**Explicación código Red Neuronal Neumonía!**

**Datos importantes antes de entrar a fondo con la red:**

* **Contexto Médico del Proyecto**

En este proyecto, se está construyendo una red neuronal para clasificar imágenes de rayos X de tórax con el objetivo de detectar **neumonía**. Vamos a ver qué significa esto y cuáles son las diferencias entre las imágenes de rayos X normales y las de neumonía.

1. **Radiografía de Tórax Normal (Panel Izquierdo)**

* Una radiografía de tórax normal muestra pulmones **claros**, sin áreas de opacidad anormal. Esto significa que no hay signos de infección o acumulación de líquido en los pulmones.

1. **Neumonía Bacteriana (Panel del Medio)**

* La neumonía bacteriana suele manifestarse como una **consolidación lobar focal**, lo que significa que una parte específica de un pulmón muestra una opacidad más densa, que se ve como una "mancha blanca". En el caso del ejemplo mencionado, la opacidad está en el **lóbulo superior derecho**.

1. **Neumonía Viral (Panel Derecho)**

* La neumonía viral tiende a exhibir un patrón más **difuso e intersticial**, afectando ambos pulmones. Esto da lugar a un aspecto más tenue y difuso en comparación con la consolidación localizada de la neumonía bacteriana.

Imagen que contiene corbata, foto, hombre, radiografía

Descripción generada automáticamente

* El dataset

**Organización**: El dataset está organizado en tres carpetas:

* **Entrenamiento (train)**, **Validación (val)** y **Pruebas (test)**, cada una con subcarpetas para las dos categorías: **Pneumonia** y **Normal**.
* **Contenido**: Hay **5,863 imágenes** de rayos X (en formato JPEG) y las categorías son **Neumonía** y **Normal**.
  + **Origen**: Las imágenes son de pacientes pediátricos de **uno a cinco años de edad** en Guangzhou Women and Children’s Medical Center, en Guangzhou, China. Fueron tomadas como parte del cuidado clínico rutinario.
  + **Control de Calidad**: Todas las radiografías fueron revisadas para controlar la calidad, asegurándose de eliminar las imágenes de baja calidad o difíciles de interpretar. Dos expertos médicos revisaron los diagnósticos antes de usarlas para el entrenamiento del sistema, y la evaluación final fue realizada por un tercer experto.
* **Red Neuronal Convolucional (CNN) para Clasificación de Imágenes de Rayos X**

Vamos a diseñar una **Red Neuronal Convolucional (CNN)** para clasificar estas imágenes de rayos X en dos clases: **Normal** o **Neumonía**. Las redes convolucionales son ideales para procesar datos en forma de imágenes debido a su capacidad para **detectar patrones visuales**.

**¿Cómo Funciona la CNN?**

Una CNN tiene varias capas organizadas de la siguiente manera:

1. **Capas Convolucionales (Conv2D)**:
   * Son el corazón de una CNN. Estas capas aplican **filtros** (o kernels) sobre las imágenes de entrada para detectar **características locales** como bordes, texturas y otros patrones relevantes.
   * En este proyecto, las primeras capas convolucionales detectarán **patrones simples** como bordes, mientras que capas posteriores identificarán patrones más complejos (como la consolidación de la neumonía).
2. **Capas de MaxPooling (MaxPooling2D)**:
   * Las capas de **Pooling** se usan para **reducir el tamaño de las características** detectadas, manteniendo las más importantes. Esto disminuye la cantidad de información que debe ser procesada por el modelo, haciendo que sea más eficiente.
   * MaxPooling toma el valor máximo dentro de una pequeña ventana, resumiendo los resultados.
3. **Capas de Normalización y Dropout**:
   * **BatchNormalization** ayuda a que la red sea **más estable** y se entrene de manera más rápida al normalizar la activación de cada capa.
   * **Dropout** es una técnica de regularización que evita el **sobreajuste** desconectando aleatoriamente algunas neuronas durante el entrenamiento.
4. **Capas Densas (Fully Connected)**:
   * Después de las capas convolucionales, la imagen pasa a una **capa plana** (flatten) y luego a una o más **capas densas** que actúan como un clasificador.
   * La **última capa densa** tiene dos unidades de salida con activación **sigmoide** (o softmax) para clasificar las imágenes como **Neumonía** o **Normal**.

* **Arquitectura Propuesta para la CNN:**
  + **Capas Convolucionales**: Varias capas para aprender patrones de las imágenes.
  + Ejemplo: Conv2D(32, (3,3), activation='relu')
  + **Pooling**: Reducir las dimensiones de la imagen, mantener los patrones importantes.
  + **Dropout y Normalización**: Mantener el modelo robusto y evitar sobreajuste.
  + **Capas Densas**: Finalmente, usar capas densas para clasificar la imagen en una de las dos clases.
* **Optimizador y Pérdida:**
  + **Optimizador**: Puedes usar **Adam** o **Adamax**, ambos son adaptativos y eficaces para entrenar redes profundas.
  + **Función de pérdida**: Para la clasificación binaria, se suele usar la **binary\_crossentropy**, que es adecuada para problemas con dos clases.

**EN EL CASO QUE NO QUIERAS LEER POR FLOJO:**

**Resumen del Proceso:**

1. **Preprocesamiento de las Imágenes**:
   * Las imágenes se **escalarán** para normalizarlas y se aplicarán transformaciones (rotaciones, zoom) para aumentar la **variabilidad** del dataset.
2. **Entrenamiento del Modelo**:
   * Se entrenará usando las imágenes de entrenamiento (train), mientras que las imágenes de validación (val) se usarán para ajustar los hiperparámetros.
3. **Evaluación**:
   * Una vez entrenado, se evaluará con las imágenes de prueba (test) para comprobar la precisión del modelo en datos no vistos

Ahora sí, vamos a lo bueno :D.

**EXPLICACION DE CODIGO PASO A PASO PARA QUE NO TENGAS DUDAS Y NO ME ANDES PREGUNTANDO!!!!!!!!!!**

Paso 1. Importación de librerías

Texto

Descripción generada automáticamente

* **os y numpy (np):** Para manejar el sistema de archivos y cálculos numéricos.
* **pandas (pd)**: Para manipular los datos de una manera más estructurada, útil para crear el DataFrame.
* **PIL, cv2, matplotlib.pyplot (plt), y seaborn (sns):** Para trabajar con imágenes, visualizar datos, y gráficos.
* **sklearn:** Para dividir el conjunto de datos y generar métricas como la matriz de confusión.
* **tensorflow**: Para construir y entrenar un modelo de aprendizaje profundo.
* **warnings:** Suprimir las advertencias que puedan aparecer en el código.

Texto

Descripción generada automáticamente

Estas líneas configuran un par de cosas:

* **Desactivar advertencias** para evitar spam de mensajes no necesarios en el notebook.
* Configuración del estilo de visualización de gráficos usando seaborn para hacerlos más estéticos.

Después hay un mensaje de “modules load” que nomas dice que todo estuvo correcto pero es como que no importante ps

Texto

Descripción generada automáticamente

En lugar de descargar la BD que pesa como 2 GB es mejor importarla directo desde Kaggle jeje

* **dataset\_download()**: Este método descarga el dataset y devuelve la ruta donde se guardan los archivos.



Se define una **ruta base** para acceder a los datos descargados. Es importante tener una referencia para el resto del código…

Texto

Descripción generada automáticamente con confianza media

En este punto:

* Se verifica si la ruta del dataset **existe** usando os.path.exists().
* Si existe, **listamos** los archivos que hay dentro para entender la estructura.
* Si no existe, lanzamos un **error** (FileNotFoundError) con un mensaje informativo.



Se define la ruta específica de los datos de **entrenamiento**, que normalmente estarán en una carpeta llamada "train"

Texto

Descripción generada automáticamente

Esta es la ultima parte de la primera parte del código, no es como taaal de suma importancia pero nunca está de mas hacerlo.

* Se verifica la existencia del directorio de entrenamiento.
* Se crean listas vacías para almacenar las rutas de los archivos (filepaths) y las etiquetas (labels).
* Listo las carpetas dentro del directorio de entrenamiento (Normal y Pneumonia). Estas carpetas representan las clases del problema: imágenes de personas sanas y de personas con neumonía.
* Se itera sobre cada imagen en las carpetas y guardo:
  + La ruta completa del archivo en filepaths.
  + La etiqueta (Normal o Pneumonia) en labels.

Luego, se crea un DataFrame con pandas usando las listas filepaths y labels para facilitar el manejo de datos en los siguientes pasos.

Ya por ultimo nomas pa’ estar seguros se imprime las primeras filas del DataFrame para verificar que los datos se hayan cargado correctamente jeje (uno nunca sabe)

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

* df: Al ejecutar df, simplemente se muestra el contenido del DataFrame para inspeccionar las imágenes de entrenamiento y ver cómo se organizan.
* df['label'].value\_counts(): Esta línea cuenta la cantidad de imágenes de cada clase (Normal o Pneumonia) en los datos de entrenamiento. Esto es muy útil para verificar si el dataset está balanceado (cantidad similar de imágenes en ambas clases). Un dataset muy desbalanceado puede resultar en un modelo que esté sesgado hacia la clase con más ejemplos.

Texto

Descripción generada automáticamente

**Definición de la Ruta**:

* Se define valid\_data\_dir que apunta a la carpeta con los datos de validación (val). Estos datos se usan durante el entrenamiento para **ajustar** el modelo y ayudar a detectar problemas como sobreajuste.

**Verificación de Existencia**:

* Se verifica que el directorio exista para asegurarte de que no haya errores en la estructura del dataset. Si no existe, se lanza un error informativo (FileNotFoundError).

**Recorrido y Creación de Listas**:

* **Itero sobre las carpetas** de Normal y Pneumonia dentro de val.
* Para cada imagen, se guarda:
  + La **ruta completa** (filepaths) para que el modelo pueda acceder al archivo.
  + La **etiqueta** (labels) que indica si la imagen corresponde a un caso Normal o Pneumonia.

**Crear un DataFrame**:

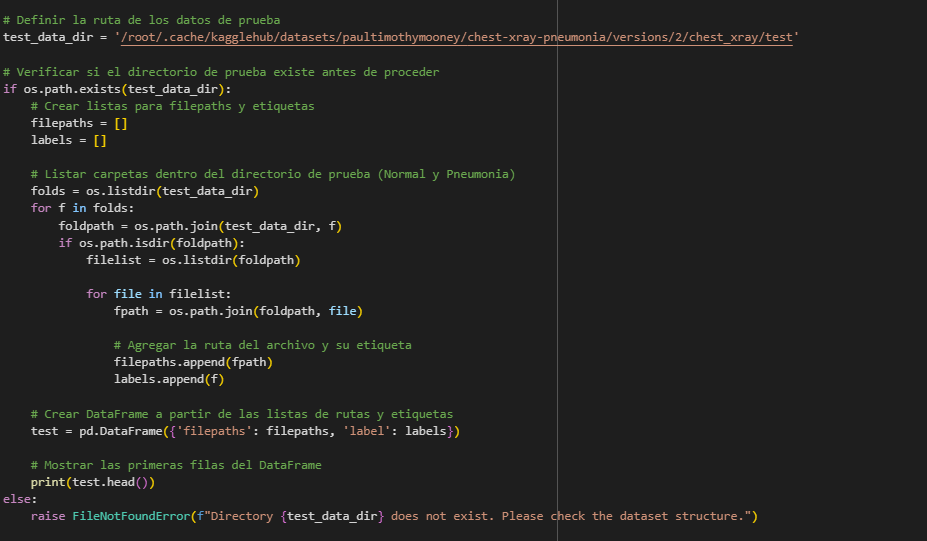
* Se utiliza pandas para crear un DataFrame llamado valid, que contiene dos columnas: filepaths (rutas de imágenes) y label (etiqueta de la clase).
* Esto hace que sea mucho más fácil trabajar con los datos durante el entrenamiento del modelo.

**Mostrar las Primeras Filas**:

* Se imprime las primeras filas del DataFrame para verificar que los datos se han estructurado correctamente. (tampoco es tan importante)



Esta línea te ayuda a ver cuántas imágenes tienes de cada clase (Normal o Pneumonia) en el conjunto de validación



Este bloque organiza los **datos de pruebas**, de una manera muy similar a los datos de validación.

1. **Definir la Ruta**:
   * test\_data\_dir apunta a la carpeta de datos de prueba (test). Estos datos **nunca** se usan durante el entrenamiento y solo se usan al final para evaluar la **eficacia real** del modelo.
2. **Verificación de Existencia**:
   * Verificas que la carpeta test exista.
3. **Recorrer y Crear Listas**:
   * Al igual que con los datos de validación, recorres las carpetas de Normal y Pneumonia, y guardas las rutas y etiquetas en listas.
4. **Crear un DataFrame de Pruebas**:
   * **test** es un DataFrame que contiene las imágenes de prueba y sus etiquetas correspondientes. Estas imágenes se usarán para verificar qué tan bien generaliza el modelo cuando se enfrenta a datos completamente nuevos.

**Importancia de Tener Train, Validation y Test (de suma importancia stupid)**

1. **Entrenamiento (train)**:
   * Estas imágenes se usan para entrenar el modelo, es decir, para enseñarle los patrones que debe reconocer.
2. **Validación (val)**:
   * Este conjunto se usa durante el entrenamiento para monitorear el rendimiento del modelo y ajustar los hiperparámetros, como la tasa de aprendizaje. Ayuda a detectar si el modelo está aprendiendo correctamente o si está **sobreajustándose** (aprender demasiado bien los datos de entrenamiento sin poder generalizar).
3. **Pruebas (test)**:
   * Este conjunto de imágenes se utiliza **al final** para medir el rendimiento real del modelo. Nos da una idea de cómo se comportará el modelo en la práctica, con datos nuevos.

AHORA SÍ, LLEGADO A ESTE PUNTO NETA LEE PORQUE AQUÍ EN ADELANTE SE EMPIEZA A PONER SERIA LA COSA!!!!!! EN ESTA SECCION NOS VAMOS A ENFOCAR AHORA SÍ EN PREPARAR LOS DATOS PARA UTILIZARLOS EN LA RED NEURONAL :D



**train\_test\_split() de sklearn nos permite dividir los datos en dos subconjuntos.**

* Aquí se usa para dividir el DataFrame df (que contiene las rutas y etiquetas de las imágenes) en **entrenamiento** y **otro subconjunto temporal (dummy\_df)**.
* **train\_size=0.8** significa que el 80% de los datos serán para entrenamiento.
* Luego se toma el dummy\_df y se divide en **validación** (valid\_df) y **pruebas** (test\_df), con una proporción **50-50**. Esto significa que cada uno de estos conjuntos contendrá el **10%** del total original de los datos (el 20% que queda se divide en dos).
* **shuffle=True** indica que los datos se barajan antes de dividirlos para asegurar que estén distribuidos aleatoriamente.
* **random\_state=42** se usa para garantizar **reproducibilidad**: el mismo valor siempre generará la misma división de los datos.



* **batch\_size** define cuántas imágenes se procesarán **simultáneamente** durante el entrenamiento. En este caso, será de 16.
* **img\_size** es el tamaño al que se redimensionarán las imágenes: **224 x 224** píxeles, lo cual es común para modelos de visión por computadora

Texto

Descripción generada automáticamente

* **ImageDataGenerator** de tensorflow.keras.preprocessing.image se usa para crear generadores que cargan imágenes **en lotes**. (**cuando veas el código te vas a dar cuenta porque se usa)**
* tr\_gen, ts\_gen, y val\_gen son instancias de ImageDataGenerator. Cada una se asociará a un conjunto de datos (entrenamiento, pruebas y validación).

Texto

Descripción generada automáticamente

**Configuración del generador:**

**flow\_from\_dataframe()** se utiliza para cargar las imágenes y sus etiquetas a partir de un DataFrame. Esto se hace en **batches** para reducir la carga de memoria.

* **train\_df** es el DataFrame de entrenamiento que se usa para cargar las imágenes.
* **x\_col='filepaths'** y **y\_col='label'** especifican qué columnas usar para las rutas de las imágenes (filepaths) y las etiquetas (label).
* **target\_size=img\_size** redimensiona las imágenes a **224 x 224 píxeles**.
* **class\_mode='categorical'** indica que las etiquetas son **categóricas** (es decir, una clase entre varias posibles, Normal o Pneumonia).
* **color\_mode='rgb'** carga las imágenes en **color** (3 canales de color: rojo, verde, azul).
* **shuffle=True** asegura que las imágenes se carguen en **orden aleatorio**, lo cual es beneficioso durante el entrenamiento.

**valid\_gen** y **test\_gen** funcionan igual que train\_gen, pero se aplican a los conjuntos de **validación** (valid\_df) y **pruebas** (test\_df).

La única diferencia es que, para **test\_gen**, se usa **shuffle=False**, porque durante la prueba no necesitamos que los datos estén en un orden aleatorio; queremos que la evaluación sea consistente.

Texto

Descripción generada automáticamente

**class\_indices** es un atributo del generador que devuelve un **diccionario** donde las clases (Normal y Pneumonia) se asocian a un índice numérico.

**list(gen\_dict.keys())** convierte las claves del diccionario (Normal y Pneumonia) en una **lista** llamada classes que usaremos más adelante para la visualización.

**next(train\_gen)** se usa para obtener un **batch de imágenes y etiquetas** del generador. Cada batch tendrá **16 imágenes** (como se definió en batch\_size).

**images** y **labels** son matrices de datos que contienen las imágenes redimensionadas y sus etiquetas correspondientes.

Lo demás es solo para mostrar las imágenes de entrenamiento, para entender mejor el dataset y verificar que las clases estén correctamente etiquetadas

**AHORA SI AGARRATE QUE AHORA SE VIENE LA CREACION Y EL ENTRENAMIENTO DE LA RED NEURONAL (CNN) !!!!!!**

Texto

Descripción generada automáticamente

No te asustes, aquí te va la explicación jeje

* **create\_cnn\_model()** es una función que define la arquitectura de la CNN.
* **input\_shape=(224, 224, 3)** indica que el tamaño de entrada de las imágenes será de **224 x 224 píxeles** y tendrá **3 canales de color** (RGB).
* **models.Sequential()**: Se usa un modelo secuencial, lo cual significa que las capas serán **apiladas una después de otra**, como una serie de pasos consecutivos.

**PRIMER BLOQUE DE CAPAS DEL MODELO:**

 **Primera Bloque de Capas Convolucionales**:

* **layers.Conv2D(32, (3, 3), activation='relu', padding='same', input\_shape=input\_shape)**:
  + Se añade una **capa convolucional** con **32 filtros**, tamaño de kernel de **3x3**.
  + **activation='relu'** aplica una función de activación ReLU (Rectified Linear Unit) para introducir no linealidad.
  + **padding='same'** asegura que el tamaño de la salida de la capa convolucional sea el mismo que el de la entrada.
  + **input\_shape=input\_shape** se aplica solo a la primera capa para definir el tamaño de la entrada.
* **Otra Conv2D** similar sigue después para capturar más características.

**Capa de MaxPooling**:

* **layers.MaxPooling2D((2, 2))** reduce el tamaño de la imagen a la mitad (reducción de **dimensiones**), manteniendo solo la información más importante. Aquí se aplica un "filtro de tamaño **2x2**", seleccionando el valor máximo en cada ventana.

**Capa de Dropout**:

* **layers.Dropout(0.25)** desactiva aleatoriamente el **25%** de las neuronas durante cada paso del entrenamiento, lo cual ayuda a prevenir el **sobreajuste** (memorizar los datos de entrenamiento).

En cada repetición, **se aumenta los filtros** a **64**, lo cual le permite a la red aprender **patrones más complejos**. También se incrementa la cantidad de **Dropout** en un **40%**, para asegurar que las capas posteriores no aprendan en exceso.

**Capa Flatten**:

* **model.add(layers.Flatten())** convierte la salida de las capas anteriores (que tienen una estructura bidimensional) en un **vector unidimensional** para pasarlo a la capa completamente conectada.

**Capa Densa**:

* **layers.Dense(256, activation='relu')** es una **capa densa** (también conocida como "fully connected"), que tiene **256 neuronas**. Esta capa toma todas las características previamente extraídas y trata de combinarlas de manera útil para la clasificación.

**Dropout**:

* **layers.Dropout(0.5)** elimina aleatoriamente **el 50%** de las neuronas para prevenir el sobreajuste.

**Capa de Salida**:

* **layers.Dense(2, activation='softmax')**:
  + Es la **capa de salida** con **2 neuronas**, ya que tenemos dos clases (Normal y Pneumonia).
  + **activation='softmax'** genera probabilidades para cada clase, asegurando que la suma de las probabilidades sea **1**.

**Por ultimo:**

**create\_cnn\_model()** crea la CNN con la arquitectura definida.

**model.summary()** imprime un **resumen del modelo**, mostrando la cantidad de capas, parámetros entrenables y la forma de las salidas en cada capa.



**Compilación del Modelo**:

* **Adamax(learning\_rate=0.001)** es una variante del optimizador Adam. Es adecuado para este tipo de modelos porque ajusta dinámicamente el tamaño del paso durante el entrenamiento.
* **loss='categorical\_crossentropy'** se usa porque se trata de un problema de clasificación con más de una clase.
* **metrics=['accuracy']**: La métrica de evaluación seleccionada es la **precisión**, que indica qué tan bien el modelo clasifica correctamente las imágenes.



**Número de Épocas**:

* **epochs = 15** indica que el modelo entrenará durante **15 iteraciones completas** sobre el conjunto de datos de entrenamiento. Cada época significa que todas las imágenes del conjunto de entrenamiento han sido utilizadas una vez.

**Entrenar el Modelo**:

* **model.fit()** se usa para entrenar el modelo.
* **train\_gen** es el generador de imágenes de entrenamiento que proporciona las imágenes en lotes.
* **epochs=epochs** indica cuántas épocas entrenará.
* **verbose=1** proporciona una descripción detallada del progreso del entrenamiento.
* **validation\_data=valid\_gen** especifica el generador de validación, lo cual permite evaluar el rendimiento del modelo en los datos de validación durante cada época.
* **shuffle=False** indica que las imágenes no se barajarán en cada época, algo que usualmente se hace en el generador.

Texto

Descripción generada automáticamente

**history.history** contiene los valores registrados durante el entrenamiento para cada época.

**tr\_acc y tr\_loss** almacenan la **precisión** y la **pérdida** del entrenamiento respectivamente.

**val\_acc y val\_loss** almacenan la **precisión** y la **pérdida** en el conjunto de **validación**. Estos valores permiten ver cómo se comporta el modelo en datos no vistos durante el entrenamiento.

**np.argmin(val\_loss)**: Encuentra el **índice de la pérdida mínima** en el conjunto de validación. Este es el momento en el que el modelo tenía el menor valor de pérdida, lo cual generalmente significa el mejor ajuste.

**val\_lowest = val\_loss[index\_loss]**: Obtiene el **valor más bajo de la pérdida** de validación.

**np.argmax(val\_acc)**: Encuentra el **índice de la precisión más alta** en el conjunto de validación.

**acc\_highest = val\_acc[index\_acc]**: Obtiene el **valor más alto de la precisión** en el conjunto de validación.

Lo demás ya lo deberías de saber jeje igual lo ultimo del código que no esta explicado no creo que lo necesites la verdad, creo que hasta aquí sería una buena explicación para tu tarea, es todo gracias like y sub