seguimiento de experimentos**: Facilita el registro de métricas de entrenamiento y validación durante el proceso de entrenamiento, así como el seguimiento de experimentos a través de integraciones con herramientas populares como TensorBoard.
- **Reproducibilidad**: PyTorch Lightning promueve buenas prácticas para garantizar la reproducibilidad de los resultados experimentales, incluyendo la fijación de semillas aleatorias y la gestión del entorno de ejecución.

4.2.2. Componentes principales

PyTorch Lightning se compone de varios elementos esenciales que estructuran y simplifican el desarrollo de modelos. Los componentes principales incluyen:

- **LightningModule**: Es una clase base que encapsula la lógica de entrenamiento de un modelo, incluyendo la definición de la arquitectura del modelo, la lógica de entrenamiento y la lógica de evaluación.
- **Trainer**: Es una clase que gestiona el ciclo de entrenamiento del modelo, incluyendo la iteración sobre lotes de datos, el cálculo de gradientes, la actualización de parámetros y la ejecución de lógica adicional como el registro de métricas y el seguimiento de experimentos.

- **DataModule**: Es una clase que encapsula la lógica de carga y preprocesamiento de los datos, separando esta lógica del modelo y permitiendo una mejor modularidad y reutilización del código.

4.2.3. Llamadas de retorno (Callbacks)

En PyTorch Lightning, los callbacks son objetos que se utilizan para personalizar y extender el comportamiento del entrenamiento del modelo. Estos callbacks se activan en diferentes puntos del ciclo de entrenamiento y pueden realizar acciones como realizar registros, guardar modelos, realizar ajustes dinámicos en hiperparámetros, etc. Son una forma conveniente de agregar funcionalidades adicionales al proceso de entrenamiento sin modificar directamente el código del modelo o del entrenador.

- ModelCheckpoint: Este callback permite guardar el modelo con la mejor métrica durante el entrenamiento. Puede configurarse para guardar los modelos según la métrica de pérdida o cualquier otra métrica específica.
- EarlyStopping: Este callback detiene el entrenamiento si la métrica especificada no mejora después de un número determinado de épocas. Esto ayuda a evitar el sobreajuste del modelo y a ahorrar tiempo de entrenamiento.
- ProgressBar: Este callback muestra una barra de progreso durante el entrenamiento, que proporciona información sobre el progreso del entrenamiento, el tiempo estimado restante y las métricas de entrenamiento.
- TensorBoardLogger: Este callback registra las métricas de entrenamiento en archivos compatibles con TensorBoard, lo que permite visualizar y comparar las métricas a lo largo del entrenamiento.

4.2.4. Aplicación de LightningModel a nuestro caso

En esta sección, se detalla cómo se configura y utiliza una clase específica en PyTorch Lightning para entrenar y evaluar un modelo de red neuronal de manera eficiente. La clase se inicializa con dos parámetros fundamentales:

- **Modelo de PyTorch**: Este es el modelo de red neuronal que se desea entrenar. PyTorch es un framework ampliamente utilizado en machine learning por su flexibilidad y eficiencia.
- **Tasa de aprendizaje**: Este parámetro esencial determina la magnitud de los ajustes en los pesos del modelo durante la optimización, influenciando directamente la velocidad y eficacia del entrenamiento.

Dentro de la clase, se encuentran varios métodos que estructuran y simplifican el proceso de entrenamiento y evaluación:

- **Propagación hacia adelante**: Este método define cómo se generan las predicciones del modelo a partir de las entradas, describiendo la manera en que el modelo procesa la información para producir resultados.

- **Cálculo de pérdida y predicciones**: Un método específico encapsula el código repetitivo necesario para calcular la pérdida y las predicciones del modelo. Esto promueve la reutilización del código y mejora la claridad y organización del mismo.
- **Fases del entrenamiento, validación y prueba**: Cada fase tiene métodos dedicados:
  - **Entrenamiento**: Se calcula la pérdida y se actualizan los pesos del modelo, permitiendo que el modelo aprenda de los datos.
  - **Validación**: Se evalúa el rendimiento del modelo calculando la pérdida y otras métricas importantes durante el proceso de validación.
  - **Prueba**: Similar a la validación, pero enfocada en evaluar el rendimiento final del modelo tras el entrenamiento.
- **Configuración del optimizador**: Se especifica cómo se configura el optimizador que ajustará los parámetros del modelo durante el entrenamiento. En este caso, se usa un optimizador adaptativo con la tasa de aprendizaje previamente definida.
- **Configuración de callbacks**: Se establecen herramientas adicionales que controlan aspectos importantes del entrenamiento. Por ejemplo:
  - **Early Stopping**: Detiene el entrenamiento si la mejora en la pérdida es insignificante después de varias épocas, optimizando así el tiempo de entrenamiento.
  - **ModelCheckpoint**: Guarda el mejor modelo basado en la precisión durante la validación, asegurando que se preserva el mejor desempeño alcanzado.

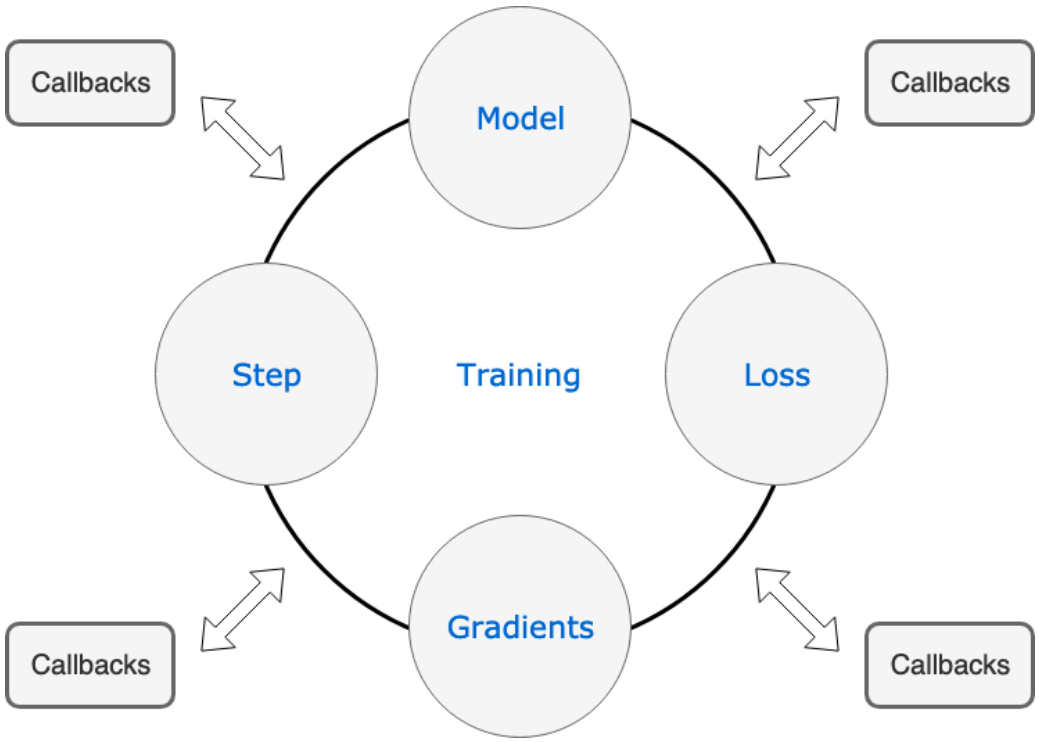


Figura 4.2: Esquema de funcionamiento de los callbacks. Fuente: GitHub.

Capítulo 5

Resultados obtenidos

Para mi proyecto, he decidido utilizar la técnica de transfer learning aprovechando modelos preentrenados como VGG16 y LeNet-5, reconocidos por su excelente rendimiento en la clasificación de imágenes. Estos modelos, entrenados en conjuntos de datos masivos como ImageNet, han desarrollado un profundo entendimiento de características generales de las imágenes, como formas, texturas y patrones.

Al aplicar transfer learning, se puede construir sobre este conocimiento previo, adaptando los modelos a la tarea específica que se desea abordar. Esta estrategia permite aprovechar la experiencia acumulada durante el entrenamiento inicial, logrando un proceso de ajuste más eficiente y un mejor rendimiento en la tarea de clasificación de imágenes.

En transfer learning, se retiene la mayor parte del conocimiento del modelo preentrenado y se ajustan solo las capas finales, conocidas como 'capas de cabeza', para que se adapten a la nueva tarea. Esto reduce significativamente el tiempo y los recursos necesarios para el entrenamiento, además de ayudar a evitar el sobreajuste, ya que el modelo preentrenado ya ha capturado una representación generalizada de los datos y solo requiere ajustes específicos.

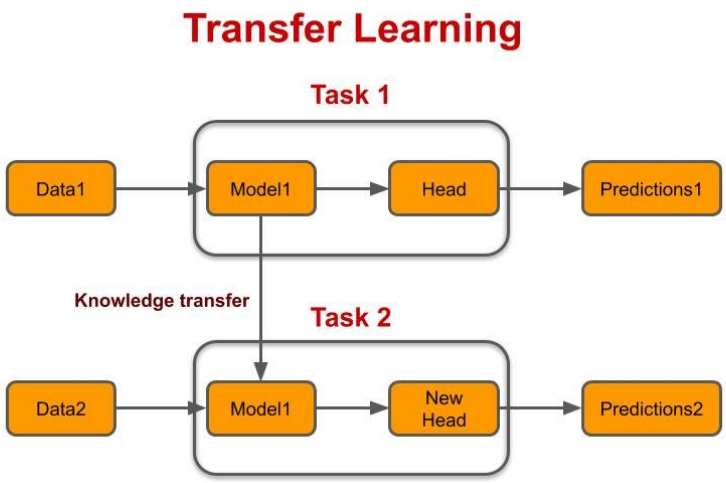


Figura 5.1: Esquema de Transfer Learning. Fuente: Medium.

5.1. Modelo VGG16

El modelo VGG16 es una arquitectura de red neuronal convolucional (CNN) diseñada por el grupo de investigación Visual Geometry Group (VGG) en la Universidad de Oxford. Fue presentada en la conferencia ILSVRC (ImageNet Large Scale Visual Recognition Challenge) en 2014. Esta red neuronal es conocida por su simplicidad relativa y su profundidad, y ha sido ampliamente utilizada en la comunidad de aprendizaje profundo para tareas de clasificación de imágenes.

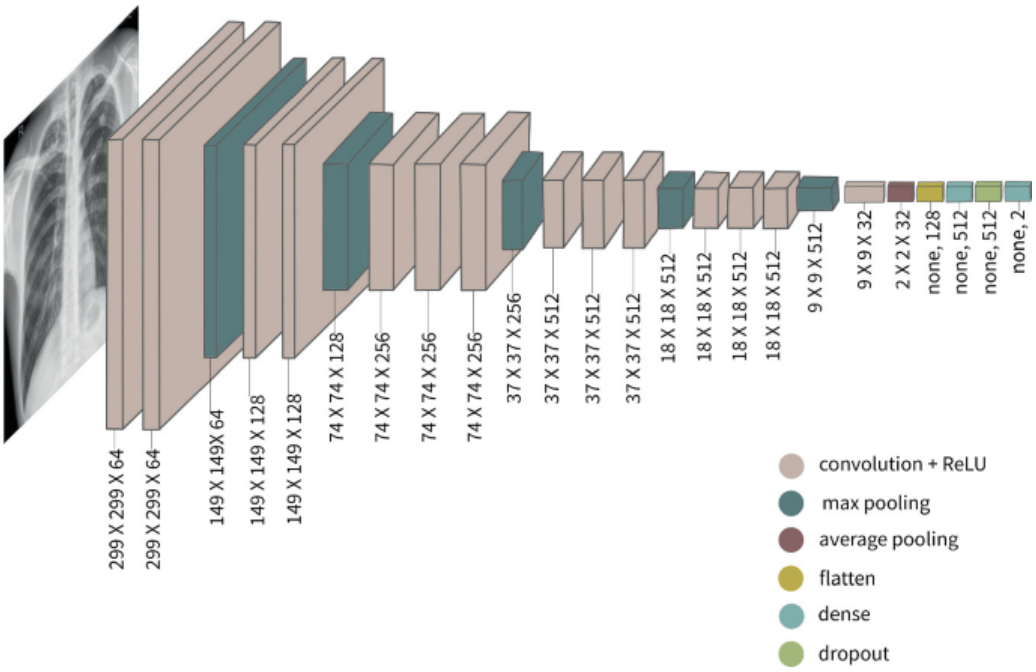


Figura 5.2: Esquema de la arquitectura VGG16 utilizada. Fuente: Google Imágenes.

1. **Arquitectura de Capas Convolucionales Profundas**: El modelo VGG16 consta de 16 capas de procesamiento convolucional, de ahí el nombre "VGG16". Estas capas están organizadas en bloques, donde cada bloque consiste en una serie de capas convolucionales seguidas por una capa de agrupación (pooling) para reducir la dimensionalidad.

2. **Tamaño de Filtro Pequeño y Convoluciones Aprofundadas**: A diferencia de algunas arquitecturas más modernas que utilizan filtros de convolución más grandes, VGG16 utiliza filtros de 3x3 en todas sus capas convolucionales. Esto ayuda a mantener una representación local de las características y a aprender representaciones más profundas y abstractas.

3. **Capas de Agrupación (Pooling)**: Después de cada conjunto de capas convolucionales, se utiliza una capa de agrupación máxima (max pooling) para reducir la dimensionalidad espacial de la representación, conservando las características más importantes.

4. **Capas Completamente Conectadas**: Después de múltiples capas de convolución y agrupación, la representación de la imagen se pasa a través de una o más capas completamente conectadas, que son similares a las capas ocultas de una red neuronal estándar. Estas capas finales están diseñadas para realizar la clasificación final de las características aprendidas en las capas convolucionales.

5. **Función de Activación ReLU**: En cada capa convolucional y completamente conectada, la función de activación Rectified Linear Unit (ReLU) se utiliza comúnmente para introducir no linealidad en la red y ayudar a aprender representaciones más complejas de los datos.

6. **Finalización con Softmax**: La última capa del modelo VGG16 utiliza la función de activación Softmax para generar una distribución de probabilidad sobre las posibles clases de salida. Esto hace que el modelo sea adecuado para tareas de clasificación de imágenes donde se necesita predecir la clase de un objeto en una imagen.

5.1.1. Transformaciones previas

Para el modelo VGG16, se han definido una serie de transformaciones previas que se aplican a las imágenes de entrada. Estas transformaciones tienen como objetivo preparar las imágenes para que sean compatibles con la arquitectura del modelo, así como mejorar la capacidad del modelo para generalizar a nuevos datos a través de técnicas de aumento de datos. A continuación, se explican y razonan detalladamente cada una de estas transformaciones:

- **Redimensionamiento a 224x224:** VGG16 espera imágenes de entrada con dimensiones de 224x224 píxeles. La transformación de redimensionamiento asegura que todas las imágenes tengan este tamaño estándar, lo cual es crucial para que el modelo funcione correctamente. Al redimensionar las imágenes a 224x224, se garantiza que la red convolucional pueda procesar las características espaciales de las imágenes de manera adecuada y consistente.

- **Conversión a tensor:** La transformación `ToTensor()` convierte las imágenes de un formato de datos estándar (como PIL Image) a tensores, que es el formato requerido por PyTorch para el procesamiento y entrenamiento de modelos. Esta conversión es esencial para que las imágenes puedan ser utilizadas por las operaciones de PyTorch, permitiendo así que las imágenes sean manipuladas en el espacio de datos tensoriales.

- **Normalización:** La normalización de las imágenes se realiza mediante la transformación `Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))`. Esta transformación ajusta los valores de los píxeles de las imágenes para que tengan una media de 0.5 y una desviación estándar de 0.5 para cada uno de los tres canales (rojo, verde y azul). La normalización centra los valores de los píxeles alrededor de cero, lo cual es beneficioso para el entrenamiento del modelo ya que acelera la convergencia del algoritmo de optimización y ayuda a prevenir problemas numéricos.

- **Conversión a escala de grises:** Aunque VGG16 fue diseñado originalmente para trabajar con imágenes en color, en este contexto específico se ha decidido convertir las imágenes a escala de grises. Esto reduce la complejidad del modelo al disminuir el número de canales de color de tres (RGB) a uno. La conversión a escala de grises puede ayudar a enfocarse en las características de textura y forma, que son cruciales para la clasificación de rayos X.

- **Volteo horizontal aleatorio:** La transformación de volteo horizontal aleatorio introduce variabilidad en el conjunto de entrenamiento al voltear las imágenes

horizontalmente con una cierta probabilidad. Esta técnica de aumento de datos ayuda a que el modelo sea más robusto frente a las variaciones en la orientación de las imágenes. En el caso de las radiografías de tórax, aunque los pulmones suelen estar en una posición estándar, esta variabilidad adicional combate el sobreajuste y mejora la generalización del modelo.

- **Rotación aleatoria:** Se aplica una rotación aleatoria de hasta 10 grados a las imágenes. Esta transformación simula pequeñas variaciones en el ángulo de las radiografías que pueden ocurrir en la práctica clínica. Exponer el modelo a estas variaciones durante el entrenamiento mejora su capacidad para manejar discrepancias en la orientación y mantener una alta precisión en la clasificación, incluso cuando las imágenes no estén perfectamente alineadas.

5.1.2. Información del modelo

El modelo VGG16 contiene aproximadamente 134 millones de parámetros. Todos estos parámetros son entrenables, lo que significa que se pueden ajustar durante el proceso de entrenamiento para minimizar la pérdida y mejorar la precisión del modelo. La gran cantidad de parámetros permite al modelo capturar detalles complejos en las imágenes, pero también lo hace más susceptible al sobreajuste si no se maneja correctamente.

El tamaño total estimado del modelo es de aproximadamente 537 MB. Este tamaño incluye todos los parámetros y estructuras necesarias para ejecutar el modelo. Un modelo de este tamaño requiere recursos computacionales significativos para el entrenamiento y la inferencia, lo cual debe tenerse en cuenta al planificar la infraestructura de hardware.

5.1.3. Precisión de validación

Durante el entrenamiento, la precisión de validación (`valid_acc`) alcanzó un valor máximo de aproximadamente 71.98 %. Sin embargo, en el paso 89, la precisión suavizada mostró un valor de 0.78, indicando que el modelo estaba mejorando y ajustándose bien a los datos de validación. Esta medida es crucial porque refleja el rendimiento del modelo en datos que no ha visto durante el entrenamiento, proporcionando una indicación de su capacidad de generalización.

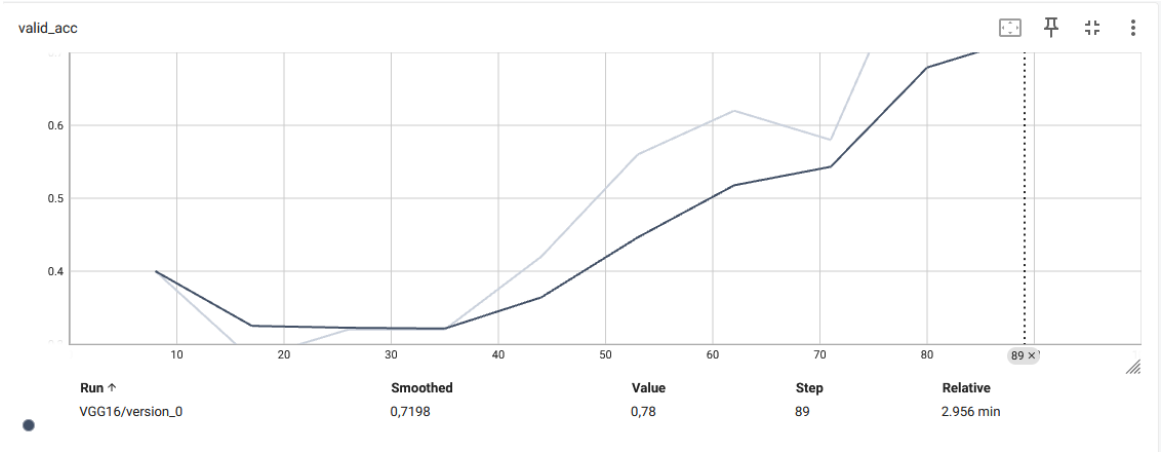


Figura 5.3: Valid Accuracy VGG16

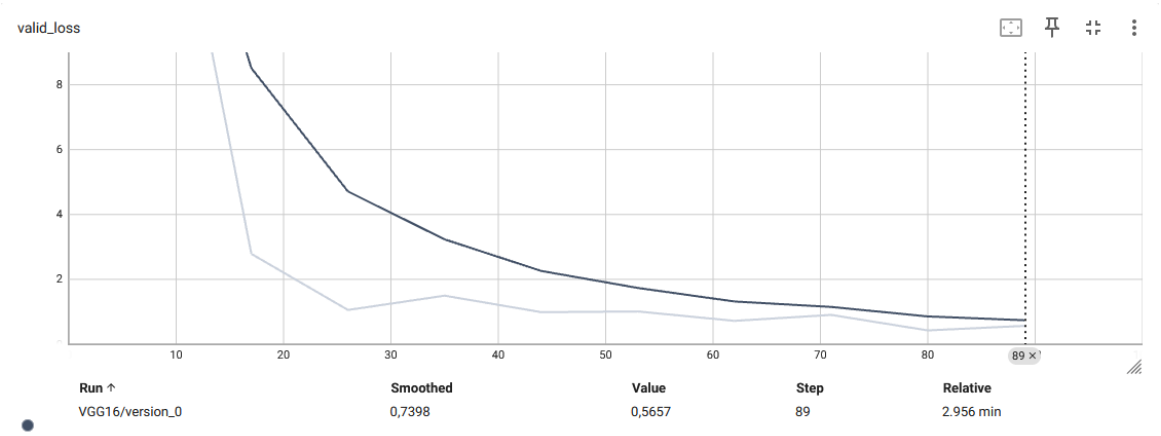


Figura 5.4: Valid Loss VGG16

5.1.4. Precisión de la Prueba

Después del entrenamiento, el modelo alcanzó una precisión de prueba (`test_acc`) de aproximadamente 72.7 %. Esto significa que, en promedio, el modelo clasifica correctamente el 72.7 % de las imágenes en el conjunto de prueba. Aunque es un buen punto de partida, hay margen para mejoras, especialmente en contextos críticos como el diagnóstico médico, donde se espera una precisión más alta.

5.1.5. Análisis de la Matriz de Confusión

La matriz de confusión obtenida tras la prueba del modelo muestra cómo se están clasificando las diferentes categorías de imágenes:

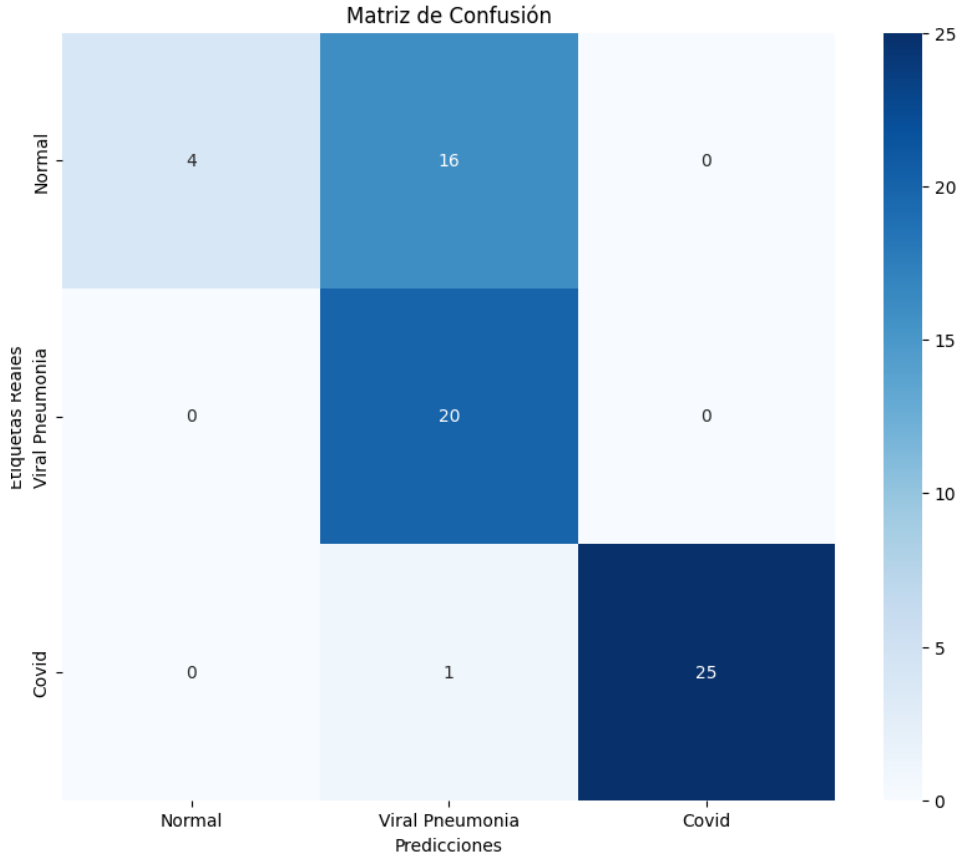


Figura 5.5: Matriz de confusión de VGG16

Este resultado nos permite identificar claramente los puntos fuertes y débiles del modelo:

- El modelo logró clasificar correctamente 4 imágenes como normales, pero erróneamente identificó 16 imágenes de neumonía como normales. Este es un problema serio, ya que clasificar incorrectamente la neumonía como normal puede llevar a una falta de tratamiento adecuado para los pacientes afectados.
- En contraste, el modelo mostró una alta precisión al clasificar las imágenes de neumonía, acertando en las 20 imágenes sin cometer errores. Esto sugiere que el modelo ha aprendido bien las características distintivas de la neumonía.
- Además, el desempeño del modelo fue sólido al clasificar imágenes de COVID-19, con 25 clasificaciones correctas. Sin embargo, hubo un error donde una imagen de neumonía fue clasificada como COVID-19.

5.1.6. Problema Identificado y análisis

La matriz de confusión revela un problema crítico: una alta tasa de falsas negativas en la categoría de neumonía, donde estas imágenes se clasifican incorrectamente como normales. Esto es particularmente preocupante porque en un contexto clínico, confundir casos de neumonía con imágenes normales puede llevar a que los pacientes no reciban el tratamiento adecuado a tiempo, resultando en serias complicaciones de salud.

El modelo muestra un sesgo hacia la clasificación de imágenes de neumonía como normales, lo que puede deberse a varios factores:

- **Desequilibrio en el conjunto de datos**: Si hay significativamente más imágenes de una clase que de otra en el conjunto de datos de entrenamiento, el modelo puede volverse sesgado hacia la clase más representada. Esto podría explicar los numerosos errores al clasificar neumonía como normal.
- **Similitud visual**: Las características visuales de las imágenes de neumonía y las normales pueden ser muy similares, dificultando que el modelo las distinga correctamente. Esto sugiere que el modelo necesita aprender características más específicas o que se debe mejorar la calidad del preprocesamiento de las imágenes.
- **Efectividad del preprocesamiento**: Las técnicas actuales de aumento de datos y preprocesamiento pueden no estar resaltando adecuadamente las diferencias entre las clases. Mejorar estas técnicas podría ayudar al modelo a aprender mejor las características distintivas.

Para abordar estos problemas, se pueden considerar las siguientes acciones:

- **Balancear el conjunto de datos**: Asegurar que haya una representación equitativa de todas las clases en el conjunto de datos de entrenamiento para reducir el sesgo del modelo.
- **Mejorar el preprocesamiento**: Implementar técnicas avanzadas de preprocesamiento para resaltar las características distintivas entre las imágenes de neumonía y las normales.
- **Ajuste de hiperparámetros y arquitectura**: Experimentar con diferentes arquitecturas de modelos y ajustar los hiperparámetros para mejorar la capacidad del modelo de distinguir entre clases similares.

5.2. Modelo LeNet-5

El modelo LeNet-5 es una arquitectura de red neuronal convolucional (CNN) desarrollada por Yann LeCun y su equipo a principios de la década de 1990. Fue una de las primeras arquitecturas de CNN en ganar prominencia y se utilizó principalmente para tareas de reconocimiento de caracteres escritos a mano.

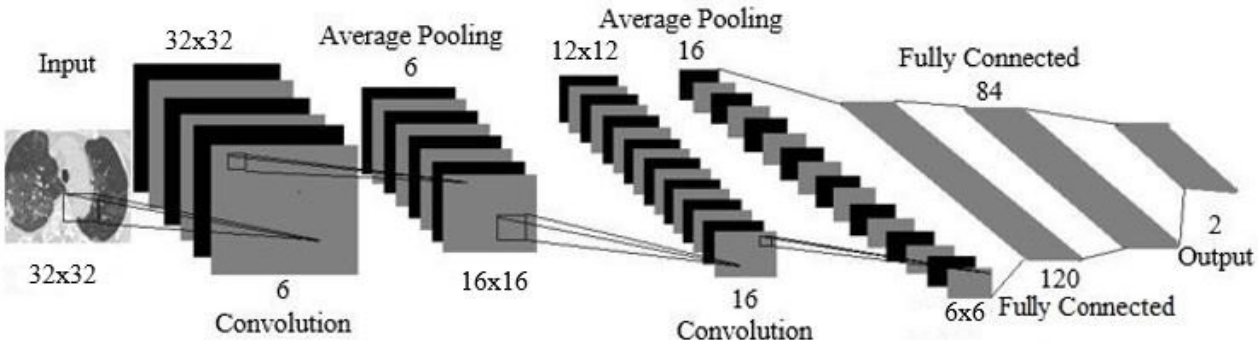


Figura 5.6: Esquema de la arquitectura Lenet5 utilizada. Fuente: Google Imágenes.

1. **Capas Convolutivas y Pooling**: El modelo LeNet-5 consta de varias capas convolutivas y de pooling (agrupación). Utiliza convoluciones 2D para extraer características

de las imágenes de entrada, seguidas de capas de pooling para reducir la dimensionalidad y conservar las características más importantes.

2. **Funciones de Activación**: En cada capa convolutiva y de pooling, se aplica una función de activación no lineal para introducir no linealidad en la red. En la versión original de LeNet-5, se utilizaba la función de activación sigmoide, aunque en implementaciones modernas, se prefiere comúnmente ReLU (Rectified Linear Unit).

3. **Capas Completamente Conectadas**: Después de las capas convolutivas y de pooling, la representación resultante se pasa a través de una o más capas completamente conectadas para realizar la clasificación final. Estas capas utilizan una combinación de operaciones de matriz y funciones de activación para generar las salidas finales.

4. **Clasificación y Softmax**: La última capa del modelo LeNet-5 utiliza la función de activación Softmax para generar una distribución de probabilidad sobre las posibles clases de salida. Esto permite al modelo realizar la clasificación de objetos en las imágenes de entrada.

Aunque el modelo LeNet-5 fue desarrollado hace décadas, su arquitectura influyó en el diseño de muchas arquitecturas de CNN modernas y sigue siendo un punto de referencia importante en el campo del aprendizaje profundo.

5.2.1. Transformaciones previas

Para el modelo LeNet5, se han definido una serie de transformaciones previas que se aplican a las imágenes de entrada. Estas transformaciones no solo preparan las imágenes para ser compatibles con la arquitectura del modelo, sino que también mejoran la capacidad del modelo para generalizar a nuevos datos mediante técnicas de aumento de datos. A continuación, se explican y razonan detalladamente cada una de estas transformaciones:

- **Redimensionamiento a 32x32:** La arquitectura original de LeNet5 espera imágenes de tamaño 32x32 píxeles. Por lo tanto, la primera transformación redimensiona todas las imágenes a este tamaño estándar. Esto asegura que las imágenes de entrada sean compatibles con las dimensiones esperadas por el modelo, evitando errores durante el proceso de entrenamiento y asegurando que las características espaciales relevantes sean capturadas adecuadamente.

- **Conversión a escala de grises:** LeNet5 fue diseñado originalmente para trabajar con imágenes en escala de grises, como las imágenes del conjunto de datos MNIST. La conversión de las imágenes a escala de grises reduce la complejidad del modelo al disminuir el número de canales de color de tres (RGB) a uno. Esto no solo simplifica el procesamiento de la imagen, sino que también puede ayudar a enfocarse en las características de textura y forma, que son cruciales para la clasificación de rayos X.

- **Volteo horizontal aleatorio:** Esta transformación realiza un volteo horizontal aleatorio de las imágenes con una cierta probabilidad. El aumento de datos mediante el volteo horizontal introduce variabilidad en el conjunto de entrenamiento, ayudando a que el modelo sea más robusto frente a las variaciones en la orientación de las imágenes. En el caso de las radiografías de tórax, aunque los pulmones suelen estar

en una posición estándar, esta técnica ayuda a combatir el sobreajuste y a mejorar la generalización del modelo.

- **Rotación aleatoria:** Se aplica una rotación aleatoria de hasta 10 grados a las imágenes. Esta transformación simula pequeñas variaciones en el ángulo de las radiografías que pueden ocurrir en la práctica clínica. Al exponer el modelo a estas variaciones durante el entrenamiento, se mejora su capacidad para manejar discrepancias en la orientación y mantener una alta precisión en la clasificación, incluso cuando las imágenes no estén perfectamente alineadas.
- **Conversión a tensor:** La transformación `ToTensor()` convierte las imágenes de un formato de datos estándar (por ejemplo, PIL Image) a tensores, que son el formato requerido por PyTorch para el procesamiento y entrenamiento de modelos. Esta conversión es esencial para que las imágenes puedan ser utilizadas por las operaciones de PyTorch.
- **Normalización:** Finalmente, las imágenes se normalizan utilizando la transformación `Normalize((0.5,), (0.5,))`. La normalización ajusta los valores de los píxeles de las imágenes para que tengan una media de 0.5 y una desviación estándar de 0.5. Esto centra los valores de los píxeles alrededor de cero, lo cual es beneficioso para el entrenamiento del modelo ya que acelera la convergencia del algoritmo de optimización y ayuda a prevenir problemas numéricos.

5.2.2. Información del modelo

El modelo LeNet5 contiene aproximadamente 61,100 parámetros entrenables. Esta cantidad de parámetros es significativamente menor comparada con los 134 millones de parámetros de VGG16. Menos parámetros implican menos recursos computacionales necesarios para entrenar el modelo y una menor probabilidad de sobreajuste.

El tamaño total estimado del modelo es de aproximadamente 0.244 MB. Este tamaño compacto hace que LeNet5 sea ideal para aplicaciones con limitaciones de memoria y procesamiento.

5.2.3. Precisión de validación

Durante el entrenamiento, la precisión de validación (`valid_acc`) alcanzó un valor máximo de aproximadamente 82.17 % en el paso 31, con una pérdida de validación (`valid_loss`) de 0.4282. Estos valores indican que el modelo se ajusta bien a los datos de validación, sugiriendo una buena capacidad de generalización.

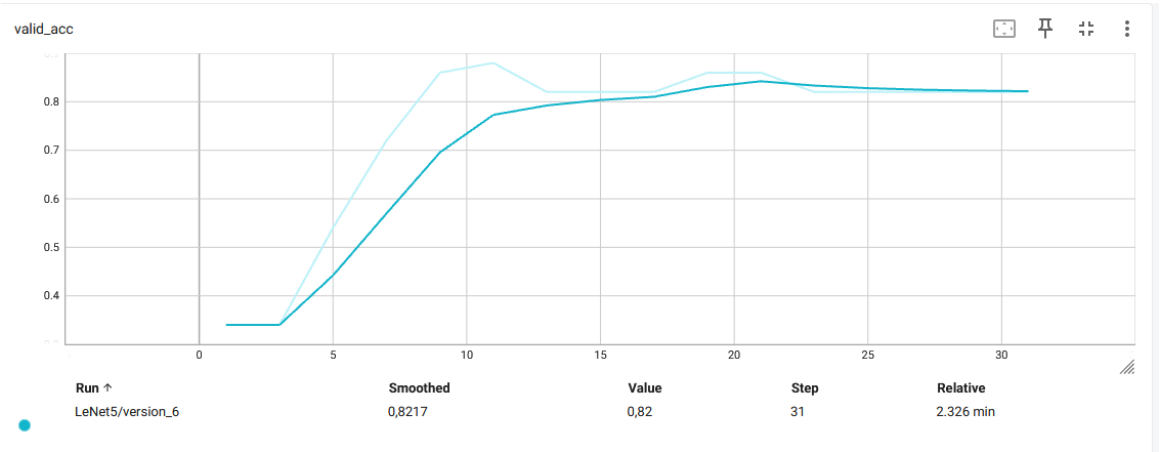


Figura 5.7: Valid Accuracy Lenet-5

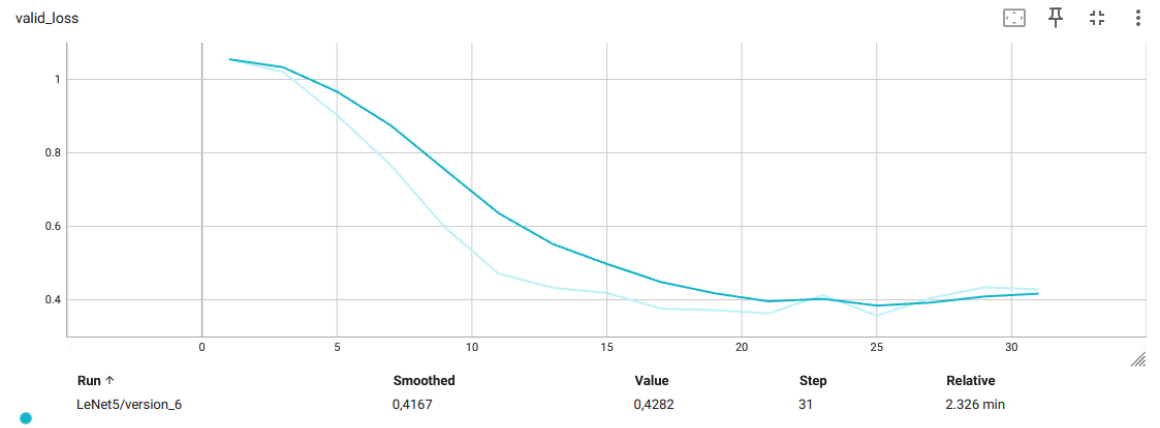


Figura 5.8: Valid Loss Lenet-5

5.2.4. Precisión de la prueba

Después del entrenamiento, el modelo LeNet5 alcanzó una precisión de prueba (`test_acc`) de aproximadamente 84.8 %. Este resultado es notablemente mejor que la precisión obtenida con el modelo VGG16 en este contexto específico, lo que sugiere que LeNet5 es más adecuado para este conjunto de datos o que las técnicas de preprocesamiento y aumento de datos han sido más efectivas.

5.2.5. Análisis de la matriz de confusión

La matriz de confusión obtenida tras la prueba del modelo LeNet5 muestra cómo se están clasificando las diferentes categorías de imágenes:

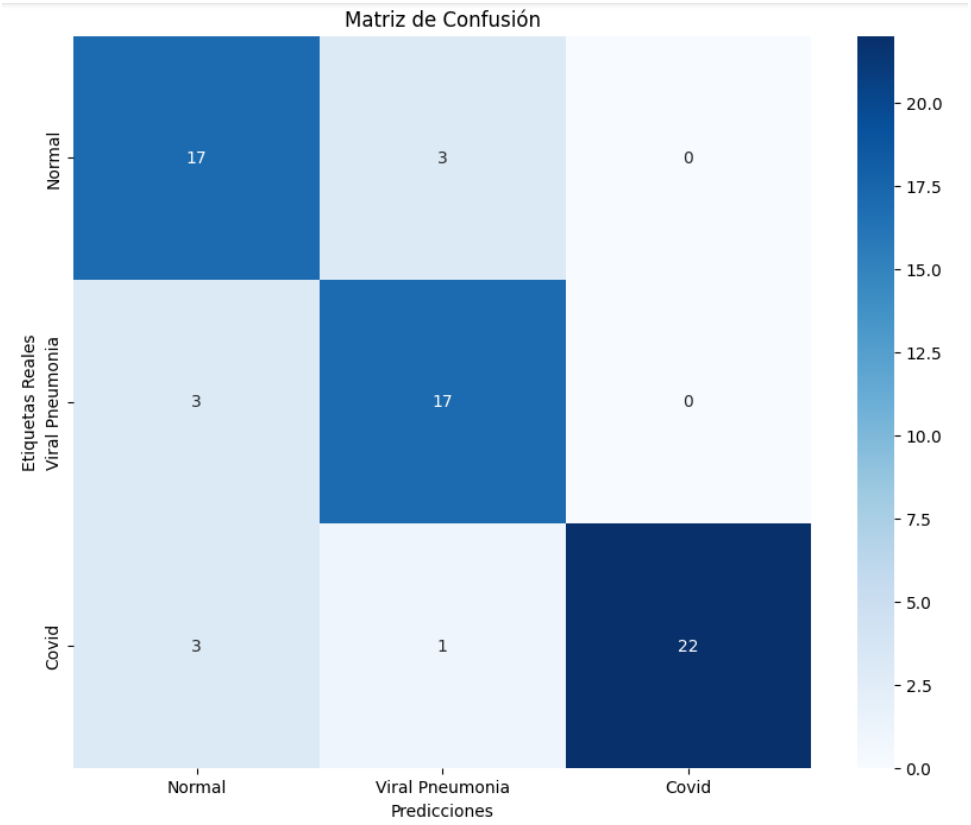


Figura 5.9: Matriz de confución de Lenet-5

El análisis comparativo entre el modelo LeNet5 y VGG16 revela una mejora significativa en la capacidad de LeNet5 para clasificar correctamente casos de neumonía.

- LeNet5 logró predecir correctamente 17 imágenes como normales, con solo 3 errores al clasificar imágenes de neumonía como normales. Esta mejora respecto a VGG16 es crucial, ya que reduce el riesgo de diagnósticos incorrectos, evitando que pacientes con neumonía sean tratados erróneamente como sanos.
- Además, mostró un buen desempeño al clasificar 17 imágenes de neumonía correctamente, con solo 3 imágenes clasificadas incorrectamente como normales. Aunque todavía hay margen de mejora, el número de errores es notablemente menor en comparación con VGG16, indicando una mejor distinción entre las clases.
- En cuanto a la categoría de COVID-19, LeNet5 clasificó correctamente 22 imágenes. Sin embargo, 3 imágenes de COVID-19 fueron clasificadas incorrectamente como normales y 1 imagen de neumonía fue clasificada como COVID-19.

El análisis de la matriz de confusión sugiere que LeNet5 tiene una mayor capacidad para diferenciar entre imágenes normales y de neumonía, lo cual es crucial en aplicaciones médicas donde la precisión diagnóstica es fundamental.

5.2.6. Posibles razones del mejor rendimiento

LeNet5, siendo un modelo más simple con menos parámetros, puede generalizar mejor en este conjunto de datos específico. La simplicidad del modelo ayuda a evitar el sobreajuste y permite una convergencia más rápida durante el entrenamiento.

El preprocesamiento de datos y las técnicas de aumento de datos pueden haber sido más efectivas en LeNet5, resaltando mejor las características distintivas entre las clases de imágenes. Estas técnicas son cruciales para mejorar la capacidad del modelo para diferenciar entre clases.

Capítulo 6

Conclusiones finales

En este trabajo, se ha realizado una comparativa exhaustiva entre dos arquitecturas de redes neuronales convolucionales (CNN) populares: VGG16 y LeNet5, aplicadas a un conjunto de datos de rayos X de tórax para clasificar imágenes en tres categorías: normal, neumonía y COVID-19. A través del análisis de los resultados obtenidos, se ha podido determinar la idoneidad de cada modelo para esta tarea específica.

6.1. Comparación entre VGG16 y LeNet5

VGG16 es una red neuronal convolucional profunda, reconocida por su excelente desempeño en tareas de clasificación de imágenes generales debido a su capacidad de extraer características complejas gracias a sus múltiples capas convolucionales y de pooling. Sin embargo, en este contexto específico, los resultados obtenidos muestran que LeNet5 ha superado a VGG16 en varios aspectos clave.

Precisión del Modelo La precisión de prueba obtenida con LeNet5 fue de aproximadamente 84.8 %, en comparación con el 72.7 % alcanzado por VGG16. Este incremento en la precisión indica que LeNet5 es más adecuado para la tarea de clasificación de imágenes de rayos X de tórax en este conjunto de datos. La matriz de confusión revela que LeNet5 comete menos errores al clasificar imágenes de neumonía como normales, lo que es crítico en un contexto médico donde un diagnóstico incorrecto puede tener graves consecuencias.

Simplicidad y Eficiencia Computacional LeNet5 cuenta con aproximadamente 61,100 parámetros entrenables, una fracción de los 134 millones de parámetros de VGG16. Esta simplicidad no solo reduce los requisitos de memoria y tiempo de entrenamiento, sino que también disminuye el riesgo de sobreajuste, permitiendo que el modelo generalice mejor a nuevos datos. En aplicaciones prácticas donde los recursos computacionales pueden ser limitados, la eficiencia de LeNet5 es un factor decisivo.

Equilibrio en la Clasificación La matriz de confusión del modelo LeNet5 muestra una distribución más equilibrada en la clasificación de las diferentes categorías. Aunque aún se presentan errores, estos son menos frecuentes y menos críticos que los observados con VGG16. En particular, la capacidad de LeNet5 para minimizar la confusión entre neumonía y normalidad es un avance significativo, dado que un diagnóstico incorrecto en esta área puede llevar a tratamientos inapropiados.

6.2. Beneficios de Utilizar PyTorch Lightning

La implementación de los modelos utilizando PyTorch Lightning ha sido altamente beneficiosa por varias razones:

PyTorch Lightning abstrae gran parte de la complejidad del código de entrenamiento, permitiendo centrarse más en la lógica del modelo y menos en los detalles de implementación. Esto no solo reduce la probabilidad de errores, sino que también facilita la lectura y mantenimiento del código.

La clase `LightningModule` permite una gestión eficiente de los hiperparámetros y el seguimiento de los mismos durante los experimentos. Al utilizar la función `save_hyperparameters`, se asegura que todos los ajustes relevantes queden documentados y accesibles, lo que facilita la reproducibilidad y el ajuste fino del modelo.

PyTorch Lightning proporciona herramientas integradas para la monitorización del entrenamiento y el uso de callbacks como `EarlyStopping` y `ModelCheckpoint`. Estos callbacks son esenciales para evitar el sobreentrenamiento y para guardar los mejores modelos basados en métricas de validación, garantizando que se mantenga la mejor versión del modelo sin necesidad de intervención manual.

El framework facilita la escalabilidad del modelo a múltiples GPUs o nodos, lo que es crucial para grandes conjuntos de datos o modelos más complejos. Aunque LeNet5 no requiere esta capacidad debido a su menor tamaño, la opción de escalabilidad es un valor añadido importante para futuros proyectos con mayores demandas computacionales.

6.3. Consideraciones Finales

En conclusión, aunque VGG16 es un modelo más complejo y profundo, los resultados obtenidos con LeNet5 fueron superiores en el contexto de nuestra tarea específica de clasificación de imágenes de rayos X de tórax. LeNet5 no solo mostró una mayor precisión global, sino que también cometió menos errores críticos al clasificar casos de neumonía como normales. Esto subraya la importancia de elegir el modelo adecuado según la naturaleza de la tarea y los datos disponibles.

Finalmente, se adjunta el notebook de Google Colab utilizado durante este proyecto, que contiene el grueso de las horas dedicadas al desarrollo y experimentación del TFM. Este notebook proporciona una visión detallada del proceso de entrenamiento, validación y pruebas de los modelos, así como de las técnicas de preprocesamiento y aumento de datos empleadas. Esperamos que este trabajo contribuya a la comprensión y mejora de los sistemas de clasificación de imágenes médicas mediante redes neuronales convolucionales

Capítulo 7

Final conclusions

This work presents a comprehensive comparison between the popular convolutional neural network architectures VGG16 and LeNet5, applied to a chest X-ray dataset to classify images into three categories: normal, pneumonia, and COVID-19. The results indicate that LeNet5 outperforms VGG16 in several key aspects. LeNet5 achieved a test accuracy of approximately 84.8 %, compared to 72.7 % for VGG16, showing fewer errors in classifying pneumonia as normal, which is crucial in a medical context. Additionally, with only 61,100 trainable parameters versus VGG16's 134 million, LeNet5 is more computationally efficient, reducing the risk of overfitting and improving generalization to new data. The simplicity and efficiency of LeNet5, along with its ability to maintain a balanced classification across different categories, highlight its suitability for this specific task.

The implementation of the models using PyTorch Lightning has proven beneficial, facilitating hyperparameter management, training monitoring, and providing integrated tools like EarlyStopping and ModelCheckpoint to prevent overtraining and ensure the best models are saved. Furthermore, PyTorch Lightning supports scalability to multiple GPUs or nodes, which, although not necessary for LeNet5 due to its smaller size, is a valuable feature for future projects with higher computational demands.

In conclusion, while VGG16 is a more complex model, LeNet5 showed superior results in the specific task of classifying chest X-ray images, underscoring the importance of selecting the appropriate model based on the task and available data. The Google Colab notebook used in this project is attached, documenting the training, validation, and testing process of the models, as well as the preprocessing and data augmentation techniques employed, with the hope of contributing to the improvement of medical image classification systems using convolutional neural networks.

Apéndice A

Apéndice

INSTALACIONES PREVIAS:

para pruebas: https://www.kaggle.com/datasets/amanullahasraf/covid19-pneumonia-normal-chest-xray-pa-dataset

```
!pip install pytorch-lightning
```

```
import os
import torch
import torchvision
import pytorch_lightning as pl
from torch.utils.data import DataLoader, Dataset, random_split
import torchvision.transforms as transforms
import torch.nn as nn
import torch.optim as optim
from pytorch_lightning.loggers import TensorBoardLogger
import torch.nn.functional as F
from pytorch_lightning.callbacks import ModelCheckpoint

device = "cuda" if torch.cuda.is_available() else "cpu"

torch.cuda.is_available()
```

```
True
```

```
device = "cuda" if torch.cuda.is_available() else "cpu"
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.m
```

```
# Establecer la variable de entorno JAX_PLATFORM_NAME en 'gpu'
os.environ['JAX_PLATFORM_NAME'] = 'gpu'

# Importar JAX después de configurar la variable de entorno
import jax
```

MODELO VGG16:

```
BATCH_SIZE = 25
NUM_EPOCHS = 10
LEARNING_RATE = 0.001
NUM_WORKERS = 2
```

```
num_classes=3

import torch
import torch.nn as nn

class PyTorchVGG16(nn.Module):

    def __init__(self, num_classes):
        super().__init__()

        # calculate same padding:
        # (w - k + 2*p)/s + 1 = o
        # => p = (s(o-1) - w + k)/2

        self.block_1 = nn.Sequential(
                nn.Conv2d(in_channels=1,
                          out_channels=64,
                          kernel_size=(3, 3),
                          stride=(1, 1),
                          # (1(32-1)- 32 + 3)/2 = 1
                          padding=1),
                nn.ReLU(),
                nn.Conv2d(in_channels=64,
                          out_channels=64,
                          kernel_size=(3, 3),
                          stride=(1, 1),
                          padding=1),
                nn.ReLU(),
                nn.MaxPool2d(kernel_size=(2, 2),
                             stride=(2, 2))
        )

        self.block_2 = nn.Sequential(
                nn.Conv2d(in_channels=64,
                          out_channels=128,
                          kernel_size=(3, 3),
                          stride=(1, 1),
                          padding=1),
                nn.ReLU(),
                nn.Conv2d(in_channels=128,
                          out_channels=128,
                          kernel_size=(3, 3),
                          stride=(1, 1),
                          padding=1),
                nn.ReLU(),
                nn.MaxPool2d(kernel_size=(2, 2),
                             stride=(2, 2))
        )

        self.block_3 = nn.Sequential(
                nn.Conv2d(in_channels=128,
                          out_channels=256,
                          kernel_size=(3, 3),
                          stride=(1, 1),
                          padding=1),
                nn.ReLU(),
```

```
            nn.Conv2d(in_channels=256,
                      out_channels=256,
                      kernel_size=(3, 3),
                      stride=(1, 1),
                      padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=256,
                      out_channels=256,
                      kernel_size=(3, 3),
                      stride=(1, 1),
                      padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2),
                         stride=(2, 2))
        )

        self.block_4 = nn.Sequential(
            nn.Conv2d(in_channels=256,
                      out_channels=512,
                      kernel_size=(3, 3),
                      stride=(1, 1),
                      padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=512,
                      out_channels=512,
                      kernel_size=(3, 3),
                      stride=(1, 1),
                      padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=512,
                      out_channels=512,
                      kernel_size=(3, 3),
                      stride=(1, 1),
                      padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2),
                         stride=(2, 2))
        )

        self.block_5 = nn.Sequential(
            nn.Conv2d(in_channels=512,
                      out_channels=512,
                      kernel_size=(3, 3),
                      stride=(1, 1),
                      padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=512,
                      out_channels=512,
                      kernel_size=(3, 3),
                      stride=(1, 1),
                      padding=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=512,
                      out_channels=512,
                      kernel_size=(3, 3),
                      stride=(1, 1),
                      padding=1),
```

```
                    padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2),
                         stride=(2, 2))
        )

        self.features = nn.Sequential(
            self.block_1, self.block_2,
            self.block_3, self.block_4,
            self.block_5,
            nn.AdaptiveAvgPool2d((7, 7))  # Añadir capa de agrupación global promedio
        )

        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096),  # Ajustar el número de entradas
            nn.ReLU(True),
            nn.Dropout(p=0.5),
            nn.Linear(4096, 4096),
            nn.ReLU(True),
            nn.Dropout(p=0.5),
            nn.Linear(4096, num_classes),
        )

        for m in self.modules():
            if isinstance(m, torch.nn.Conv2d):
                m.weight.detach().normal_(0, 0.05)
                if m.bias is not None:
                    m.bias.detach().zero_()
            elif isinstance(m, torch.nn.Linear):
                m.weight.detach().normal_(0, 0.05)
                m.bias.detach().detach().zero_()

    def forward(self, x):

        x = self.features(x)
        x = x.view(x.size(0), -1)
        logits = self.classifier(x)

        return logits

import pytorch_lightning as pl
from pytorch_lightning.callbacks import EarlyStopping
import torchmetrics


# LightningModule that receives a PyTorch model as input
class LightningModel(pl.LightningModule):
    def __init__(self, model, learning_rate):
        super().__init__()

        self.learning_rate = learning_rate
        # The inherited PyTorch module
        self.model = model

        # Save settings and hyperparameters to the log directory
        # but skip the model parameters
        self.save_hyperparameters(ignore=['model'])
```

```
        # Set up attributes for computing the accuracy
        self.train_acc = torchmetrics.Accuracy(task='MULTICLASS', num_classes=num_classes)
        self.valid_acc = torchmetrics.Accuracy(task='MULTICLASS', num_classes=num_classes)
        self.test_acc = torchmetrics.Accuracy(task='MULTICLASS', num_classes=num_classes)

    # Defining the forward method is only necessary
    # if you want to use a Trainer's .predict() method (optional)
    def forward(self, x):
        return self.model(x)

    # A common forward step to compute the loss and labels
    # this is used for training, validation, and testing below
    def _shared_step(self, batch):
        features, true_labels = batch
        logits = self(features)
        loss = torch.nn.functional.cross_entropy(logits, true_labels)
        predicted_labels = torch.argmax(logits, dim=1)

        return loss, true_labels, predicted_labels

    def training_step(self, batch, batch_idx):
        loss, true_labels, predicted_labels = self._shared_step(batch)
        self.log("train_loss", loss)

        # To account for Dropout behavior during evaluation
        self.model.eval()
        with torch.no_grad():
            _, true_labels, predicted_labels = self._shared_step(batch)
        self.train_acc.update(predicted_labels, true_labels)
        self.log("train_acc", self.train_acc, on_epoch=True, on_step=False)
        self.model.train()
        return loss  # this is passed to the optimzer for training

    def validation_step(self, batch, batch_idx):
        loss, true_labels, predicted_labels = self._shared_step(batch)
        self.log("valid_loss", loss)
        self.valid_acc(predicted_labels, true_labels)
        self.log("valid_acc", self.valid_acc,
                 on_epoch=True, on_step=False, prog_bar=True)

    def test_step(self, batch, batch_idx):
        loss, true_labels, predicted_labels = self._shared_step(batch)
        self.test_acc(predicted_labels, true_labels)
        self.log("test_acc", self.test_acc, on_epoch=True, on_step=False)

    def configure_optimizers(self):
        optimizer = torch.optim.Adam(self.parameters(), lr=self.learning_rate)
        return optimizer

    def configure_callbacks(self):
        # Configurar EarlyStopping
        early_stopping_callback = EarlyStopping(
            monitor='valid_loss',  # Métrica a monitorear (puede ser 'valid_acc' u otra métrica)
            min_delta=0.001,        # Cambio mínimo en la métrica para considerar como mejora
            patience=3,             # Número de épocas sin mejora antes de detener el entrenamiento
```

```
            verbose=True,           # Mostrar mensajes sobre la parada temprana
            mode='min'              # 'min' para minimizar la métrica, 'max' para maximizar
        )

        # Configurar ModelCheckpoint para guardar el mejor modelo según la precisión en el conjunto de v
        checkpoint_callback = ModelCheckpoint(
            monitor='valid_acc',  # Métrica para determinar el mejor modelo
            mode='max',           # 'max' para maximizar la métrica (por ejemplo, precisión)
            dirpath='checkpoints/',  # Directorio donde se guardarán los modelos
            filename='model-{epoch:02d}-{valid_acc:.2f}',  # Nombre del archivo de checkpoint
            save_top_k=1,         # Número máximo de modelos a guardar (en este caso, solo el mejor)
        )

        return [early_stopping_callback, checkpoint_callback]

from PIL import Image

class CustomDataset(Dataset):
    def __init__(self, root_dir, transform=None):
        self.root_dir = root_dir
        self.transform = transform
        self.classes = os.listdir(root_dir)

        self.data = []
        for label, class_name in enumerate(self.classes):
            class_dir = os.path.join(root_dir, class_name)
            for img_name in os.listdir(class_dir):
                img_path = os.path.join(class_dir, img_name)
                self.data.append((img_path, label))

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        img_path, label = self.data[idx]
        image = Image.open(img_path).convert('RGB')  # Read image with PIL
        if self.transform:
            image = self.transform(image)
        return image, label

from torch.utils.data import random_split, DataLoader
from torchvision import transforms

# Transformaciones de datos
transform = transforms.Compose([
    transforms.Resize((224, 224)),  # VGG espera imágenes de 224x224
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
    transforms.Grayscale(),  # Convertir a escala de grises
    transforms.RandomHorizontalFlip(),  # Volteo horizontal aleatorio
    transforms.RandomRotation(degrees=10),  # Rotación aleatoria hasta 10 grados
])

# Dataset para entrenamiento y prueba
train_dataset = CustomDataset(root_dir='/content/drive/MyDrive/TFM/Covid19-dataset/train', transform=tra
test_dataset = CustomDataset(root_dir='/content/drive/MyDrive/TFM/Covid19-dataset/test', transform=trans
```

```
# Calcular el tamaño del conjunto de validación (20% del tamaño del conjunto de entrenamiento)
val_size = int(0.2 * len(train_dataset))

# Calcular el tamaño del conjunto de entrenamiento (80% del tamaño del conjunto de entrenamiento)
train_size = len(train_dataset) - val_size

# Dividir el conjunto de entrenamiento en conjuntos de entrenamiento y validación
train_dataset, val_dataset = random_split(train_dataset, [train_size, val_size])

# Crear los DataLoader para entrenamiento, validación y prueba
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, num_workers=NUM_WORKERS)
val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=False, num_workers=NUM_WORKERS)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False, num_workers=NUM_WORKERS)
```

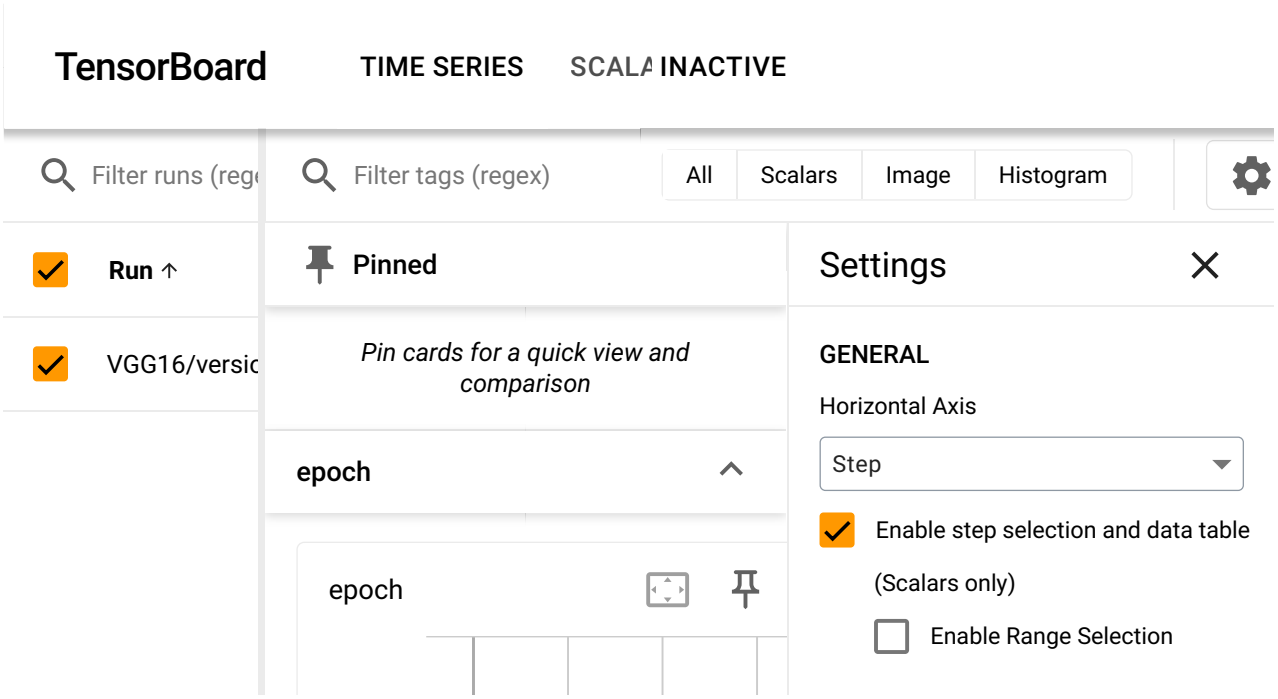
```
from torch.utils.tensorboard import SummaryWriter

# Configurar TensorBoard
writer = SummaryWriter()

# Instanciar el modelo de Lightning
model = LightningModel(model=PyTorchVGG16(num_classes=3), learning_rate=LEARNING_RATE)

trainer = pl.Trainer(
    max_epochs=NUM_EPOCHS,
    log_every_n_steps=1,
    logger=pl.loggers.TensorBoardLogger('logs/', name='VGG16'),
    callbacks=model.configure_callbacks(),
)
```

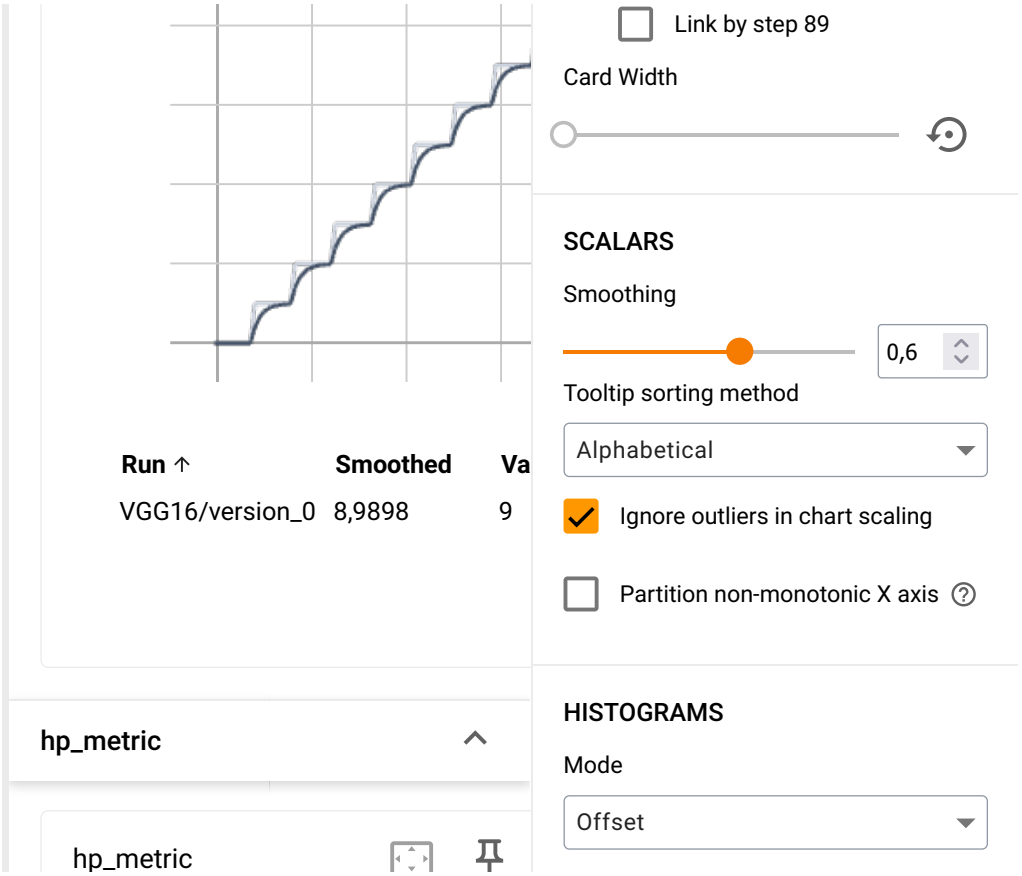
```
trainer.fit(model, train_loader, val_loader)

# Cerrar TensorBoard al finalizar
writer.close()
```

```
INFO:pytorch_lightning.utilities.rank_zero:GPU available: True (cuda), used: True
INFO:pytorch_lightning.utilities.rank_zero:TPU available: False, using: 0 TPU cores
INFO:pytorch_lightning.utilities.rank_zero:IPU available: False, using: 0 IPUs
INFO:pytorch_lightning.utilities.rank_zero:HPU available: False, using: 0 HPUs
INFO:pytorch_lightning.utilities.rank_zero:The following callbacks returned in `Light
WARNING:pytorch_lightning.loggers.tensorboard:Missing logger folder: logs/VGG16
INFO:pytorch_lightning.accelerators.cuda:LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
INFO:pytorch_lightning.callbacks.model_summary:
  | Name      | Type               | Params
-------------------------------------------------
0 | model     | PyTorchVGG16       | 134 M
1 | train_acc | MulticlassAccuracy | 0
2 | valid_acc | MulticlassAccuracy | 0
3 | test_acc  | MulticlassAccuracy | 0
-------------------------------------------------
134 M     Trainable params
0         Non-trainable params
134 M     Total params
537.087   Total estimated model params size (MB)
```

```
/usr/lib/python3.10/multiprocessing/popen_fork.py:66: RuntimeWarning: os.fork() was c
  self.pid = os.fork()
Epoch 9: 100%                                    9/9 [00:11<00:00,  0.75it/s, v_num=0, valid_acc=0.780]

INFO:pytorch_lightning.callbacks.early_stopping:Metric valid_loss improved. New best

INFO:pytorch_lightning.callbacks.early_stopping:Metric valid_loss improved by 15.274

INFO:pytorch_lightning.callbacks.early_stopping:Metric valid_loss improved by 1.730 >



INFO:pytorch_lightning.callbacks.early_stopping:Metric valid_loss improved by 0.068 >



INFO:pytorch_lightning.callbacks.early_stopping:Metric valid_loss improved by 0.268 >



INFO:pytorch_lightning.callbacks.early_stopping:Metric valid_loss improved by 0.294 >

INFO:pytorch_lightning.utilities.rank_zero:`Trainer.fit` stopped: `max_epochs=10` rea
```

```
%load_ext tensorboard
%tensorboard --logdir logs/
```






```
# Evaluar el modelo con los datos de prueba
trainer.test(model, test_loader)
```

```
INFO:pytorch_lightning.utilities.rank_zero:The following callbacks returned in `Light
INFO:pytorch_lightning.accelerators.cuda:LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
/usr/lib/python3.10/multiprocessing/popen_fork.py:66: RuntimeWarning: os.fork() was c
  self.pid = os.fork()
```

Testing DataLoader 0: 100% 3/3 [00:14<00:00, 0.21it/s]

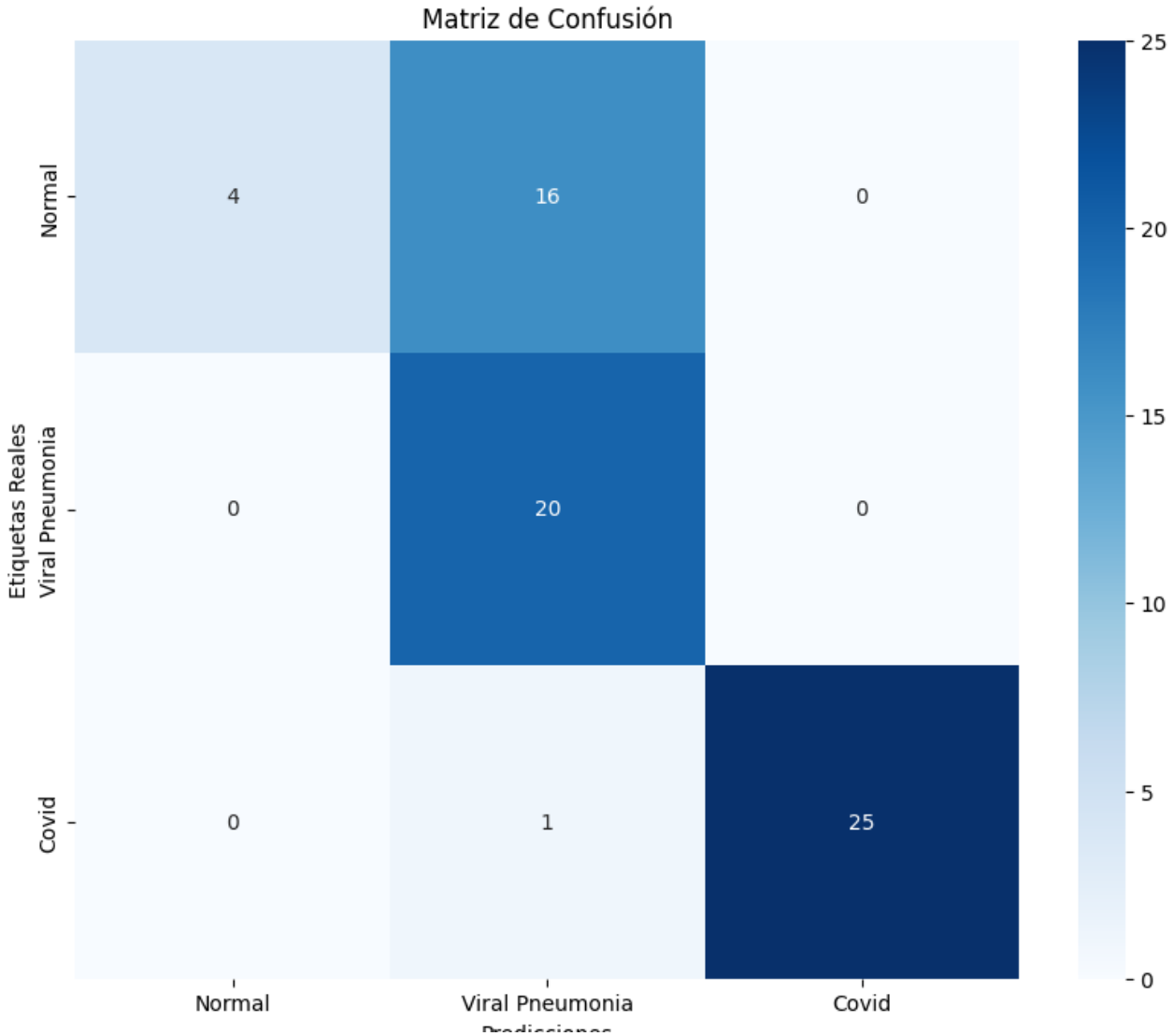
Test metric	DataLoader 0
test_acc	0.7272727489471436

```
/usr/lib/python3.10/multiprocessing/popen_fork.py:66: RuntimeWarning: os.fork() was c
  self.pid = os.fork()
[{'test_acc': 0.7272727489471436}]
```

```
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np

# Obtener las predicciones y etiquetas reales de los datos de prueba
predictions = []
targets = []
model.eval()
with torch.no_grad():
```

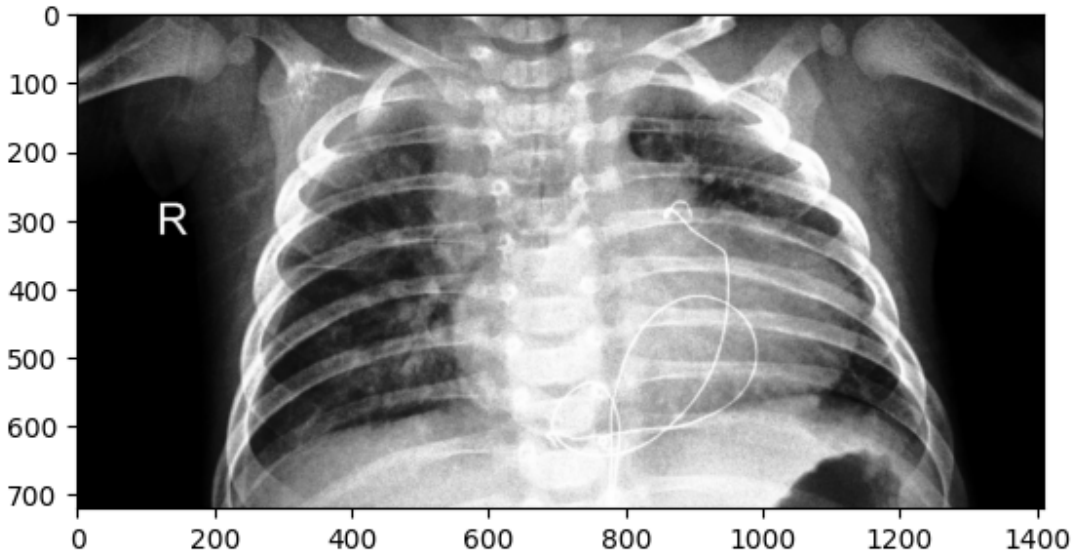
```
    for data, target in test_loader:
        output = model(data)
        predictions.extend(output.argmax(dim=1).tolist())
        targets.extend(target.tolist())

# Calcular la matriz de confusión
conf_matrix = confusion_matrix(targets, predictions)

# Obtener los nombres de las clases del conjunto de datos original
class_names = train_dataset.dataset.classes

# Visualizar la matriz de confusión con nombres de clases
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=class_names, yticklabels=class_n
plt.xlabel('Predicciones')
plt.ylabel('Etiquetas Reales')
plt.title('Matriz de Confusión')
plt.show()
```

```
/usr/lib/python3.10/multiprocessing/popen_fork.py:66: RuntimeWarning: os.fork() was c
  self.pid = os.fork()
/usr/lib/python3.10/multiprocessing/popen_fork.py:66: RuntimeWarning: os.fork() was c
  self.pid = os.fork()
```



Predicciones

```
from PIL import Image


image = Image.open('/content/drive/MyDrive/TFM/Covid19-dataset/test/Viral Pneumonia/0111.jpeg')
plt.imshow(image, cmap='Greys')
plt.show()
```




```
image_chw = transform(image)
print(image_chw.shape)
```

```
torch.Size([1, 224, 224])
```

```
image_nchw = image_chw.unsqueeze(0)
print(image_nchw.shape)
```

```
torch.Size([1, 1, 224, 224])
```

```
with torch.no_grad():  # since we don't need to backprop
    logits = model(image_nchw)
    probas = torch.softmax(logits, dim=1)  # Calcular las probabilidades con softmax a lo largo del eje
    predicted_label = torch.argmax(probas, dim=1)  # Predecir la clase con la probabilidad más alta


print(f'Class-membership probability: {probas[0][predicted_label.item()]*100:.2f}%')
print(f'Predicted label: {predicted_label.item()}')
```

```
Class-membership probability: 95.10%
Predicted label: 1
```

```
Predicted label: 1
```

MODELO LENET5:

```
BATCH_SIZE = 128
NUM_EPOCHS = 20
LEARNING_RATE = 0.001
NUM_WORKERS = 4
num_classes = 3

import torch

class PyTorchLeNet5(torch.nn.Module):

    def __init__(self, num_classes, grayscale=True):
        super().__init__()

        self.grayscale = grayscale
        self.num_classes = num_classes

        if self.grayscale:
            in_channels = 1
        else:
            in_channels = 3

        self.features = torch.nn.Sequential(
            torch.nn.Conv2d(in_channels, 6, kernel_size=5),
            torch.nn.Tanh(),
            torch.nn.MaxPool2d(kernel_size=2),
            torch.nn.Conv2d(6, 16, kernel_size=5),
            torch.nn.Tanh(),
            torch.nn.MaxPool2d(kernel_size=2)
        )

        self.classifier = torch.nn.Sequential(
            torch.nn.Linear(16*5*5, 120),
            torch.nn.Tanh(),
            torch.nn.Linear(120, 84),
            torch.nn.Tanh(),
            torch.nn.Linear(84, num_classes),
        )

    def forward(self, x):
        x = self.features(x)
        x = torch.flatten(x, start_dim=1)
        logits = self.classifier(x)
        return logits

from torchvision import transforms

# Define transforms for data augmentation and normalization
transform = transforms.Compose([
    transforms.Resize((32, 32)),  # LeNet-5 espera imágenes de 32x32
```

```
    transforms.Resize((32, 32)),  # LeNet-5 espera imágenes de 32x32
    transforms.Grayscale(),  # Convertir a escala de grises
    transforms.RandomHorizontalFlip(),  # Volteo horizontal aleatorio
    transforms.RandomRotation(degrees=10),  # Rotación aleatoria hasta 10 grados
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

# Dataset para entrenamiento y prueba
train_dataset = CustomDataset(root_dir='/content/drive/MyDrive/TFM/Covid19-dataset/train', transform=tra
test_dataset = CustomDataset(root_dir='/content/drive/MyDrive/TFM/Covid19-dataset/test', transform=trans

# Calcular el tamaño del conjunto de validación (20% del tamaño del conjunto de entrenamiento)
val_size = int(0.2 * len(train_dataset))

# Calcular el tamaño del conjunto de entrenamiento (80% del tamaño del conjunto de entrenamiento)
train_size = len(train_dataset) - val_size

# Dividir el conjunto de entrenamiento en conjuntos de entrenamiento y validación
train_dataset, val_dataset = random_split(train_dataset, [train_size, val_size])

# Crear los DataLoader para entrenamiento, validación y prueba
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, num_workers=NUM_WORKERS)
val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=False, num_workers=NUM_WORKERS)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False, num_workers=NUM_WORKERS)


from torch.utils.tensorboard import SummaryWriter
import pytorch_lightning as pl

# Configurar TensorBoard
writer = SummaryWriter()

# Instanciar el modelo de Lightning
model = LightningModel(PyTorchLeNet5(num_classes=3), learning_rate=LEARNING_RATE)

# Configura el Trainer de PyTorch Lightning
trainer = pl.Trainer(
    max_epochs=NUM_EPOCHS,
    log_every_n_steps=1,
    logger=pl.loggers.TensorBoardLogger('logs/', name='LeNet5'),
    callbacks=model.configure_callbacks(),
)

# Entrena el modelo
trainer.fit(model, train_loader, val_loader)

# Cerrar TensorBoard al finalizar
writer.close()
```

```
INFO:pytorch_lightning.utilities.rank_zero:GPU available: True (cuda), used: True
INFO:pytorch_lightning.utilities.rank_zero:TPU available: False, using: 0 TPU cores
INFO:pytorch_lightning.utilities.rank_zero:IPU available: False, using: 0 IPUs
INFO:pytorch_lightning.utilities.rank_zero:HPU available: False, using: 0 HPUs
INFO:pytorch_lightning.utilities.rank_zero:The following callbacks returned in `Light
/usr/local/lib/python3.10/dist-packages/pytorch_lightning/callbacks/model_checkpoint.
INFO:pytorch_lightning.accelerators.cuda:LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
```

```
INFO:pytorch_lightning.callbacks.model_summary:
  | Name      | Type               | Params
-----------------------------------------------
0 | model     | PyTorchLeNet5      | 61.1 K
1 | train_acc | MulticlassAccuracy | 0
2 | valid_acc | MulticlassAccuracy | 0
3 | test_acc  | MulticlassAccuracy | 0
-----------------------------------------------
61.1 K    Trainable params
0         Non-trainable params
61.1 K    Total params
0.244     Total estimated model params size (MB)


/usr/local/lib/python3.10/dist-packages/torch/utils/data/dataloader.py:558: UserWarni
  warnings.warn(_create_warning_msg(
/usr/lib/python3.10/multiprocessing/popen_fork.py:66: RuntimeWarning: os.fork() was c
  self.pid = os.fork()
```

Epoch 15: 100% 2/2 [00:09<00:00, 0.20it/s, v_num=6, valid_acc=0.820]

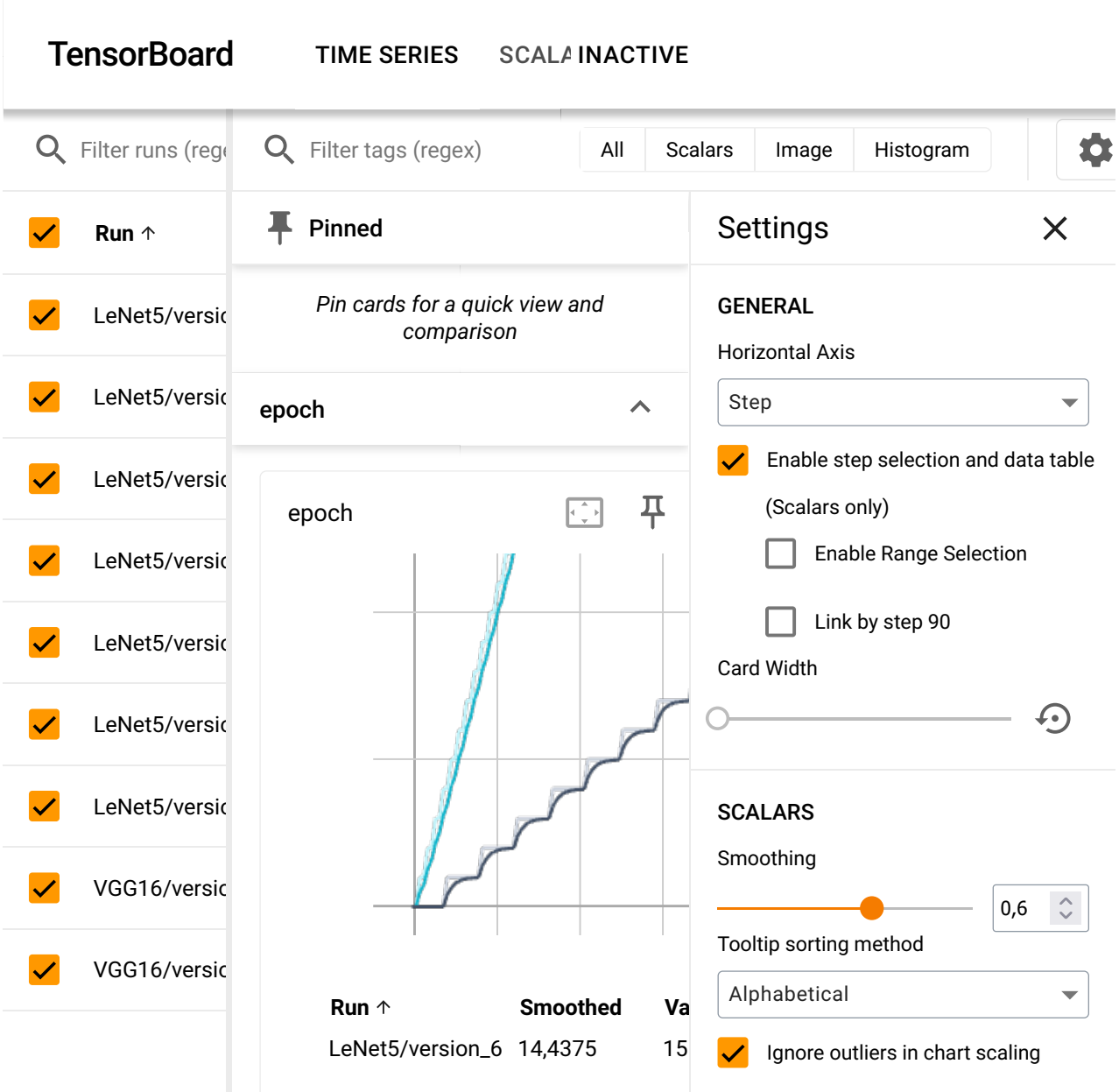
```
/usr/lib/python3.10/multiprocessing/popen_fork.py:66: RuntimeWarning: os.fork() was c
  self.pid = os.fork()


INFO:pytorch_lightning.callbacks.early_stopping:Metric valid_loss improved. New best


INFO:pytorch_lightning.callbacks.early_stopping:Metric valid_loss improved by 0.034 >


INFO:pytorch_lightning.callbacks.early_stopping:Metric valid_loss improved by 0.118 >


INFO:pytorch_lightning.callbacks.early_stopping:Metric valid_loss improved by 0.137 >


INFO:pytorch_lightning.callbacks.early_stopping:Metric valid_loss improved by 0.169 >


INFO:pytorch_lightning.callbacks.early_stopping:Metric valid_loss improved by 0.126 >


INFO:pytorch_lightning.callbacks.early_stopping:Metric valid_loss improved by 0.038 >


INFO:pytorch_lightning.callbacks.early_stopping:Metric valid_loss improved by 0.014 >


INFO:pytorch_lightning.callbacks.early_stopping:Metric valid_loss improved by 0.043 >


INFO:pytorch_lightning.callbacks.early_stopping:Metric valid_loss improved by 0.004 >


INFO:pytorch_lightning.callbacks.early_stopping:Metric valid_loss improved by 0.009 >
```

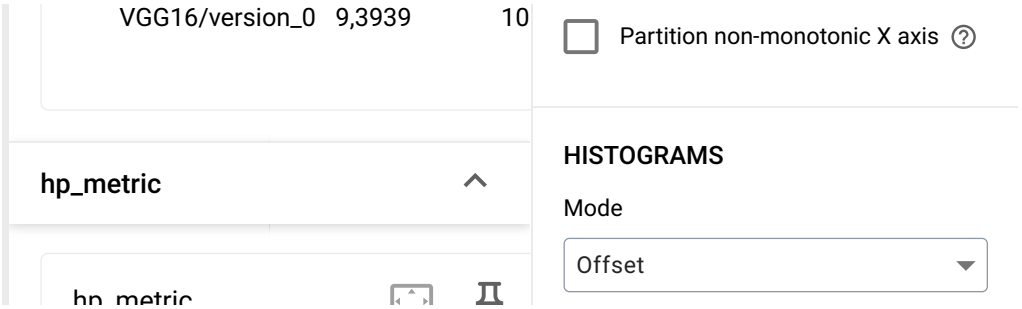
```
INFO:pytorch_lightning.callbacks.early_stopping:Metric valid_loss improved by 0.006 >
```

```
INFO:pytorch_lightning.callbacks.early_stopping:Monitored metric valid_loss did not i
```

```
%load_ext tensorboard
%tensorboard --logdir logs/
```

```
The tensorboard extension is already loaded. To reload it, use:
  %reload_ext tensorboard
Reusing TensorBoard on port 6006 (pid 21795), started 0:04:59 ago. (Use '!kill
21795' to kill it.)
```






```
trainer.test(model, test_loader)
```

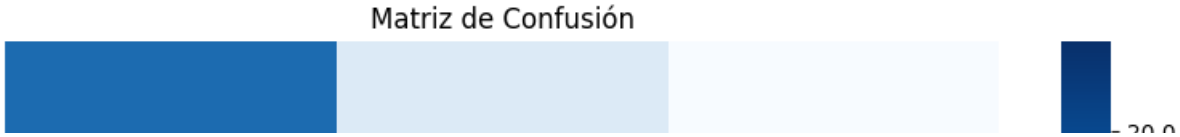
```
INFO:pytorch_lightning.utilities.rank_zero:The following callbacks returned in `Light
INFO:pytorch_lightning.accelerators.cuda:LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
```

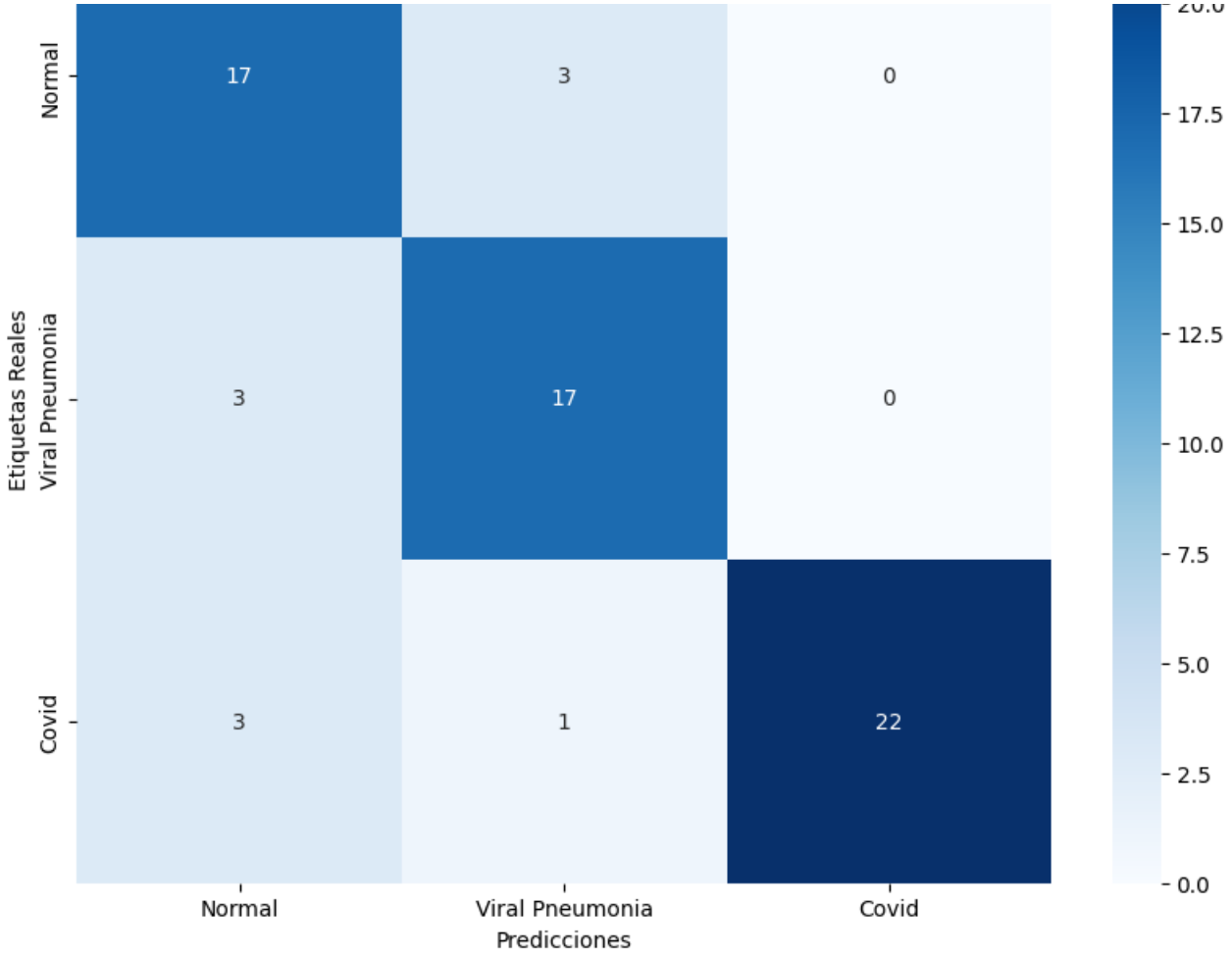
Testing DataLoader 0: 100% 1/1 [00:00<00:00, 28.33it/s]

Test metric	DataLoader 0
test_acc	0.8484848737716675

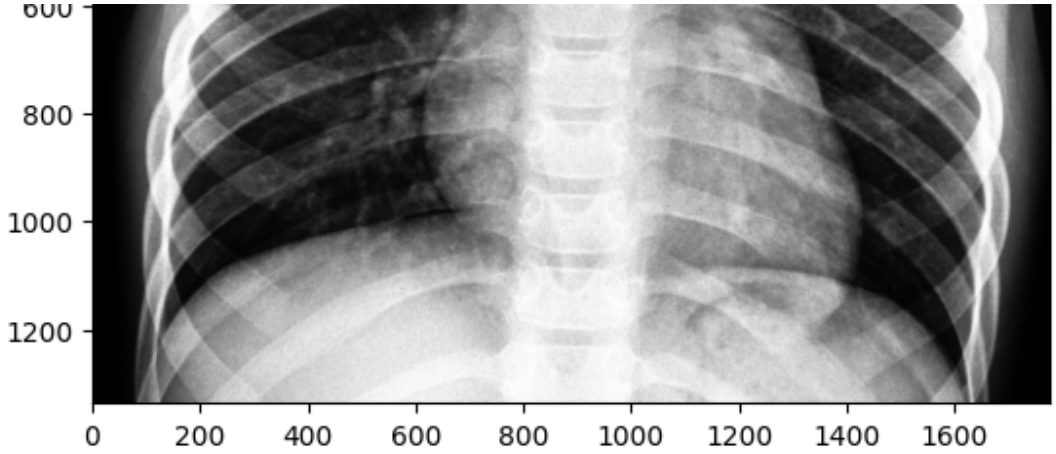
```
[{'test_acc': 0.8484848737716675}]
```

```
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np

# Obtener las predicciones y etiquetas reales de los datos de prueba
predictions = []
targets = []
model.eval()
with torch.no_grad():
    for data, target in test_loader:
        output = model(data)
        predictions.extend(output.argmax(dim=1).tolist())
        targets.extend(target.tolist())

# Calcular la matriz de confusión
conf_matrix = confusion_matrix(targets, predictions)

# Obtener los nombres de las clases del conjunto de datos original
class_names = train_dataset.dataset.classes

# Visualizar la matriz de confusión con nombres de clases
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=class_names, yticklabels=class_n
plt.xlabel('Predicciones')
plt.ylabel('Etiquetas Reales')
plt.title('Matriz de Confusión')
plt.show()
```






```
from PIL import Image


image = Image.open('/content/drive/MyDrive/TFM/Covid19-dataset/test/Normal/0116.jpeg')
plt.imshow(image, cmap='Greys')
plt.show()
```




```
image_chw = transform(image)
print(image_chw.shape)
```

```
torch.Size([1, 32, 32])
```

```
image_nchw = image_chw.unsqueeze(0)
print(image_nchw.shape)
```

```
torch.Size([1, 1, 32, 32])
```

```
with torch.no_grad():  # since we don't need to backprop
    logits = model(image_nchw)
    probas = torch.softmax(logits, dim=1)  # Calcular las probabilidades con softmax a lo largo del eje
    predicted_label = torch.argmax(probas, dim=1)  # Predecir la clase con la probabilidad más alta
```

Bibliografía

[1] Deep Learning vs Linear Regression. Towards Data Science.
https://medium.com/towards-data-science/deep-learning-vs-linear-regression-ea74aca115ea

[2] Coding Deep Learning for Beginners — Linear Regression: Part 2 (Cost Function). Towards Data Science.
https://medium.com/towards-data-science/coding-deep-learning-for-beginners-linear-regression-part-2-cost-function-49545303d29f

[3] Neural Networks Demystified. YouTube.
https://www.youtube.com/watch?v=_fDvfGxwW20

[4] Breve historia de las arquitecturas de Redes Neuronales Convolucionales (CNN). Juan Sensio.
https://juansensio.com/blog/043_cnn_arquitecturas

[5] Capsule Networks Explained. YouTube.
https://www.youtube.com/watch?v=4NgPVGt67Es

[6] Sets de entrenamiento, test y validación: ¿Cómo utilizarlos correctamente? Aprende Machine Learning.
https://www.aprendemachinelearning.com/sets-de-entrenamiento-test-validacion-cruzada/

[7] Breve historia de las redes neuronales artificiales. Steemit.
https://steemit.com/spanish/@iars.geo/breve-historias-de-las-redes-neuronales-artificiales-articulo-1

[8] Xception: Deep Learning with Depthwise Separable Convolutions. arXiv.
http://arxiv.org/abs/1702.07800

[9] The History of Deep Learning. Medium.
https://medium.com/@sreyan806/history-of-deep-learning-c176e2d3cddf

[10] Nature article on deep learning. Nature.
https://www.nature.com/articles/nature14539

[11] Types of Convolution Kernels Simplified, Toward Data Science.
https://towardsdatascience.com/types-of-convolution-kernels-simplified-f040cb307c37.

[12] Basic Overview of Convolutional Neural Network (CNN), Medium, DataSeries.
https://medium.com/dataseries/basic-overview-of-convolutional-neural-network-cnn-4fcc7dbb4f17.

[13] Understanding Conv2d and MaxPool2d Layers in PyTorch, GitHub, a2kiti.
https://github.com/a2kiti/ConvTranspose2d_for_ONNX.

[14] PyTorch Training Loop, DZLab.
https://dzlab.github.io/dl/2019/03/16/pytorch-training-loop/.