Compás
capacitación e investigación

Guía Avanzada de Deep Learning para profesionales de la IA

Wilmer B. Rivas Asanza
Bertha E. Mazón Olivo
Joffre J. Cartuche Calva

Compás
capacitación e investigación

Bertha E. Mazón Olivo
Joffre J. Cartuche Calva

Primera edición 2025-11-09

ISBN: 978-9942-53-028-8
DOI: http://doi.org/10.48190/9789942530288
Distribución online
Acceso abierto

Cita
Rivas, W., Mazón, B., Cartuche, J. (2025) Guía Avanzada de Deep Learning para profesionales de la IA. Editorial Grupo Compás

Este libro es parte de la colección de la Univesidad Técnica de Machala y ha sido debidamente examinado y valorado en la modalidad doble par ciego con fin de garantizar la calidad de la publicación.

DEDICATORIA

A nuestras familias, especialmente a nuestras parejas e hijos, por su constante apoyo, paciencia y comprensión durante cada etapa de este proceso. Su respaldo ha sido esencial para la concreción de esta obra.

A NUESTROS ESTUDIANTES DE LA CARRERA DE TECNOLOGÍAS DE LA INFORMACIÓN, CIENCIA DE DATOS E INTELIGENCIA ARTIFICIAL, QUIENES SE INICIAN EN EL ESTUDIO RIGUROSO Y APLICADO DEL *DEEP LEARNING*. QUE ESTE LIBRO SIRVA COMO UNA GUÍA SÓLIDA EN SU FORMACIÓN ACADÉMICA, IMPULSANDO EL PENSAMIENTO CRÍTICO, LA INNOVACIÓN Y EL COMPROMISO CON EL DESARROLLO TECNOLÓGICO.

Y a todos quienes promueven la educación y la generación de conocimiento como motores de transformación social y científica.

INTRODUCCIÓN DEL LIBRO

El ***Deep Learning*** o aprendizaje profundo ha transformado el campo de la inteligencia artificial, permitiendo avances sorprendentes en tareas como visión por computadora, reconocimiento facial, procesamiento de lenguaje natural y detección automática de objetos. Este paradigma se basa en redes neuronales profundas capaces de aprender representaciones jerárquicas a partir de grandes volúmenes de datos, superando ampliamente los métodos tradicionales en precisión y eficiencia.

Este libro tiene como propósito guiar al lector en la comprensión teórica y práctica de las principales técnicas y modelos de ***Deep learning*** aplicados a la visión artificial. Está organizado en cuatro capítulos que combinan fundamentos, implementaciones en código y casos de estudio reales, permitiendo un aprendizaje progresivo y aplicado.

El texto está dirigido a estudiantes, docentes y profesionales de carreras relacionadas con la ciencia de datos, la informática, la inteligencia artificial o la ingeniería, que deseen adquirir una base sólida en ***Deep learning*** desde un enfoque práctico, claro y actualizado.

OBJETIVO DEL LIBRO

Brindar una formación teórico-práctica en Deep learning enfocada en visión por computadora, mediante el estudio de redes neuronales convolucionales, modelos preentrenados, detección de objetos y reconocimiento facial, desarrollando competencias para diseñar, entrenar y evaluar modelos aplicados a problemas reales.

ESTRUCTURA GENERAL DEL LIBRO

A continuación, se presenta un resumen de lo que se tratará en cada capítulo del libro:

CAPÍTULO 1: REDES NEURONALES CONVOLUCIONALES

Este capítulo introduce los fundamentos de las redes neuronales convolucionales (CNN), una de las arquitecturas más utilizadas en tareas de visión por computadora. Se explican de manera detallada las capas clave que conforman una CNN, incluyendo las capas convolucionales, de activación, de agrupamiento y completamente conectadas. A través de un caso de estudio centrado en la clasificación de imágenes de perros y gatos, se guía al lector en el proceso completo de construcción, entrenamiento y evaluación de una CNN desde cero, utilizando un enfoque práctico con Python.

CAPÍTULO 2: MODELOS PREENTRENADOS Y TRANSFER LEARNING

En este capítulo se aborda el uso de modelos preentrenados y la técnica de ***transfer learning***, una estrategia eficiente para aplicar el conocimiento de redes previamente entrenadas a nuevos problemas. Se explican las diferencias entre la extracción de características y el ajuste fino (***fine-tuning***), así como las ventajas de emplear arquitecturas como VGG, ResNet o MobileNet. El caso práctico muestra cómo adaptar un modelo preentrenado a un nuevo conjunto de datos, comparando su rendimiento con un modelo entrenado desde cero.

CAPÍTULO 3: DETECCIÓN DE OBJETOS CON YOLOV8

Este capítulo se centra en la detección de objetos, introduciendo la arquitectura YOLOv8, una de las más precisas y rápidas en la actualidad. Se analizan los conceptos fundamentales como las cajas delimitadoras (***Bounding boxes***), la puntuación de confianza y las métricas de evaluación como el mAP. El lector aprenderá a entrenar un modelo de detección

personalizado utilizando YOLOv8 y a interpretar los resultados obtenidos en un caso de estudio aplicado a escenas urbanas.

Capítulo 4: Reconocimiento Facial con Modelos de Deep Learning

El capítulo final explora el campo del reconocimiento facial, explicando cómo se generan representaciones vectoriales (***embeddings***) y se comparan utilizando métricas de similitud. Se presentan modelos como FaceNet y ArcFace, resaltando sus aplicaciones y desafíos. Además, se desarrolla un caso práctico de reconocimiento de rostros y se construye una aplicación web funcional con Flask, que permite capturar imágenes desde una cámara y procesarlas en tiempo real mediante un modelo entrenado.

Estrategias metodológicas

Este libro ha sido concebido con un enfoque teórico-práctico, orientado tanto a la comprensión conceptual de los modelos de ***Deep Learning*** como a su aplicación en escenarios reales. Las estrategias metodológicas empleadas se alinean con una enseñanza activa y basada en competencias, permitiendo al lector no solo adquirir conocimientos, sino también desarrollar habilidades para la implementación efectiva de soluciones con inteligencia artificial.

Las principales estrategias utilizadas son:

- **Aprendizaje basado en proyectos (ABP):** Cada capítulo incluye un caso de estudio práctico que permite al lector aplicar los conceptos aprendidos en contextos reales, como clasificación de imágenes, detección de objetos y reconocimiento facial.
- **Desarrollo progresivo de contenidos:** Se parte desde conceptos fundamentales (como redes convolucionales) hacia aplicaciones más complejas (como YOLOv8 y modelos preentrenados), asegurando una construcción gradual del conocimiento.
- **Codificación práctica con Python:** Se emplean bibliotecas estándar como Tensor Flow, Keras, PyTorch y Ultralytics, permitiendo a los lectores implementar modelos desde cero y adaptarlos a sus necesidades.
- **Transferencia al entorno web:** Como parte del aprendizaje aplicado, se integra el desarrollo de una aplicación web con Flask, permitiendo experimentar cómo los modelos de Deep Learning pueden integrarse en sistemas interactivos de uso real.

- **Evaluación e interpretación de resultados:** En cada proyecto se analizan métricas de rendimiento, lo que fomenta el pensamiento crítico y la capacidad de mejora continua de los modelos.
- **Uso de recursos abiertos y replicables:** Se han seleccionado datasets públicos y herramientas de código abierto para facilitar el acceso y la reproducibilidad de los experimentos propuestos.

Estas estrategias buscan que el lector no solo entienda cómo funcionan los modelos de Deep Learning, sino que esté en capacidad de construir, adaptar y desplegar soluciones inteligentes en su entorno académico o profesional.

Indice

INDICE DE TABLAS

INDICE DE ILUSTRACIONES

CAPÍTULO 1: REDES NEURONALES CONVOLUCIONALES (CNN) CON KERAS

RESUMEN DEL CAPÍTULO

En este capítulo se introduce el concepto de redes neuronales convolucionales (CNN), uno de los pilares del Deep learning aplicado a visión por computadora. Se analiza su arquitectura, capa por capa, y se explican de forma didáctica los mecanismos de convolución, activación, agrupamiento y conexión total. A través de un caso práctico de clasificación de imágenes de perros y gatos, se muestra cómo entrenar una CNN desde cero, aplicar técnicas de mejora como aumento de datos y evaluar su rendimiento.

INTRODUCCIÓN

Las redes neuronales convolucionales han revolucionado la forma en que las máquinas interpretan imágenes, permitiendo avances sin precedentes en campos como el reconocimiento facial, los vehículos autónomos o el diagnóstico médico. En este capítulo, aprenderás a comprender su estructura interna, cómo extraen características relevantes de una imagen y cómo se entrenan para tareas específicas. A través de un enfoque práctico, construiremos una CNN para distinguir entre imágenes de perros y gatos, uno de los problemas más clásicos y accesibles para iniciarse en la visión por computadora.

PREGUNTAS DE ENFOQUE

- ¿Qué elementos componen una red neuronal convolucional?
- ¿Cómo se procesan las imágenes a través de las capas de una CNN?
- ¿Qué rol cumplen las operaciones de convolución, activación y pooling?
- ¿Cómo puede una CNN aprender a clasificar imágenes?
- ¿Qué técnicas pueden mejorar el rendimiento del modelo?
- ¿Qué desafíos comunes se enfrentan en la clasificación de imágenes?

OBJETIVOS DEL CAPÍTULO

- Comprender la arquitectura y funcionamiento de las redes convolucionales.
- Identificar el propósito de cada tipo de capa dentro de una CNN.
- Desarrollar una red CNN básica utilizando un conjunto de imágenes.
- Implementar un caso práctico de clasificación binaria (perros vs. gatos).
- Evaluar el desempeño del modelo con métricas de precisión y pérdida.

RESULTADOS DE APRENDIZAJE ESPERADOS

Al finalizar este capítulo, el lector será capaz de:

- Explicar el funcionamiento de las redes convolucionales en problemas de visión.
- Diseñar e implementar una CNN básica con Keras/Tensor Flow.
- Aplicar técnicas de aumento de datos y regularización para mejorar el rendimiento.
- Interpretar los resultados obtenidos durante el entrenamiento y evaluación.
- Enfrentar problemas reales de clasificación de imágenes con redes profundas.

PROBLEMAS A RESOLVER

- ¿Cómo se puede diseñar una arquitectura adecuada para clasificación de imágenes?
- ¿Qué se debe hacer cuando el modelo sobreentrena?
- ¿Cómo abordar el problema de clases desbalanceadas?
- ¿Qué hacer si las imágenes presentan ruido o baja calidad?
- ¿Cómo mejorar la generalización sin aumentar la complejidad?

DEFINICIÓN

Las **Redes Neuronales Convolucionales (Convolutional Neural Networks, CNN)** son un tipo de red neuronal profunda diseñada específicamente para **procesar datos con estructura espacial**, como imágenes, videos o señales. Se caracterizan por el uso de **capas convolucionales** que aprenden automáticamente a detectar patrones locales como bordes, texturas o formas, simulando el comportamiento de las neuronas visuales en el córtex del cerebro humano (Khan et al., 2020; Sharma & Jain, 2022).

¿PARA QUÉ SIRVEN LAS CNN?

Las CNN permiten que las computadoras **"vean" e interpreten imágenes** mediante el aprendizaje automático. Sus aplicaciones principales incluyen:

- **Clasificación de imágenes** (por ejemplo, distinguir gatos de perros)
- **Detección de objetos** (reconocer personas, autos, rostros)

- **Segmentación semántica** (resaltar regiones específicas de una imagen)
- **Reconocimiento facial y biometría**
- **Diagnóstico médico asistido por IA** (análisis de radiografías, resonancias, etc.)
- **Visión para robots y vehículos autónomos**
- **Reconocimiento de escritura, placas o gestos**.

Estas aplicaciones han sido ampliamente exploradas en los últimos años, demostrando la versatilidad de las CNN en distintos dominios (Abiodun et al., 2021; Alzubaidi et al., 2021).

TRABAJOS PREVIOS Y APLICABILIDAD EN LA INDUSTRIA

Las **redes neuronales convolucionales secuenciales (CNN)** son arquitecturas emblemáticas en visión artificial. Se estructuran en capas que transforman progresivamente los datos: capas de **convolución** capaces de extraer patrones locales mediante filtros entrenables; capas de **pooling** que reducen la dimensionalidad preservando características esenciales; y capas **fully connected** encargadas de consolidar la representación para tareas de clasificación (Yamashita et al., 2018). Múltiples investigaciones recientes han demostrado la efectividad de las CNN en entornos industriales, desde la inspección visual automatizada en manufactura hasta la detección de defectos o anomalías (Dai et al., 2023; FlatWorld Solutions, 2025). En el ámbito médico, las CNN han sido aplicadas con éxito para clasificación de enfermedades, localización de estructuras anatómicas y mejora de imagen (Jia et al., 2024; Li et al., 2025), lo que permite comprender cómo estos modelos combinan fundamentos arquitectónicos sólidos con una amplia aplicabilidad práctica.

LÍNEA DE TIEMPO DE LAS REDES NEURONALES CONVOLUCIONALES

Tabla 1. Línea de tiempo de las principales arquitecturas de redes neuronales convolucionales, desde perceptron (1958) hasta efficientNet (2023)

Año	Hito	Descripción
1958	Perceptrón (Rosenblatt)	Primer modelo formal de una neurona artificial.
1980	Neocognitron (Fukushima)	Arquitectura pionera que inspiró las CNN modernas. Introdujo capas convolucionales y de pooling.
1998	LeNet-5	CNN para reconocimiento de dígitos manuscritos. Usada en cheques bancarios.
2012	AlexNet	Revoluciona el campo al ganar el concurso ImageNet con una CNN profunda y GPU. Abre la era del Deep Learning.
2014	VGGNet	Propone usar muchas capas pequeñas (3×3). Arquitectura simple pero poderosa.
2015	ResNet	Introduce conexiones residuales. Permite entrenar redes mucho más profundas (hasta 152 capas).
2017	MobileNet	CNN ligera diseñada para dispositivos móviles. Usa convoluciones separables en profundidad.
2019–2023	EfficientNet, ConvNeXt, Visión Transformers	Avances híbridos que optimizan precisión, velocidad y escalabilidad. Combinan CNN y mecanismos de atención.

FUENTE: (TAN & LE, 2019; LIU ET AL., 2022)

FUNCIONAMIENTO PASO A PASO DE UNA RED NEURONAL CONVOLUCIONAL (CNN)

REPRESENTACIÓN DE LA IMAGEN COMO MATRIZ

Primero, la imagen (por ejemplo, una foto de un gato) se convierte en una **matriz de números**. Cada valor representa la **intensidad de píxeles**:

- Si es **blanco y negro**, la matriz tiene una sola capa (escala de grises), una matriz 2D (como 6x6).

Ilustración 1. Representación de una imagen en escala de grises (blanco y negro), donde cada píxel se codifica con un valor de intensidad luminosa entre 0 (negro) y 255 (blanco), lo que permite visualizar la información sin componentes de color.
FUENTE: ELABORACIÓN PROPIA

- Si es **a color (RGB)**, se representa con **tres matrices**, una por canal: rojo, verde y azul formando un tensor 3D: 6**x6x3** Cada valor numérico representa la intensidad de un píxel.

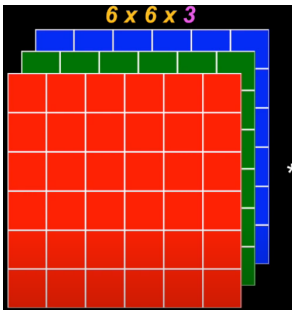


Ilustración 2. Representación de una imagen a color mediante el modelo RGB, donde cada píxel se codifica con tres valores que corresponden a la intensidad de los canales rojo, verde y azul. La combinación de estos tres componentes permite reconstruir la imagen con sus tonalidades y matices originales.
FUENTE: ELABORACIÓN PROPIA

***DEFINICIÓN DEL** KERNEL O FILTRO*

Un **kernel** (o filtro) es una pequeña matriz de valores numéricos, comúnmente de tamaño **3x3 o 5x5 (la dimensión hay que definirla)**, cuya dimensión debe ser **menor que la de la imagen de entrada**. Al diseñar una red convolucional, también se define **cuántos filtros se utilizarán**, ya que **cada filtro aprende a detectar una característica distinta** de la imagen, como bordes, texturas, esquinas o contornos (Alzubaidi et al., 2021; Sharma & Jain, 2022).

El kernel puede imaginarse como una lupa inteligente que recorre la imagen, extrayendo características específicas como bordes, texturas o esquinas (Khan et al., 2020). Al superponerse sobre distintas regiones, el filtro resalta los patrones visuales más relevantes, tal como se ilustra a continuación.

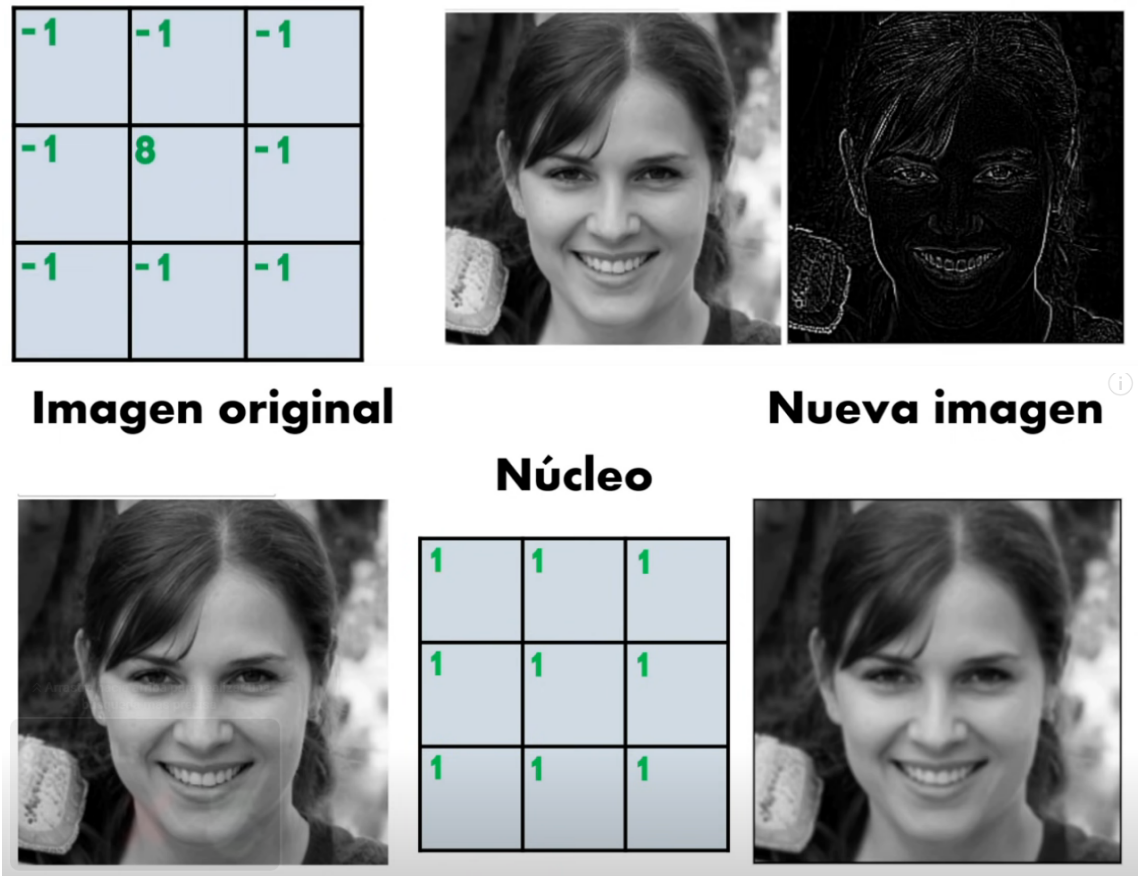


Ilustración 3. Ejemplo del proceso de convolución en una red neuronal: se aplica un núcleo (filtro o máscara) sobre la imagen original y se obtiene como resultado una nueva imagen (mapa de características) que resalta los patrones o bordes detectados.

FUENTE: ELABORACIÓN PROPIA

Definir stride y padding

Stride: Es cuántos pasos se mueve el kernel cada vez.

- Stride 1: se mueve 1 píxel.
- Stride 2: se mueve 2 píxeles (reduce más la imagen).

Padding en Redes Convolucionales

Padding es una técnica que consiste en **agregar filas y columnas de ceros (u otro valor constante)** alrededor de la imagen o matriz de entrada antes de aplicar la operación de convolución.

🔹 Tipos de Padding

- **Valid**: sin padding, lo que reduce el tamaño de salida.
- **Same**: añade padding suficiente para que la salida conserve las mismas dimensiones espaciales que la entrada.

🎯 ¿Para qué sirve el Padding?

El objetivo principal del padding es:

1. **Preservar el tamaño** de la imagen después de aplicar convoluciones.
2. **Evitar pérdida de información**, especialmente en los bordes.
3. **Permitir redes más profundas**, al mantener tamaños intermedios más grandes por más tiempo.

✅ Beneficios del Padding

Beneficio	Descripción
Conserva dimensiones	El padding 'same' permite mantener el ancho y alto originales de la imagen.
Protege la información de bordes	Sin padding, los píxeles de los bordes son menos procesados; con padding, se integran mejor.
Controla el tamaño de salida	Puedes ajustar la dimensión resultante de cada capa convolucional.
Mejora la profundidad del modelo	Al evitar una reducción rápida de tamaño, permite usar más capas convolucionales.

Ilustración 4. Beneficios del padding en convoluciones: permite mantener las dimensiones originales de la imagen tras aplicar el núcleo, evita la pérdida de información en los bordes y facilita que el filtro se desplace uniformemente sobre toda la imagen.
FUENTE: ELABORACIÓN PROPIA

¿CÓMO SE EXTRAEN CARACTERÍSTICAS O PATRONES?
Al aplicar un filtro a la imagen, se realiza una operación de convolución: el kernel se superpone sobre una región específica, multiplica sus valores con los píxeles correspondientes y luego los suma para generar un único valor. Este resultado indica la intensidad con la que una característica (como un borde o textura) está presente en esa zona.

Al desplazarse por toda la imagen, se genera una nueva matriz llamada mapa de características (feature map), que refleja la activación del filtro en distintas regiones. Este desplazamiento se realiza por defecto de un píxel a la vez, y ese paso se denomina stride. Al incrementar el stride (por ejemplo, a 2), el filtro salta más posiciones, lo cual reduce el tamaño del mapa de salida.

El siguiente ejemplo ilustra si el stride=2

🎯 **Supuestos**

- **Imagen de entrada**: una matriz 5x5
- **Filtro (kernel)**: tamaño 3x3
- **Stride**: 2
- **Padding**: sin padding

a. Imagen de entrada (5x5)

[1, 2, 3, 0, 1]
[4, 5, 6, 1, 2]
[7, 8, 9, 0, 1]
[1, 3, 5, 7, 9]
[0, 2, 4, 6, 8]

b. Filtro (3x3)

[1, 0, -1]
[1, 0, -1]
[1, 0, -1]

Este filtro puede usarse, por ejemplo, para detectar bordes verticales.

c. Aplicación con stride = 2

Con stride 2, el filtro se **mueve de dos en dos** sobre la imagen (tanto horizontal como verticalmente), reduciendo la cantidad de pasos posibles.

👉 *POSICIONES VÁLIDAS DEL FILTRO:*

1. Fila 0, Columna 0
2. Fila 0, Columna 2
3. Fila 2, Columna 0
4. Fila 2, Columna 2

Solo 4 posiciones posibles → la **salida será de tamaño 2x2**

Primer deslizamiento Fila 0, Columna 0
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]

Segundo deslizamiento Fila 0, Columna 2
[3, 0, 1]
[6, 1, 2]
[9, 0, 1]

Tercer deslizamiento Fila 2, Columna 0
[7, 8, 9]
[1, 3, 5]
[0, 2, 4]

Cuarto deslizamiento Fila 2, Columna 2
[9, 0, 1]
[5, 7, 9]
[4, 6, 8]

Resultado final: matriz convolucional de salida de dimensiones 2x2. Más adelante se presentarán los cálculos matemáticos detallados que explican cómo se obtuvieron los valores contenidos en esta matriz.

[-6, -2]
[-10, 6]

¿Qué aprendemos?

- **Stride 2** reduce el tamaño de la salida (menos operaciones, más rápido).

- Se pierde algo de detalle, pero **aumenta la eficiencia computacional**.
- Útil para reducir dimensionalidad y evitar overfitting en redes profundas.

✓ El siguiente ejemplo muestra cómo se modifica la dimensión de la matriz resultante cuando se aplica una convolución a una imagen de 3×3 utilizando padding para conservar el tamaño original.

Imagina que tienes esta imagen 3x3:

1 2 3
4 5 6
7 8 9

Ahora, si aplicas **padding de 1 píxel**:

0 0 0 0 0
0 1 2 3 0
0 4 5 6 0
0 7 8 9 0
0 0 0 0 0

Con esta imagen extendida, el filtro puede moverse más veces y generar una salida de **tamaño mayor**, lo que permite **más aprendizaje** y **menos pérdida de información** en los bordes.

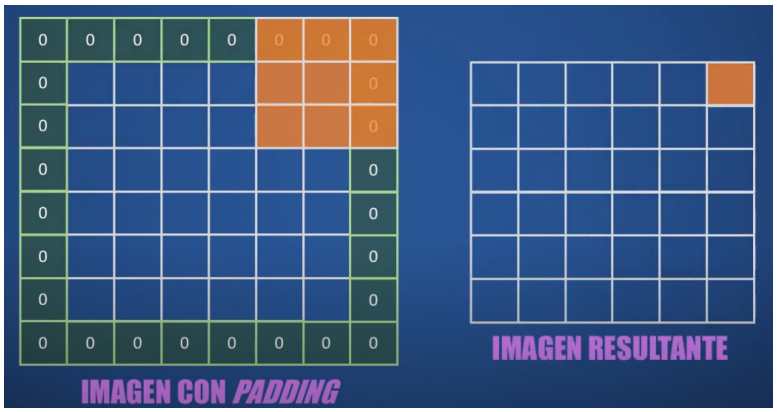


Ilustración 5. Imagen con padding: al añadir un borde de píxeles (generalmente ceros) alrededor de la imagen original, se preservan las dimensiones tras la convolución y se mejora la detección de patrones en los bordes.
FUENTE: ELABORACIÓN PROPIA

CÁLCULOS MATEMÁTICOS DE LA CONVOLUCIÓN

El filtro se desliza sobre la imagen y en cada posición se realiza una operación de **producto punto** entre el filtro y el fragmento de imagen que cubre.
El resultado se guarda en una nueva matriz llamada **capa convolucional** o **feature map**.

Ejemplo que incluye si el kernel es de 3*3, stride=1 y sin padding. Además de incluir los cálculos matemáticos entre la imagen de entrada y el kernel para obtener la capa convolucional.

[-1 0 1]
[-1 0 1]
[-1 0 1]

Primera iteración.

- Posición (0,0) de la matriz resultante (capa convolucional)

Submatriz de la imagen:

[3 0 1]
[1 5 8]
[2 7 2]

Multiplicamos elemento a elemento con el kernel:

(3*-1) + (0*0) + (1*1) + (1*-1) + (5*0) + (8*1) + (2*-1) + (7*0) + (2*1) = -3 + 0 + 1 – 1 + 0 + 8 -2 + 0 + 2 = 5

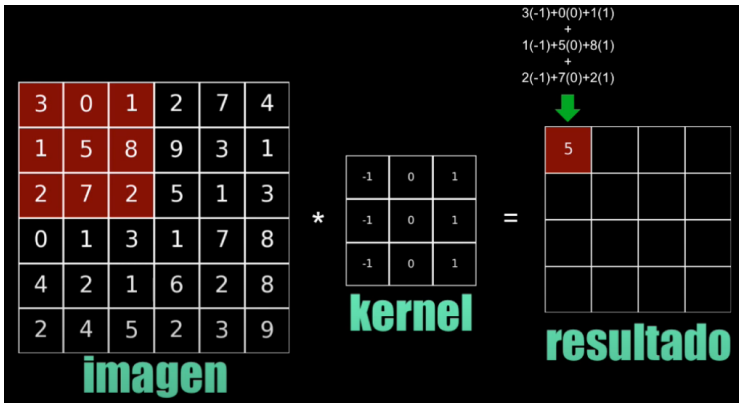


Ilustración 6. Cálculo matemático al aplicar un kernel sobre la imagen: se realiza la multiplicación elemento a elemento entre los valores del núcleo y los píxeles de la región seleccionada, seguida de la suma de los productos para obtener el valor resultante.
FUENTE: ELABORACIÓN PROPIA

Segunda iteración

- Posición (0,1) de la matriz resultante (capa convolucional)

Submatriz de la imagen:

[0 1 2]
[5 8 9]
[7 2 5]

Multiplicamos elemento a elemento con el kernel:

(0*-1) + (1*0) + (2*1) + (5*-1) + (8*0) + (9*1) + (7*-1) + (2*0) + (5*1) = 0 + 0 + 2 – 5 + 0 + 9 -7 + 0 + 5 = 4

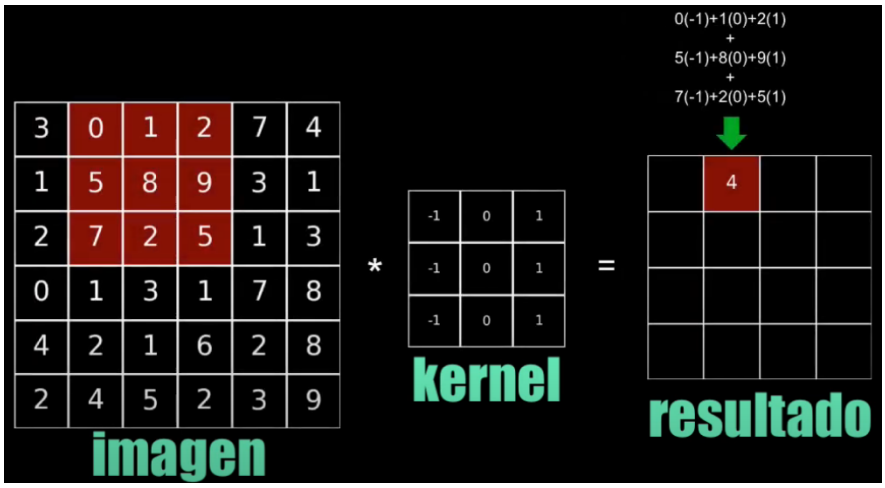


Ilustración 7. Segunda interacción convolucional: la imagen procesada pasa nuevamente por un kernel, generando mapas de características más profundos y complejos.
FUENTE: ELABORACIÓN PROPIA

Tercera iteración

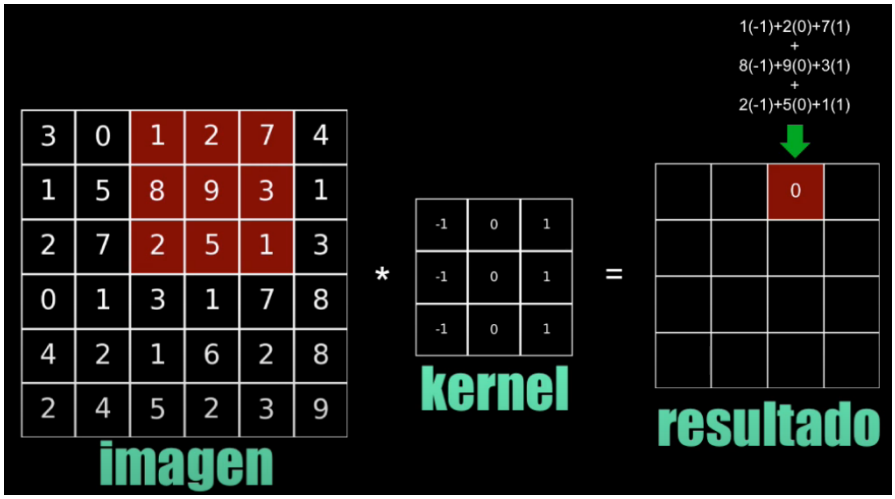


Ilustración 8. TERCERA interacción convolucional: la imagen procesada pasa nuevamente por un kernel, generando mapas de características más profundos y complejos.
FUENTE: ELABORACIÓN PROPIA

Cuarta iteración

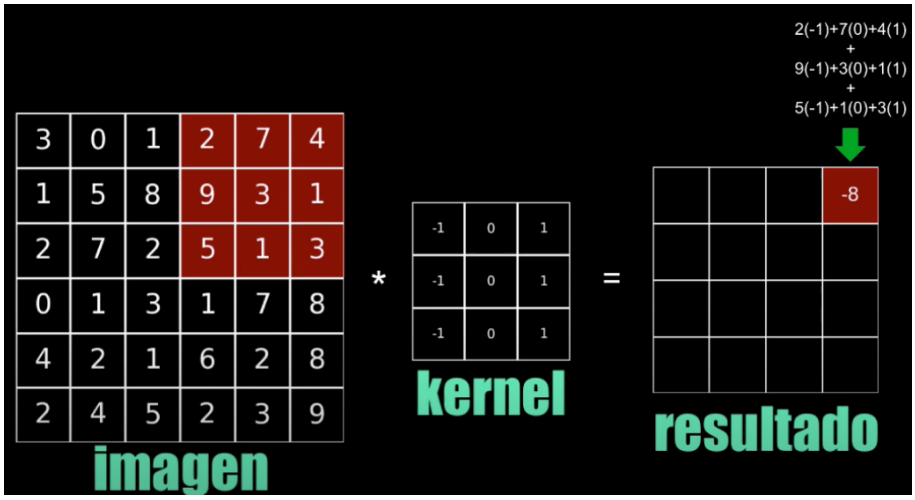


Ilustración 9. CUARTA interacción convolucional: la imagen procesada pasa nuevamente por un kernel, generando mapas de características más profundos y complejos.
FUENTE: ELABORACIÓN PROPIA

Cuando el kernel alcanza el extremo derecho de la imagen, se desplaza una posición hacia abajo –según el valor del ***stride***– y reinicia su recorrido horizontal desde la izquierda. Este proceso se repite de forma sistemática hasta completar un barrido total de la imagen. El resultado de esta operación es una nueva matriz de valores conocida como **capa convolucional** o **mapa de características**, la cual contiene la activación del filtro en cada región escaneada

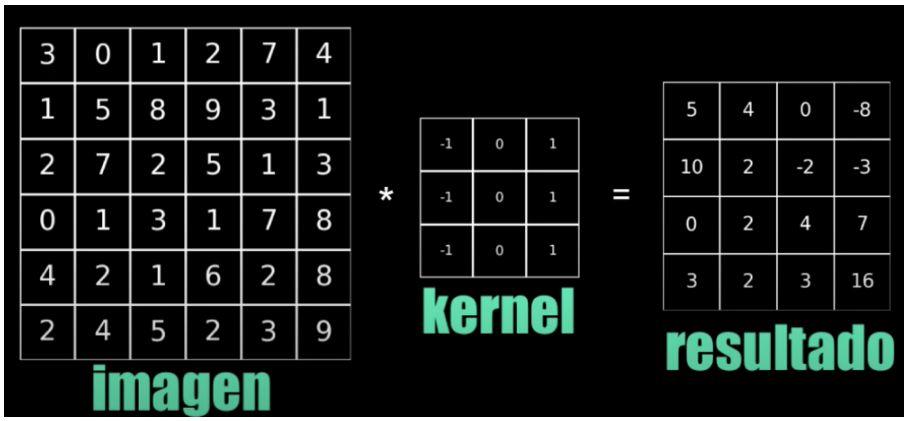


Ilustración 10. resultado al aplicar la convolucion
FUENTE: ELABORACIÓN PROPIA

A continuación, se ilustra en forma explicativa del funcionamiento de una red neuronal convolucional (CNN) al procesar una imagen en escala de grises, correspondiente al dígito '4' del dataset MNIST, que contiene números escritos a mano. Primero se representa la imagen como una matriz de píxeles, y posteriormente se aplican 32 filtros distintos; cada uno de ellos extrae una característica específica del dígito. Como resultado, se generan 32 mapas de características (feature maps), también conocidos como capas convolucionales.

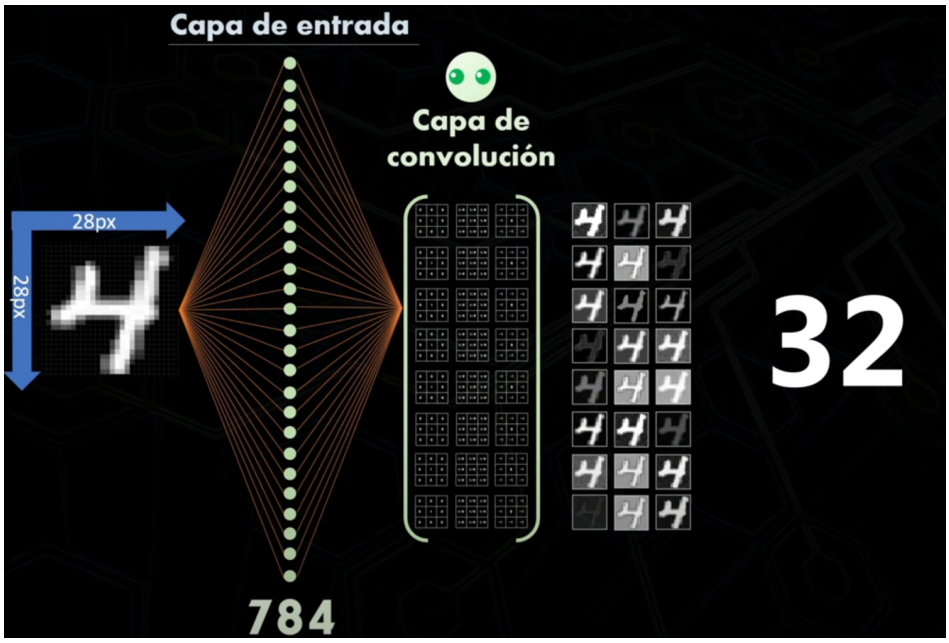


Ilustración 11. Proceso de una red convolucional con 32 filtros, cada uno encargado de resaltar diferentes aspectos de la imagen.
FUENTE: ELABORACIÓN PROPIA

Imagen de Entrada (28x28 píxeles)

- La imagen a la izquierda es un dígito "4" en blanco y negro, de **28 píxeles por 28 píxeles**.
- Cada píxel tiene un valor de intensidad (de 0 a 255), lo que genera una **matriz de 784 valores** (28 × 28 = **784**).
- Esta matriz se "aplana" o convierte en un **vector** de tamaño 784. Esa es la **capa de entrada**.

- **Dato clave**: Aquí no hay color, solo un canal (escala de grises).

Capa de Convolución

- A continuación, la imagen pasa por la **capa convolucional**.
- En esta etapa, la red aplica **filtros (kernels)** que se deslizan por toda la imagen para detectar características (como bordes, esquinas, curvas...).

En la imagen:

- Cada bloque negro con cuadrícula representa un **filtro**.
- Se están aplicando **32 filtros distintos**, lo que genera **32 mapas de características** (o feature maps).
- Cada filtro aprende a detectar una característica diferente del dígito.

Por ejemplo:

- Un filtro podría aprender a detectar líneas horizontales.
- Otro, los bucles cerrados.
- Otro, la inclinación típica del número 4.

Resultado: 32 mapas de características

- A la derecha de la capa de convolución ves los **32 pequeños recuadros grises**. Cada uno es el resultado de aplicar un filtro distinto.
- Estas imágenes transformadas ya no son simples pixeles: son **representaciones abstractas del contenido de la imagen**.
- Cada mapa **activa** (ilumina) zonas donde se detectó una característica importante.

¿Por qué esto es importante?

Porque la red **aprende a ver como lo haría un humano**:

1. Primero detecta formas básicas (líneas, bordes).
2. Luego combina esas formas para entender estructuras más complejas (por ejemplo, qué forma tiene un "4").

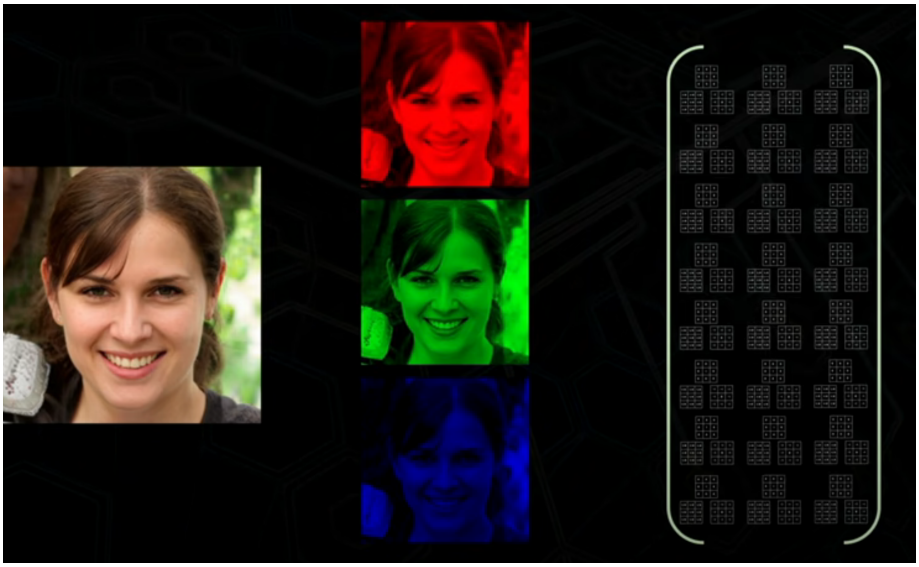
Las siguientes dos imágenes ilustran cómo una red neuronal convolucional procesa una **imagen a color (RGB)** a partir de sus **tres canales independientes**:

1. **Imagen original (izquierda):**
   Una fotografía en color que representa la entrada a la red. Visualmente es una imagen común que vemos en cualquier contexto digital.
2. **Separación en canales de color:**
   La imagen es descompuesta en **tres matrices independientes**, cada una correspondiente a un canal de color:
   - **Canal rojo (R)**: resalta las intensidades rojas.
   - **Canal verde (G)**: destaca las intensidades verdes.
   - **Canal azul (B)**: contiene las intensidades azules.

   Cada canal es una matriz de valores de intensidad que, al combinarse, reconstruyen la imagen original.

3. **Aplicación del filtro sobre los canales:**
   A la derecha, se visualiza cómo un **único filtro tridimensional (3×3×3)** se aplica **simultáneamente** sobre las tres matrices (R, G y B).
   El resultado de esta operación es una **única matriz de salida** (feature map) que refleja la activación del filtro frente a los patrones combinados de color.

La imagen ilustra que, **aunque tengamos 3 canales**, un solo filtro aplicado sobre una imagen RGB produce **una sola capa convolucional** combinando la información de los tres canales de entrada.



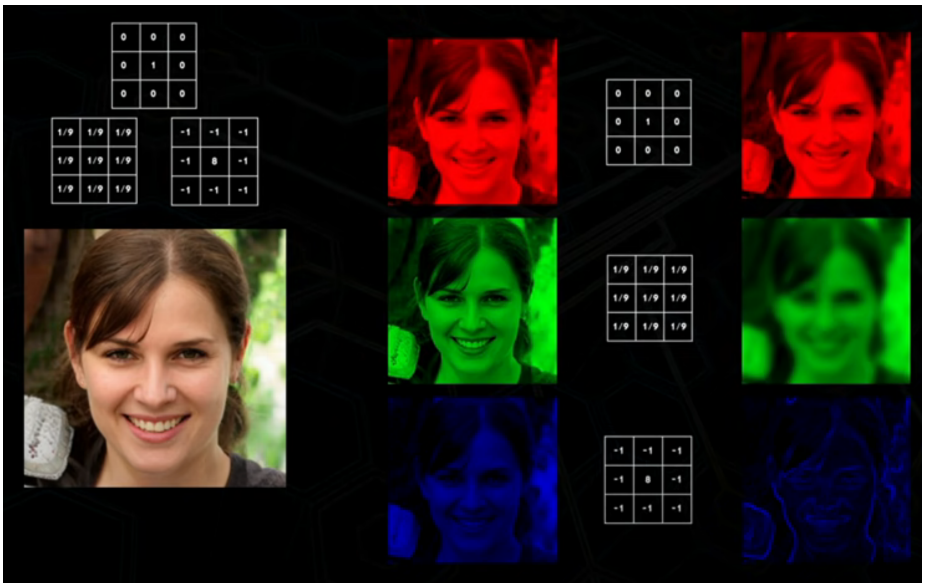


Ilustración 12. Aplicación de un filtro convolucional sobre una imagen RGB, transformando los tres canales de color (rojo, verde y azul) para resaltar patrones específicos.
FUENTE: ELABORACIÓN PROPIA

Las siguientes dos imágenes ilustran el flujo completo de procesamiento de una imagen a color (RGB) en una **red neuronal convolucional** durante la **fase de Convolución inicial**.

- 1. Capa de entrada

  - Se observa una **imagen RGB** como punto de partida, que se representa internamente como **tres matrices de canales de color** (no visibles aquí).
  - La figura central representa la transformación de la imagen en una estructura matricial conectada a la red, donde cada píxel (en cada canal) es interpretado como una entrada numérica.

- 2. Capa de convolución

  - Se aplica un conjunto de **32 núcleos (filtros)** tridimensionales, cada uno capaz de extraer diferentes patrones visuales combinando información de los canales R, G y B simultáneamente.
  - Como resultado, se generan **96 mapas de características (feature maps)**:
    - Esto sucede porque **cada uno de los 32 núcleos realiza una convolución por cada uno de los 3 canales de color**,

- Luego los tres resultados por filtro se **combinan (suman)** para generar un único mapa por filtro.

- 3. Salida de la capa convolucional

  - Se visualizan los **feature maps generados**: imágenes en escala de grises que representan distintos aspectos aprendidos como bordes, texturas, formas o colores predominantes.
  - Este conjunto de 96 salidas representa una **extracción rica de características visuales** desde distintas perspectivas.
  - Entonces, para cada uno de los **32 filtros**, se realizan **3 convoluciones** (una por canal).
  - Estas **3 convoluciones se suman** (canal por canal) y se pasa por ReLU → resultado: **1 feature map por filtro**.

En resumen:

Concepto	**Valor en el ejemplo**
Canales de la imagen	3 (R, G, B)
Núcleos/filtros utilizados	32
Convoluciones parciales	32 filtros × 3 canales = **96**
Capas convolucionales finales	**32** (una por filtro)

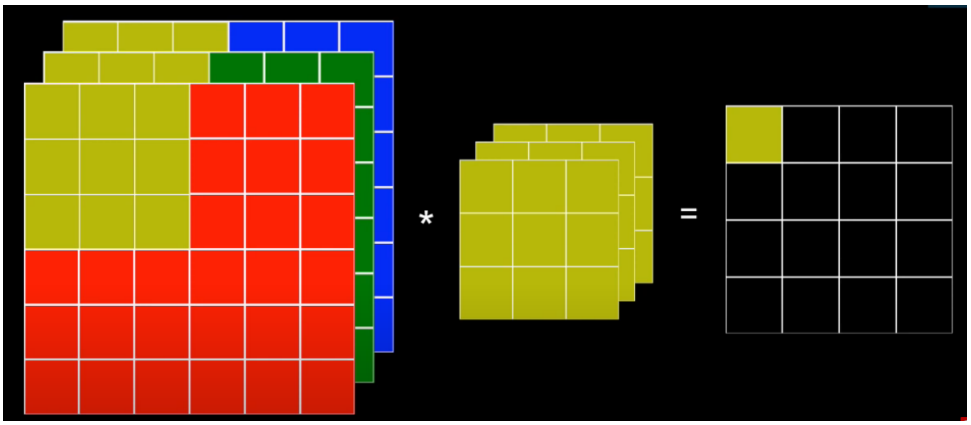


Ilustración 13. Aplicación de filtros convolucionales en imágenes RGB, mostrando cómo se procesan los tres canales de color (rojo, verde y azul) para extraer características relevantes.
FUENTE: ELABORACION PRÓPIA

APLICACIÓN DE LA FUNCIÓN DE ACTIVACIÓN RELU

En una red neuronal, **una función de activación** decide si una neurona se activa o no.
Después de hacer los cálculos (por ejemplo, una convolución), los resultados pasan por esta función.

¿Qué es ReLU?

ReLU significa **Rectified Linear Unit**, y es la función de activación más usada en redes neuronales modernas.

Su fórmula es simple:

$ReLU(x) = max(0, x)$

Es decir:

- Si el valor es **positivo**, lo deja igual.
- Si es **negativo**, lo convierte en **cero**.

¿Por qué se usa ReLU?

1. ✅ **Evita valores negativos** innecesarios → hace que la red aprenda más rápido.
2. 🚀 **Es computacionalmente eficiente** (solo compara con cero).
3. 📈 **Ayuda a reducir el problema del "desvanecimiento del gradiente"** en redes profundas.
4. 🌊 **Genera una red más activa**: solo se activan las neuronas útiles.

¿Cómo se usa ReLU en una red convolucional?

1. Se hace la **convolución** (el filtro escanea la imagen y genera una matriz de valores).
2. Esos valores pueden ser negativos o positivos.
3. Luego se aplica **ReLU a esa matriz**:
    - Se eliminan los valores negativos (se ponen en 0).
    - Se mantienen los positivos.
4. El resultado es el **mapa de activación** listo para la siguiente capa.

Intuición Visual:

Imagina que ReLU es como un **filtro que solo deja pasar lo útil**:

- Las activaciones positivas pueden representar **bordes, formas o patrones importantes**.
- Las negativas se consideran **ruido** y se descartan (al convertirse en 0).

Metáfora didáctica

ReLU es como un detector de ideas positivas en una lluvia de pensamientos: **Si una idea tiene energía (es positiva), la conserva. Si no aporta nada (es negativa), la ignora.**

Aplicación de la capa de Pooling

La **capa de pooling** (también llamada **submuestreo** o **downsampling**) es una etapa que se usa después de las capas de convolución en una red neuronal. Su función principal es **reducir el tamaño** de las representaciones intermedias (los mapas de características), **manteniendo la información más importante**.

¿Para qué sirve?

Tabla 2. objetivos del pooling
FUENTE: (ZHANG, REN, & ZHANG, 2020).

Objetivo	**Explicación**
Reducir dimensionalidad	Hace que los mapas de activación sean más pequeños y manejables.
Acelerar el entrenamiento	Menos datos → menos cálculos → red más rápida.
Hacer la red más robusta	Ignora pequeñas variaciones o ruidos locales.
Enfocarse en lo esencial	Conserva los patrones más relevantes, como bordes o formas.

¿Cómo funciona?

El **pooling** recorre la matriz de activaciones **en bloques** (ej. de 2x2) y aplica una operación sobre cada bloque.

Tipos más comunes:

1. Max Pooling (el más usado)

- Toma el **valor máximo** de cada bloque.
- Ejemplo:

  Bloque 2x2:
  [3, 5]
  [1, 2]

  Max Pooling = 5

2. Average Pooling

- Toma el **promedio** de los valores del bloque.
- Ejemplo:

  (3 + 5 + 1 + 2) /4 = 2.75

Ejemplo visual con Max Pooling 2x2

Entrada (mapa de activación 4x4):

[1, 3, 2, 4]
[5, 6, 1, 2]
[0, 1, 3, 1]
[4, 2, 6, 5]

Aplicamos Max Pooling 2x2 con stride 2 (nos movemos cada 2 posiciones):

Bloques	**Resultado**
[1, 3], [5, 6]	max = 6
[2, 4], [1, 2]	max = 4
[0, 1], [4, 2]	max = 4
[3, 1], [6, 5]	max = 6

Resultado (matriz 2x2):

[6, 4]
[4, 6]

Reducción del tamaño de 4x4 → 2x2.

¿Por qué es importante?

- **Reduce el tamaño**, evitando que la red se vuelva pesada.
- **Hace que la red se enfoque en lo que importa**, ignorando detalles pequeños o ruido.
- **Aumenta la invariancia espacial**, lo que significa que la red sigue reconociendo un patrón, aunque se mueva un poco en la imagen.

Metáfora didáctica

Imagina que estás tomando fotos de una ciudad desde un dron. En lugar de guardar cada píxel, solo guardas el punto **más alto** en cada vecindario (max pooling) o el **promedio** de la altura (average pooling).
Así obtienes un mapa **resumido** pero **informativo** de la ciudad.

REPETICIÓN DE BLOQUES: CONVOLUCIÓN + RELU + POOLING

La repetición de este conjunto de operaciones permite que la red aprenda **de forma progresiva y jerárquica**.
Cada repetición detecta **características más complejas** que la anterior. Es como si la red tuviera una lupa que **afina su enfoque capa por capa**.

¿Qué se obtiene al repetir este bloque?

Cada repetición permite que la red **suba un nivel de comprensión visual**:

Tabla 3. La tabla muestra los objetivos principales de repetir múltiples operaciones de convolución en una red neuronal, destacando cómo cada repetición permite extraer características progresivamente más complejas de la imagen.
FUENTE: (ZHANG, REN, & ZHANG, 2020).

Repetición	**¿Qué aprende?**	**Ejemplo visual**
1	Patrones simples	Bordes, colores, texturas
2	Combinaciones de patrones	Curvas, formas pequeñas
3	Estructuras complejas	Partes de objetos: ojos, ruedas, hojas
4+	Objetos completos o conceptos abstractos	Una cara, una letra, un gato

Metáfora intuitiva:

Imagina que estás dibujando un objeto paso a paso:

1. Primero trazas los bordes.
2. Luego agregas detalles como sombras o curvas.
3. Después reconoces partes del objeto (una oreja, una hoja, una ventana).
4. Finalmente, entiendes el objeto completo (¡es un gato en una ventana!).

Eso mismo hace la red: **aprende desde lo más básico hasta lo más significativo**, **repitiendo los bloques de convolución + ReLU + pooling**.

Resultado final

- Se obtiene una representación **compacta**, **rica en significado** y **lista para clasificar o interpretar**.
- Esta representación es enviada a las **capas densas finales** (fully connected), que hacen la **predicción final**.

***CAPA** FULLY CONNECTED (DENSA)*

La **capa fully connected (totalmente conectada)** es una **capa densa** en la que **cada neurona está conectada con todas las neuronas de la capa anterior**.

Es como la parte final de una CNN, donde la red ya dejó de "ver" píxeles y ahora empieza a **tomar decisiones**.

¿Qué función cumple?

Después de extraer características importantes de la imagen mediante:

- varias capas de **convolución + ReLU + pooling**,

...la red transforma toda esa información **en un vector de números** (flatten) y se la pasa a las capas fully connected

¿Cómo funciona paso a paso?

1. **Flatten**: La salida de las últimas capas convolucionales (matriz) se **aplana** en un vector largo.
   Ejemplo: de una matriz 7×7×64 → un vector de 3136 elementos.
2. **Multiplicación por pesos + bias**:
   Cada número se combina con otros **a través de pesos**, y se suman para formar nuevas neuronas.

Ejemplo intuitivo:

Supón que tienes una red entrenada para reconocer **tipos de frutas**. Después de pasar por las capas convolucionales, se detectaron cosas como:

- Redondez
- Textura lisa
- Color rojo

La **fully connected** toma esa combinación y concluye:

"¡Con alta probabilidad, esta fruta es una manzana!"

Metáfora didáctica:

Piensa en una red como un detective:

- Las capas convolucionales son como investigadores que recolectan pistas.
- La capa fully connected es el **juez que escucha todas las pistas y emite el veredicto final**.

¿Por qué es tan importante?

- Es la capa que **une todo lo aprendido**.
- Es responsable de **transformar conocimiento visual en decisiones concretas**.
- Permite **la clasificación, detección, segmentación o predicción final**.

ACTIVACIÓN FINAL

- Se usa **Softmax** (en clasificación) para obtener probabilidades de cada clase.
    - Ej.: [0.1, 0.8, 0.1] → clase 2 con 80% de probabilidad.

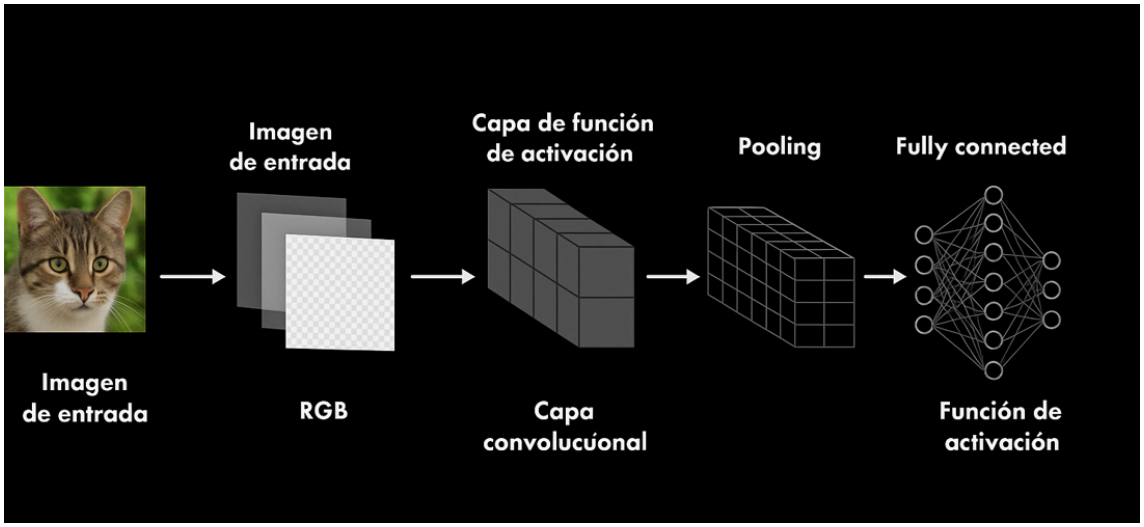


Ilustración 14. estructura de una red convolucional
FUENTE: ELABORACIÓN PROPIA

¿Quién se encarga de la clasificación final?

✅ La capa Fully Connected + las funciones de activación de salida trabajan en conjunto.

- **La capa fully connected** (FC) **hace los cálculos**: combina toda la información aprendida por la red y genera una salida **numérica** para cada clase posible.
- **La función de activación final** (como **Softmax**) **convierte esos números en probabilidades** interpretables, que te dicen **a qué clase pertenece la entrada**.

Ejemplo práctico

Supón que estás clasificando dígitos del 0 al 9 (como en MNIST).

1. La **última capa fully connected** tiene **10 neuronas**, una por cada dígito.
2. La red calcula los siguientes valores:

   [1.2, -0.3, 0.8, 2.5, -1.0, 0.2, 0.9, 0.5, 1.8, 0.0]

3. Luego se aplica la función **Softmax** sobre ese vector:
   - Convierte esos valores en una **distribución de probabilidades**:

   [0.05, 0.01, 0.03, 0.60, 0.005, 0.02, 0.04, 0.02, 0.20, 0.025]

Aquí, la mayor probabilidad es **0.60 en la posición 3**, así que la red clasifica la imagen como el **número 3**.

En resumen

Elemento	**Rol**
Fully connected	Realiza el cálculo para cada clase (combinación lineal).
Función de activación	Interpreta esa salida como probabilidades (ej. Softmax).
Resultado final	La clase con mayor probabilidad es la **predicción final**.

CASO DE ESTUDIO: CLASIFICACIÓN DE IMÁGENES MEDIANTE UNA RED NEURONAL CONVOLUCIONAL (CNN)

📌 Objetivo del caso

El presente caso de estudio tiene como objetivo aplicar una red neuronal convolucional (CNN) utilizando Keras y Tensor Flow para clasificar imágenes organizadas en carpetas por clase. Se utiliza un enfoque práctico, desde la carga de datos hasta la visualización de métricas, aplicable a problemas de clasificación de imágenes en diferentes contextos (como detección de objetos, reconocimiento de animales, diagnóstico médico, etc.).

Herramientas utilizadas

- **Lenguaje**: Python
- **Entorno**: Google Colab
- **Framework**: Tensor Flow / Keras
- **Visualización**: Matplotlib
- **Carga y preprocesamiento de imágenes**: ImageDataGenerator

📈 Resultado esperado

Al finalizar el entrenamiento, el modelo es capaz de clasificar imágenes nuevas según las clases aprendidas. Los gráficos permiten verificar que la precisión haya aumentado de forma progresiva y que la pérdida haya disminuido, tanto en entrenamiento como en validación.

💡 Aplicaciones

Este flujo puede adaptarse fácilmente a:

- Clasificación de frutas, alimentos, o prendas de ropa.
- Diagnóstico médico (clasificación de imágenes radiológicas).
- Reconocimiento facial o de objetos en tiempo real.
- Detección de emociones en imágenes.

Código	Descripción
# ✅ 1. IMPORTAR LIBRERÍAS NECESARIAS import os import matplotlib. pyplot as plt from tensorflow.keras.preprocessing.image import ImageDataGenerator	Este encabezado indica que estamos incluyendo las bibliotecas necesarias para cargar, procesar imágenes y construir una red neuronal convolucional. import os

from tensorflow.keras.models import Sequential from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout from tensorflow.keras.optimizers import Adam	🔹 os: Biblioteca estándar de Python para manejar rutas, carpetas y archivos del sistema operativo. ➡️ Útil, por ejemplo, para acceder a la carpeta del dataset, mover archivos, verificar directorios, etc. import matplotlib.pyplot as plt 🔹 matplotlib.pyplot: Biblioteca para hacer gráficos y visualizaciones. ➡️ La usaremos más adelante para graficar la precisión y la pérdida del entrenamiento (accuracy/loss). from tensorflow.keras.preprocessing.image import ImageDataGenerator 🔹 ImageDataGenerator: Herramienta de Keras que permite: Cargar imágenes directamente desde carpetas. Aplicar preprocesamiento y aumentos de datos (como rotación, zoom, etc.). Generar automáticamente los conjuntos de entrenamiento y validación.

	from tensorflow.keras.models import Sequential 🔹 Sequential: Es una forma sencilla de construir modelos capa por capa, ideal para redes simples como una CNN básica. ➡️ Permite crear el modelo agregando las capas una tras otra en orden. from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout 🔹 Estas son las capas fundamentales de una CNN: Capa ¿Para qué sirve? Conv2D Capa convolucional: extrae características (bordes, texturas, formas). MaxPooling2D Reduce el tamaño del mapa de características, conservando lo más importante. Flatten Aplana la salida para conectarla a las capas densas finales. DenseCapa totalmente conectada para tomar decisiones finales. Dropout Técnica de regularización para prevenir sobreajuste (overfitting).

	from tensorflow.keras.optimizers import Adam 🔹 Adam: Uno de los optimizadores más usados en Deep Learning. ➡️ Se encarga de ajustar automáticamente los pesos de la red durante el entrenamiento, con buenos resultados y rapidez.
# ✅ 2. DEFINIR LA RUTA DONDE ESTÁ EL DATASET ruta_dataset = '/content/mi_dataset' # CAMBIA ESTA RUTA según donde esté tu dataset	# Estructura esperada: # /ruta/mi_dataset/ # ├── clase1/ # ├── clase2/ # ├── clase3/ (si es multiclase)
# ✅ 3. PARÁMETROS DEL MODELO tamaño_imagen = (150, 150) batch_size = 32 epochs = 10 validation_split = 0.2 # 20% para validación	Este bloque define los **hiperparámetros clave** que determinan cómo se procesan los datos y cómo se entrena el modelo. tamaño_imagen = (150, 150) 🔹 Define el tamaño al que se redimensionarán todas las imágenes antes de entrar a la red neuronal. ➡️ En este caso, todas las imágenes se ajustarán a 150 píxeles de alto y 150 de ancho, independientemente de su tamaño original. ✅ Es necesario porque una red neuronal no puede trabajar con imágenes de diferentes tamaños. batch_size = 32

	🔹 Indica cuántas imágenes se procesarán a la vez en cada paso del entrenamiento (mini-lote o batch). ➡️ Un valor de 32 es común: balancea bien el uso de memoria y la estabilidad del aprendizaje. ✅ Cuanto más grande el batch, más memoria necesitas; si es muy pequeño, el entrenamiento puede volverse inestable. epochs = 10 🔹 Número de épocas: cuántas veces el modelo verá todo el conjunto de entrenamiento completo. ➡️ En este caso, entrenará durante 10 vueltas completas sobre los datos. ✅ Puedes aumentarlo si ves que aún no converge, o usar EarlyStopping para detener automáticamente cuando mejore. validation_split = 0.2 # 20% para validación 🔹 Porcentaje del dataset que se usará como conjunto de validación. ➡️ El 20% de las imágenes se reservarán para evaluar el rendimiento del modelo en datos nunca vistos durante el entrenamiento. ✅ Sirve para detectar si el modelo está sobreajustando (memoriza los datos pero no generaliza).

# ✅ 4. PREPROCESAMIENTO DE IMÁGENES CON AUMENTO Y NORMALIZACIÓN datagen = ImageDataGenerator (rescale=1. /255, validation_split=validation_split, rotation_range=20, zoom_range=0.15, width_shift_range=0.1, height_shift_range=0.1, horizontal_flip=True)	Este bloque crea un objeto llamado datagen que se encargará de **preprocesar y aumentar las imágenes automáticamente** mientras se cargan desde el disco. Esto es fundamental para: • Preparar los datos correctamente. • Aumentar la **variabilidad** de las imágenes y evitar el **sobreajuste** (overfitting). datagen = ImageDataGenerator (🔹 Creamos un generador de imágenes con las opciones que están dentro de los paréntesis. rescale=1./255, 🔸 **Normalización de píxeles**: transforma todos los valores de los píxeles de 0–255 a un rango entre 0–1, lo cual es necesario para que la red aprenda de forma estable y rápida. ✅ ¡Muy importante! Mejora el rendimiento y evita problemas numéricos. validation_split=validation_split, 🔸 Utiliza la variable validation_split = 0.2 definida antes. Esto permite que **el mismo generador se encargue de dividir los datos en entrenamiento y validación**. rotation_range=20, 🔸 Permite que las imágenes se roten aleatoriamente hasta **20 grados**.

	✅ Ayuda a la red a aprender **invariancia a la orientación**. zoom_range=0.15, 🔸 Aplica un **zoom aleatorio** de hasta 15% a las imágenes. ✅ Hace que la red sea más robusta ante acercamientos o alejamientos. width_shift_range=0.1, height_shift_range=0.1, 🔸 Desplaza aleatoriamente la imagen **horizontal y verticalmente** hasta un 10% del ancho o alto. ✅ Simula que el objeto puede aparecer en diferentes partes de la imagen. horizontal_flip=True 🔸 Invierte horizontalmente algunas imágenes (como si se reflejaran en un espejo). ✅ Útil en tareas donde la orientación izquierda/derecha no cambia la clase (por ejemplo, perros, gatos, frutas…). --- 🧠 ¿Por qué es útil? Este preprocesamiento **genera nuevas versiones de las imágenes originales en tiempo real** mientras se entrena el modelo, lo cual: ✅ Aumenta artificialmente el tamaño del dataset ✅ Ayuda a que el modelo generalice mejor ✅ Reduce el riesgo de overfitting

# ✅ 5. GENERADORES DE DATOS 	
 	
train_generator = datagen.flow_from_directory(
ruta_dataset,
target_size=tamaño_imagen,
batch_size=batch_size,
class_mode='categorical', # usa 'binary' si solo hay 2 clases
subset='training',
shuffle=True
)
 	
val_generator = datagen.flow_from_directory(
ruta_dataset,
target_size=tamaño_imagen,
batch_size=batch_size,
class_mode='categorical',
subset='validation',
shuffle=False
)
	Este bloque utiliza el ImageDataGenerator (datagen) creado previamente para cargar imágenes desde carpetas, aplicarles el preprocesamiento definido (aumento de datos, normalización, etc.) y dividir automáticamente los datos en entrenamiento y validación.
	
	Los generadores devuelven imágenes en lotes, listas para ser utilizadas durante el entrenamiento del modelo.
	
	train_generator – Conjunto de entrenamiento
	
	train_generator = datagen.flow_from_directory(
	ruta_dataset,
	🔹 Indica la **ruta principal** donde están las subcarpetas de clases. Ejemplo de estructura:
	
	/mi_dataset/
	├── gatos/
	├── perros/
	target_size=tamaño_imagen,
	🔸 Redimensiona todas las imágenes a un tamaño fijo, por ejemplo **(150, 150)**, para que sean compatibles con la red neuronal.
	
	batch_size=batch_size,
	🔸 Número de imágenes que se cargarán **en cada lote** durante el entrenamiento.
	Ejemplo: si batch_size = 32, se entrenará con 32 imágenes a la vez.
	
	class_mode='categorical', # usa 'binary' si solo hay 2 clases

	🔸 Determina **el formato de salida** de las etiquetas: Modo — ¿Cuándo usarlo? — Salida 'binary' — Cuando hay **2 clases** — 0 o 1 'categorical' — Cuando hay **más de 2 clases** — One-hot vector ✅ En este caso, se usa 'categorical' porque el modelo espera varias clases. subset='training', 🔸 Este generador usa el **80% de los datos** (porque validation_split=0.2) para **entrenamiento**. shuffle=True 🔸 Mezcla aleatoriamente las imágenes antes de cada época. ✅ Esto ayuda a que el modelo no se sesgue por el orden de los datos. 🔎 val_generator – Conjunto de validación val_generator = datagen.flow_from_directory(ruta_dataset, target_size=tamaño_imagen, batch_size=batch_size, class_mode='categorical', subset='validation', shuffle=False) 🔹 Este generador funciona igual que el anterior, pero con:

	• subset='validation' → usa el 20% restante del dataset para validar el modelo. • shuffle=False → no mezcla los datos. Esto es útil para hacer predicciones y comparaciones ordenadas más adelante. --- 🧠 ¿Por qué usar generadores? ✅ Permiten **cargar imágenes desde disco sin tener que cargarlas todas a memoria**. ✅ Aplican **preprocesamiento y aumentos en tiempo real**. ✅ Dividen el dataset automáticamente en entrenamiento y validación.
# ✅ 6. CREAR LA RED NEURONAL CONVOLUCIONAL modelo = Sequential() # Capa 1: convolución + pooling modelo.add (Conv2D(32, (3, 3), activation='relu', input_shape=(tamaño_imagen[0], tamaño_imagen[1], 3))) modelo.add (MaxPooling2D(pool_size=(2, 2))) # Capa 2	Se crea un modelo secuencial, es decir, un modelo **en el que las capas se agregan una tras otra**, de manera lineal. Es ideal para redes como CNN clásicas. modelo.add(Conv2D(32, (3, 3), activation='relu', input_shape=(tamaño_imagen[0], tamaño_imagen[1], 3))) 🔹 Crea una **capa convolucional** con 32 filtros de tamaño 3x3. 🔹 Aplica la función de activación **ReLU** para dejar pasar solo las activaciones positivas. 🔹 input_shape=(150, 150, 3) define

modelo.add(Conv2D(64, (3, 3), activation='relu')) modelo.add(MaxPooling2D(pool_size=(2, 2))) # Capa 3 modelo.add(Conv2D(128, (3, 3), activation='relu')) modelo.add(MaxPooling2D(pool_size=(2, 2))) # Aplanar y conectar modelo.add(Flatten()) modelo.add(Dense(512, activation='relu')) modelo.add(Dropout(0.5)) # Regularización para evitar overfitting modelo.add(Dense(train_generator.num_classes, activation='softmax')) # Capa de salida	la forma de entrada: imágenes de 150x150 con 3 canales (RGB). modelo.add(MaxPooling2D(pool_size=(2, 2))) 🔹 Reduce la dimensionalidad de la salida con **max pooling 2x2**, manteniendo solo los valores más altos de cada región. ✅ Ayuda a hacer la red más eficiente y resistente a pequeñas variaciones. --- Segunda capa convolucional + pooling modelo.add(Conv2D(64, (3, 3), activation='relu')) modelo.add(MaxPooling2D(pool_size=(2, 2))) 🔹 Segunda capa convolucional con 64 filtros (más capacidad para detectar patrones más complejos). 🔹 Otro max pooling para seguir reduciendo la dimensionalidad. --- Tercera capa convolucional + pooling modelo.add(Conv2D(128, (3, 3), activation='relu')) modelo.add(MaxPooling2D(pool_size=(2, 2))) 🔹 Tercera capa con 128 filtros, que ya es capaz de detectar **estructuras más abstractas** (por ejemplo, formas completas, bordes compuestos, texturas).

	🔹 El max pooling continúa reduciendo el tamaño y conservando lo esencial. --- 🔄 Aplanar y conectar con capas densas modelo.add(Flatten()) 🔹 Convierte la salida 3D del último mapa de características en un **vector 1D** para poder conectar con las capas densas (fully connected). --- modelo.add(Dense(512, activation='relu')) 🔹 Crea una **capa totalmente conectada (fully connected)** con 512 neuronas. 🔹 Aprende combinaciones más complejas de características. --- modelo.add(Dropout(0.5)) # Regularización para evitar overfitting 🔹 Apaga aleatoriamente el **50% de las neuronas durante cada paso de entrenamiento**, para **evitar que el modelo memorice demasiado los datos (overfitting)**. --- 🎯 Capa de salida modelo.add(Dense(train_generator.num_classes, activation='softmax')) # Capa de salida

	🔹 Capa de salida con tantas neuronas como clases tenga el dataset (train_generator.num_classes). 🔹 Se usa softmax para convertir los valores en **probabilidades** (una por clase). ✅ El modelo elegirá la clase con mayor probabilidad como predicción final.
# ✅ 7. COMPILAR EL MODELO modelo.compile(loss='categorical_crossentropy', # Cambiar a 'binary_crossentropy' si hay 2 clases optimizer=Adam(learning_rate=0.0001), metrics=['accuracy'])	Este bloque configura el modelo para que esté **preparado para el entrenamiento**. Aquí se definen tres aspectos clave: modelo.compile(🔹 Este método le dice a Keras cómo debe **entrenar el modelo**, es decir: Qué **función de pérdida** utilizar. Qué **algoritmo de optimización** usar para ajustar los pesos. Qué **métrica evaluar** durante el entrenamiento. loss='categorical_crossentropy', # Cambiar a 'binary_crossentropy' si hay 2 clases 🔸 **Función de pérdida (loss function)**: mide qué tan lejos están las predicciones del modelo respecto a las etiquetas reales. 'categorical_crossentropy' se usa cuando hay **más de dos clases** y las

	etiquetas están en formato **one-hot encoding**. Si tu problema es **binario (2 clases)**, se recomienda usar 'binary_crossentropy'. ✅ La función de pérdida guía al modelo para que aprenda durante el entrenamiento. optimizer=Adam(learning_rate=0.0001), 🔸 **Optimizador**: es el algoritmo que **ajusta los pesos** de la red para minimizar la pérdida. Adam es uno de los más usados porque combina lo mejor de los métodos **SGD** y **RMSprop**. learning_rate=0.0001: es la velocidad con la que el modelo aprende. Un valor bajo significa que aprende más lentamente, pero con mayor precisión. metrics=['accuracy']) 🔸 **Métrica**: define qué **medida de desempeño** se quiere observar mientras se entrena el modelo. 'accuracy' mide la **precisión del modelo**, es decir, el **porcentaje de predicciones correctas**. ✅ Esta métrica se mostrará durante el entrenamiento y validación para evaluar el rendimiento del modelo.

	🧠 ¿Qué hace este bloque? Elemento Función loss Indica cómo se penaliza el error del modelo. optimizer Ajusta los pesos para reducir el error. metrics Muestra qué tan bien está funcionando el modelo (precisión).
# ✅ 8. ENTRENAR EL MODELO historia = modelo.fit(train_generator, validation_data=val_generator, epochs=epochs)	Este bloque es el **corazón del proceso de aprendizaje**: aquí es donde el modelo **se entrena usando los datos**. Durante este paso, la red ajusta sus pesos internos para mejorar su capacidad de hacer predicciones correctas. historia = modelo.fit(🔹 modelo.fit() es el método que **inicia el entrenamiento** del modelo con los datos proporcionados. 🔹 Se guarda el resultado en una variable llamada historia, que contiene información útil sobre el proceso de entrenamiento (precisión, pérdida, etc. por época), ideal para **graficar el rendimiento después**. train_generator, 🔸 Este es el **conjunto de entrenamiento**, generado con ImageDataGenerator. Contiene las imágenes preprocesadas y organizadas en lotes para que la red pueda aprender.

validation_data=val_generator,

🔸 Especifica el **conjunto de validación**, que se usará para evaluar el rendimiento del modelo al final de cada época con datos **nunca vistos durante el entrenamiento**.

✅ Esto permite monitorear si el modelo **está generalizando bien** o si está empezando a **sobreajustarse (overfitting)**.

---

epochs=epochs

)

🔸 Define el **número de veces que el modelo recorrerá completamente el conjunto de entrenamiento**.

Por ejemplo: si epochs=10, el modelo verá todas las imágenes 10 veces (en diferentes combinaciones si hay shuffle=True).

🧠 ¿Qué hace este bloque?

Elemento	Función
train_generator	Proporciona datos de entrenamiento con aumento y normalización
val_generator	Proporciona datos para validar la precisión del modelo
epochs	Define cuántas vueltas completas dará sobre los datos

	historia	Guarda métricas de entrenamiento y validación para analizarlas luego
# ✅ 9. VISUALIZAR RESULTADOS plt.figure(figsize=(12, 5)) # Precisión plt.subplot(1, 2, 1) plt.plot(historia.history['accuracy'], label='Entrenamiento') plt.plot(historia.history['val_accuracy'], label='Validación') plt.title('Precisión') plt.xlabel('Épocas') plt.ylabel('Precisión') plt.legend() # Pérdida plt.subplot(1, 2, 2) plt.plot(historia.history['loss'], label='Entrenamiento') plt.plot(historia.history['val_loss'], label='Validación') plt.title('Pérdida') plt.xlabel('Épocas') plt.ylabel('Pérdida') plt.legend()	Este bloque crea **dos gráficos** que permiten analizar cómo se comportó el modelo durante el entrenamiento. Verás: Cómo evolucionó la **precisión (accuracy)**. Cómo evolucionó la **pérdida (loss)**. Esto te ayuda a identificar si el modelo aprendió bien, si necesita más épocas, o si está sobreajustándose. plt.figure(figsize=(12, 5)) 🔹 Crea una figura de tamaño **12 unidades de ancho por 5 de alto**, para colocar dos gráficos uno al lado del otro (precisión y pérdida). --- 📈 Subgráfico 1: Precisión (Accuracy) # Precisión plt.subplot(1, 2, 1) 🔸 Define el primer subgráfico: 1 fila, 2 columnas, y este será el primero (izquierda). plt.plot(historia.history['accuracy'], label='Entrenamiento') plt.plot(historia.history['val_accuracy'], label='Validación') 🔸 Grafica cómo fue cambiando la **precisión (accuracy)** tanto para el conjunto de **entrenamiento** como

plt.tight_layout() plt.show()	para **validación** a lo largo de las épocas. plt.title('Precisión') plt.xlabel('Épocas') plt.ylabel('Precisión') plt.legend() 🔸 Personaliza el gráfico con títulos y etiquetas, y muestra la leyenda para identificar cada línea. --- 📉 Subgráfico 2: Pérdida (Loss) # Pérdida plt.subplot(1, 2, 2) 🔸 Este es el segundo subgráfico, colocado a la derecha. plt.plot(historia.history['loss'], label='Entrenamiento') plt.plot(historia.history['val_loss'], label='Validación') 🔸 Grafica la **función de pérdida**, que mide el error del modelo. Idealmente, debería disminuir con el tiempo. plt.title('Pérdida') plt.xlabel('Épocas') plt.ylabel('Pérdida') plt.legend() 🔸 Títulos y leyendas igual que en el gráfico de precisión.

	🧪 Ajuste final y visualización plt.tight_layout() plt.show() 🔸 tight_layout() ajusta automáticamente los espacios entre los gráficos para que no se encimen. 🔸 show() muestra la figura en pantalla. --- 🧠 ¿Qué se obtiene con esto? Un **análisis visual claro** del rendimiento del modelo. Te ayuda a ver si: El modelo está aprendiendo (accuracy sube, loss baja). Hay sobreajuste (accuracy de entrenamiento sube mucho, pero validación no). Hay subajuste (ambas accuracy son bajas).

RESUMEN FINAL DEL CAPÍTULO 1

En este capítulo se presentó la estructura fundamental de las Redes Neuronales Convolucionales (CNN) en su forma secuencial, explicando con detalle los principales componentes de su arquitectura. Se abordaron las capas de convolución, encargadas de extraer patrones locales de las imágenes; las capas de pooling, que reducen la dimensionalidad y permiten un aprendizaje más eficiente; y las capas fully connected, que integran la información aprendida para realizar la clasificación final.

Además, se incluyó un ejercicio práctico en Google Colab, mediante el cual los lectores pudieron aplicar los conceptos estudiados y reforzar la comprensión del flujo de datos a través de cada capa de una CNN. Este ejercicio consolidó la conexión entre la teoría y la práctica, ofreciendo al lector una primera aproximación al diseño y entrenamiento de modelos.

El capítulo también profundizó en los aspectos matemáticos y conceptuales que sustentan el funcionamiento de las CNN. Se explicó el cálculo de convoluciones, el rol de los filtros y kernels en la detección de características, y la importancia de la función de activación ReLU para introducir no linealidad. Asimismo, se analizaron los efectos del stride y padding en el tamaño de las salidas intermedias, y se destacó la función de las capas de pooling en la reducción de la complejidad sin perder información esencial.

De esta forma, el capítulo permitió comprender cómo los elementos matemáticos y computacionales se integran en la arquitectura de una CNN, brindando al lector las herramientas necesarias para analizar y diseñar modelos de manera más consciente y fundamentada. En conclusión, este capítulo sentó las bases conceptuales y prácticas indispensables para avanzar hacia arquitecturas más complejas que se desarrollarán en los siguientes capítulos.

AUTOEVALUACIÓN

Instrucciones: Marca con ✔ si puedes responder correctamente cada pregunta. Si tienes dudas, revisa nuevamente la sección correspondiente del capítulo.

1. ¿Cuál es la función principal de una capa convolucional?
2. ¿Qué representa un kernel y cómo actúa sobre una imagen?
3. ¿Qué diferencias hay entre padding “valid” y “same”?
4. ¿Para qué se utiliza la función de activación ReLU en una CNN?
5. ¿Cuál es el propósito de la capa de pooling?
6. ¿Qué significa “aplanar” antes de pasar a las capas densas?
7. ¿Qué tipo de problemas se pueden resolver con redes convolucionales?
8. ¿Cuáles son las ventajas de utilizar técnicas como data augmentation?
9. ¿Qué métricas usaste para evaluar la CNN en el caso práctico?
10. ¿Qué pasos seguirías para mejorar el rendimiento de un modelo CNN?

EJERCICIOS DE APLICACIÓN DE CONTENIDOS

Ejercicio 1: Arquitectura CNN básica

- Diseña una red neuronal convolucional para clasificar imágenes de flores (usa un dataset como Flowers Recognition en Kaggle).
- Especifica las capas utilizadas, funciones de activación y tamaño de filtros.

Ejercicio 2: Interpretación de resultados

- Entrena una CNN básica y genera las gráficas de precisión y pérdida por época.
- Explica cómo interpretar los resultados obtenidos y si hay evidencia de overfitting o underfitting.

Ejercicio 3: Modificación de hiperparámetros

- Repite el caso práctico de perros y gatos, pero cambiando el optimizador o el tamaño de las capas.
- Compara resultados y discute qué configuración fue más eficiente.

TRABAJO AUTÓNOMO

Proyecto sugerido: Clasificador de frutas con CNN

1. Descarga un dataset de imágenes de frutas (como el Fruit Images Dataset).
2. Preprocesa las imágenes y organiza los datos en carpetas por clase.
3. Crea una CNN que identifique correctamente al menos 3 tipos de frutas.
4. Evalúa su rendimiento y experimenta con técnicas de regularización y aumento de datos.
5. Documenta los resultados e interpreta gráficamente la precisión y la pérdida.

REFERENCIA BIBLIOGRÁFICA

Abiodun, O. I., Jantan, A., Omolara, A. E., Dada, K. V., Mohamed, N. A., & Arshad, H. (2021). Comprehensive review of deep learning approaches to CNN-based image classification systems. ***IEEE Access, 9***, 103383–103401. https://doi.org/10.1109/ACCESS.2021.3093764

Khan, A., Sohail, A., Zahoora, U., & Qureshi, A. S. (2020). A survey of the recent architectures of deep convolutional neural networks. ***Artificial Intelligence Review, 53***, 5455–5516. https://doi.org/10.1007/s10462-020-09825-6

Tan, M., & Le, Q. V. (2019). EfficientNet: Rethinking model scaling for convolutional neural networks. In ***Proceedings of the 36th International Conference on Machine Learning*** (pp. 6105–6114). https://arxiv.org/abs/1905.11946

Liu, Z., Mao, H., Wu, C.-Y., Feichtenhofer, C., Darrell, T., & Xie, S. (2022). A ConvNet for the 2020s. In ***Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*** (pp. 11976–11986). https://arxiv.org/abs/2201.03545

Alzubaidi, L., Zhang, J., Humaidi, A. J., Al-Dujaili, A., Duan, Y., Al-Shamma, O., ... & Farhan, L. (2021). Review of deep learning: Concepts, CNN architectures, challenges, applications, future directions. ***Journal of Big Data, 8***(1), 1–74. https://doi.org/10.1186/s40537-021-00444-8

Sharma, A., & Jain, A. (2022). Deep convolutional neural networks: An overview. In ***2022 2nd International Conference on Intelligent Technologies*** (pp. 1–6). IEEE. https://doi.org/10.1109/ICIT57492.2022.10007104

Zhang, Y., Ren, Y., & Zhang, Y. (2020). An overview of pooling in deep learning: Progress, challenges, and future directions. ***Journal of Artificial Intelligence and Technology, 1***(1), 1–13. https://doi.org/10.1007/s42491-020-00001-w

Dai, L., Zhou, M., & Liu, H. (2023). Recent Applications of Convolutional Neural Networks in Medical Data Analysis. IGI Global.

FlatWorld Solutions. (2025). Top 7 Convolutional Neural Networks Applications. https://www.flatworldsolutions.com/data-science/articles/7-applications-of-convolutional-neural-networks.php

Jia, H., et al. (2024). Application of convolutional neural networks in medical imaging. BMC Medical Imaging, XX(X), XX–XX.

Li, M. (2025). Deep Convolutional Neural Networks in Medical Image. MDPI.

Yamashita, R., et al. (2018). Convolutional neural networks: an overview and application in medical imaging. Insights into Imaging, 9(4), 611–629.

CAPÍTULO 2. MODELOS PREENTRENADOS DE REDES CONVOLUCIONALES

RESUMEN DEL CAPÍTULO

Este capítulo introduce el uso de modelos preentrenados en deep learning como una solución eficiente para resolver tareas complejas sin necesidad de entrenar redes desde cero. Se explican los conceptos fundamentales de ***transfer learning*** y ***fine-tuning***, sus beneficios y cuándo aplicarlos. A través de un caso práctico de clasificación de imágenes de perros y gatos utilizando MobileNetV2, se demuestra cómo adaptar modelos preentrenados para tareas personalizadas. Se abordan también técnicas de aumento de datos y estrategias de ajuste fino para optimizar el rendimiento.

INTRODUCCIÓN

Entrenar redes neuronales profundas desde cero suele requerir grandes volúmenes de datos y potentes recursos computacionales. Sin embargo, gracias al aprendizaje por transferencia (transfer learning), es posible aprovechar redes previamente entrenadas en grandes conjuntos como ImageNet y adaptarlas a nuevos problemas. Este capítulo explora cómo los modelos preentrenados permiten reducir el tiempo de desarrollo y mejorar la precisión, incluso cuando se dispone de pocos datos. También se analiza el fine-tuning como técnica para afinar los pesos de las últimas capas del modelo y adaptarlo mejor al nuevo dominio.

PREGUNTAS DE ENFOQUE

- ¿Qué es un modelo preentrenado y en qué se diferencia de entrenar desde cero?
- ¿Qué ventajas ofrece el transfer learning en problemas con pocos datos?
- ¿Cuándo es recomendable aplicar fine-tuning?
- ¿Cómo se adapta un modelo preentrenado a una tarea distinta?
- ¿Qué impacto tiene el uso de data augmentation en este contexto?

OBJETIVOS DEL CAPÍTULO

- Comprender el concepto y las aplicaciones de los modelos preentrenados.

- Explicar el funcionamiento del transfer learning y su utilidad.
- Implementar el fine-tuning de modelos para una tarea específica.
- Desarrollar una solución práctica de clasificación de imágenes reutilizando modelos existentes.
- Comparar el rendimiento entre entrenar desde cero y utilizar modelos preentrenados.

RESULTADOS DE APRENDIZAJE ESPERADOS

Al finalizar este capítulo, el lector será capaz de:

- Describir qué es un modelo preentrenado y cómo aprovecharlo para nuevas tareas.
- Aplicar transferencia de aprendizaje y fine-tuning usando Keras y TensorFlow.
- Adaptar arquitecturas como MobileNet, VGG o ResNet para clasificar imágenes personalizadas.
- Analizar los resultados de entrenamiento y validar el impacto de la estrategia elegida.
- Tomar decisiones técnicas sobre qué modelo usar según los recursos disponibles.

PROBLEMAS A RESOLVER

- ¿Cómo abordar la clasificación de imágenes cuando no se dispone de un dataset grande?
- ¿Qué modelo preentrenado conviene usar según el tamaño y tipo de datos?
- ¿Qué pasos son necesarios para ajustar un modelo ya entrenado a una nueva tarea?
- ¿Cómo evitar el sobreajuste al usar modelos complejos sobre conjuntos pequeños?
- ¿Cómo evaluar y comparar distintas estrategias de transferencia?

INTRODUCCIÓN

En los últimos años, los **modelos preentrenados** han revolucionado el desarrollo de soluciones basadas en **visión por computadora** mediante redes neuronales convolucionales (CNN). A diferencia de los modelos entrenados desde cero, los modelos preentrenados se basan en arquitecturas profundas que ya han sido **entrenadas previamente sobre grandes bases de datos**, como **ImageNet**, que contiene millones de imágenes etiquetadas pertenecientes a miles de categorías (Deng et al., 2009; Tan & Le, 2019)

Estos modelos han aprendido a **detectar patrones visuales generales**, como bordes, texturas, formas y estructuras semánticas, que son comunes en muchos tipos de imágenes. Gracias a esto, pueden ser reutilizados para nuevas tareas con un costo computacional mucho menor, especialmente útil cuando se dispone de **conjuntos de datos pequeños o medianos (**Zhuang et al., 2020).

El uso de modelos preentrenados está estrechamente ligado a un concepto clave en Deep Learning: **Transfer Learning** o **aprendizaje por transferencia**, el cual permite aprovechar el conocimiento adquirido por un modelo en una tarea original para aplicarlo a una tarea diferente pero relacionada (Tan et al., 2021).

Entre sus ventajas destacan:

⚡ **Reducción del tiempo de entrenamiento**: al no empezar desde cero, los pesos ya están ajustados con un buen punto de partida.

📉 **Mejor rendimiento con pocos datos**: especialmente útil cuando no se cuenta con grandes conjuntos de imágenes.

💻 **Menor costo computacional**: ideal para entornos sin GPU potentes o con recursos limitados.

🧠 **Mejor capacidad de generalización**: gracias al conocimiento previamente aprendido en datasets diversos.

Este capítulo aborda en profundidad el concepto de modelos preentrenados, su funcionamiento, los principales modelos disponibles (como VGG, ResNet, MobileNet y EfficientNet), y cómo integrarlos a flujos de trabajo de clasificación de imágenes utilizando herramientas como **TensorFlow y Keras**.

A través de ejemplos prácticos, comparaciones y visualizaciones, se busca que el lector desarrolle un entendimiento sólido y aplicado de cómo los modelos preentrenados pueden acelerar y mejorar significativamente proyectos de inteligencia artificial.

TRABAJOS PREVIOS Y APLICABILIDAD EN LA INDUSTRIA

Los **modelos preentrenados de redes neuronales convolucionales (CNN)** han revolucionado el desarrollo de aplicaciones en visión por computadora al permitir reutilizar representaciones aprendidas en grandes conjuntos de datos como ImageNet. A través del **transfer learning**, se aprovecha el conocimiento previamente adquirido en tareas generales para aplicarlo a problemas específicos con menos datos y menor costo computacional. Dentro de esta estrategia, el **fine-tuning** consiste en ajustar parcial o totalmente los pesos del modelo preentrenado para adaptarlo mejor al nuevo dominio, lo que ha demostrado mejorar significativamente la precisión en aplicaciones como clasificación médica, inspección industrial y sistemas de vigilancia (Kornblith et al., 2019; Tan & Le, 2021). Estudios recientes resaltan la aplicabilidad de estas técnicas en la industria, desde el diagnóstico automatizado de imágenes radiológicas hasta el control de calidad en cadenas de manufactura, mostrando su efectividad y capacidad de generalización en entornos reales (Minaee et al., 2021; Wightman et al., 2021).

PRINCIPALES MODELOS PREENTRENADOS

Si accedemos a la página https://keras.io/api/applications/ podemos observar los modelos preentrenados disponibles como se muestra en la imagen.

Model	Size (MB)	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth	Time (ms) per inference step (CPU)	Time (ms) per inference step (GPU)
Xception	88	79.0%	94.5%	22.9M	81	109.4	8.1
VGG16	528	71.3%	90.1%	138.4M	16	69.5	4.2
VGG19	549	71.3%	90.0%	143.7M	19	84.8	4.4
ResNet50	98	74.9%	92.1%	25.6M	107	58.2	4.6
ResNet50V2	98	76.0%	93.0%	25.6M	103	45.6	4.4
ResNet101	171	76.4%	92.8%	44.7M	209	89.6	5.2
ResNet101V2	171	77.2%	93.8%	44.7M	205	72.7	5.4
ResNet152	232	76.6%	93.1%	60.4M	311	127.4	6.5
ResNet152V2	232	78.0%	94.2%	60.4M	307	107.5	6.6
InceptionV3	92	77.9%	93.7%	23.9M	189	42.2	6.9
InceptionResNetV2	215	80.3%	95.3%	55.9M	449	130.2	10.0
MobileNet	16	70.4%	89.5%	4.3M	55	22.6	3.4
MobileNetV2	14	71.3%	90.1%	3.5M	105	25.9	3.8
DenseNet121	33	75.0%	92.3%	8.1M	242	77.1	5.4
DenseNet169	57	76.2%	93.2%	14.3M	338	96.4	6.3
DenseNet201	80	77.3%	93.6%	20.2M	402	127.2	6.7
NASNetMobile	23	74.4%	91.9%	5.3M	389	27.0	6.7
NASNetLarge	343	82.5%	96.0%	88.9M	533	344.5	20.0
EfficientNetB0	29	77.1%	93.3%	5.3M	132	46.0	4.9
EfficientNetB1	31	79.1%	94.4%	7.9M	186	60.2	5.6
EfficientNetB2	36	80.1%	94.9%	9.2M	186	80.8	6.5
EfficientNetB3	48	81.6%	95.7%	12.3M	210	140.0	8.8
EfficientNetB4	75	82.9%	96.4%	19.5M	258	308.3	15.1
EfficientNetB5	118	83.6%	96.7%	30.6M	312	579.2	25.3
EfficientNetB6	166	84.0%	96.8%	43.3M	360	958.1	40.4
EfficientNetB7	256	84.3%	97.0%	66.7M	438	1578.9	61.6
EfficientNetV2B0	29	78.7%	94.3%	7.2M	-	-	-
EfficientNetV2B1	34	79.8%	95.0%	8.2M	-	-	-
EfficientNetV2B2	42	80.5%	95.1%	10.2M	-	-	-
EfficientNetV2B3	59	82.0%	95.8%	14.5M	-	-	-
EfficientNetV2S	88	83.9%	96.7%	21.6M	-	-	-
EfficientNetV2M	220	85.3%	97.4%	54.4M	-	-	-

Ilustración 15. modelos preentrenados disponibles
FUENTE: (TAN & LE, 2019)

¿CÓMO ESCOGER EL MODELO PREENTRENADO IDEAL?

Aquí tienes una tabla de criterios y recomendaciones basada en **4 escenarios comunes**:

Tabla 4. La tabla presenta los modelos recomendados en función de distintos requisitos, permitiendo seleccionar la alternativa más adecuada según las necesidades específicas de implementación.

Escenario / Requisito	**Modelo recomendado**	**¿Por qué?**
✅ Tienes una GPU potente y dataset grande	ResNet50, EfficientNetB0-B3	Alta precisión y eficiencia al escalar
✅ Tienes un dataset pequeño o quieres resultados rápidos	VGG16, MobileNetV2	Simples de implementar, buenos para fine-tuning
✅ Vas a implementar en dispositivos móviles o IoT	MobileNetV2, EfficientNet-lite	Son livianos y rápidos, ideales para producción móvil
✅ Necesitas máxima precisión posible (proyecto de alto impacto)	EfficientNetB3-B7, ResNet152	Modelos de última generación con alta precisión
✅ Quieres enseñar o visualizar cómo funciona una CNN	VGG16, VGG19	Arquitectura sencilla, fácil de explicar y modificar

FUENTE: WIGHTMAN ET AL. (2021) Y TAN & LE (2019)

¿Qué factores debo considerar?

Tipo de problema

- ¿Clasificación binaria, multiclase, o regresión visual?
- ¿Problema sencillo (ej. distinguir perros y gatos) o complejo (clasificar razas, enfermedades, etc.)?

Problemas simples → MobileNet, VGG16
Problemas complejos → ResNet, EfficientNet

Tamaño del dataset

- Si tienes **pocos datos (menos de 1000 imágenes por clase)**:
  - Usa modelos preentrenados **como extractores de características** y añade tus propias capas densas.
  - Usa **fine-tuning solo en las últimas capas**.

✅ Recomendado: VGG16, MobileNetV2

Recursos computacionales

- ¿Tienes acceso a una GPU? ¿Vas a entrenar localmente o en la nube?
- ¿Tu modelo se desplegará en tiempo real o en segundo plano?

🔸 Si tienes recursos limitados: MobileNet
🔹 Si cuentas con buena infraestructura: ResNet, EfficientNet

Peso del modelo y tiempo de inferencia

- Si necesitas modelos **rápidos y livianos** (por ejemplo, para apps móviles o web):
  - Usa **MobileNetV2**, **EfficientNetB0** o versiones lite.

✅ Resumen visual (elige según lo que priorizas)

¿Qué priorizas?	**Modelo sugerido**
🏆 Máxima precisión	EfficientNetB3–B7
⚡ Velocidad de inferencia	MobileNetV2
🧠 Simplicidad para enseñar	VGG16
📱 Apps móviles / web	MobileNetV2 / EfficientNet-lite
💪 Robustez para producción	ResNet50 / EfficientNetB0

Robustez frente a ruido o cambios adversos

Algunos modelos como **ResNet y EfficientNet** tienen mejor capacidad de generalizar frente a imágenes con:

- iluminación variable
- ángulos no esperados
- presencia de ruido o fondos complejos

- Esto es importante en tareas del mundo real (cámaras de seguridad, análisis ambiental, etc.)

Tamaño de entrada esperado por el modelo

Modelos como VGG16 o ResNet esperan entradas de **224x224 px**, mientras que EfficientNetB4 puede requerir **380x380 px**. Si tus imágenes son pequeñas, un modelo que requiera imágenes grandes puede forzar interpolación y pérdida de calidad.

Tabla 5. Variables para seleccionar el mejor modelo preentrenado cnn
FUENTE: LIU, Z., LIN, Y., ET AL. (2022)

Variable de selección	**¿Por qué es importante?**	**Ejemplos de modelos más adecuados**
Precisión esperada (Top-1 en ImageNet)	Define qué tan bien clasifica el modelo en tareas generales.	EfficientNetB3-B7, ResNet101
Velocidad de inferencia	Clave para aplicaciones en tiempo real o móviles.	MobileNetV2, EfficientNet-lite
Tamaño del modelo (MB)	Modelos grandes consumen más memoria y almacenamiento.	MobileNet, EfficientNetB0
Número de parámetros	Más parámetros = más potencia, pero mayor riesgo de overfitting.	VGG16 (138M), ResNet50 (25M), MobileNet (3.4M)
Requerimientos de entrada (resolución)	Algunos modelos requieren imágenes grandes, afectando procesamiento.	VGG16 (224x224), EfficientNetB4 (380x380)
Compatibilidad con librería (Keras, PyTorch, ONNX)	No todos los modelos están disponibles o bien soportados en todos los frameworks.	ResNet, EfficientNet en TensorFlow/Keras
Facilidad de fine-tuning	Algunos modelos permiten ajustar más fácilmente las últimas capas.	VGG16, MobileNetV2
Robustez ante ruido / variaciones	Algunos modelos generalizan mejor ante condiciones adversas.	ResNet, EfficientNet
Uso en dispositivos móviles	Modelos livianos permiten inferencia eficiente en dispositivos móviles o IoT.	MobileNetV2, EfficientNet-lite

Variable de selección	¿Por qué es importante?	Ejemplos de modelos más adecuados
Tipo de problema (simple vs complejo)	Problemas simples requieren modelos menos profundos.	MobileNetV2 (simple), ResNet (complejo)
Tamaño del dataset disponible	Con pocos datos es mejor usar modelos más simples o feature extraction.	MobileNet, VGG con extracción de características
Recursos computacionales disponibles	Determina si puedes usar modelos grandes o necesitas algo liviano.	ResNet50 si hay GPU; MobileNet si no
Dominio del dataset (similar a ImageNet o no)	Si el dominio es muy distinto, hay que ajustar más capas.	Usar solo primeras capas si son imágenes médicas, satelitales, etc.
Nivel de personalización necesario	Modelos simples son más fáciles de modificar para tareas especiales.	VGG16, MobileNet (secuenciales y fáciles de adaptar)
Preferencia educativa o explicativa	Modelos sencillos son ideales para enseñanza o visualización de activaciones.	VGG16 (educación), MobileNet (práctica), ResNet (profundidad)
Nivel máximo de precisión alcanzable (fine-tuning completo)	Indica el techo de rendimiento posible tras ajustar todas las capas.	EfficientNetB4-B7, ResNet152
Tiempo estimado de entrenamiento	Afecta la viabilidad del proyecto según los recursos disponibles.	MobileNetV2 (rápido), VGG16 (intermedio), ResNet101 (más lento)

CASO DE ESTUDIO: CLASIFICACIÓN DE TIPOS DE ANIMALES (GASTOS Y PERROS) USANDO MODELOS PREENTRENADOS Y TÉCNICAS AVANZADAS

Introducción

En este caso de estudio se aborda la construcción de un sistema de **clasificación automática de gatos y perros** a partir de imágenes digitales, aplicando **Deep Learning** con **modelos preentrenados**. Se utilizarán técnicas de **Transfer Learning**, **Fine-Tuning**, **Aumento de Datos** y otros métodos modernos para optimizar el rendimiento, reducir el sobreajuste y mejorar la generalización del modelo.

Objetivo

Desarrollar un modelo que clasifique imágenes de flores en **cinco categorías**:

- gatos y
- perros

Dataset

Se utilizará el dataset público **"dog and cat"** disponible en Kaggle, que contiene:

- 2300 imágenes distribuidas en 2 carpetas, una por clase.
- Variedad de tamaños, fondos y condiciones de iluminación.

Metodología

Selección del modelo preentrenado

- **Modelo elegido**: MobileNetV2 preentrenado en ImageNet.
- **Justificación**: balance óptimo entre precisión, tamaño de modelo (~14 MB) y velocidad de inferencia.

Transfer Learning

- Se utiliza la red preentrenado **como extractor de características**.
- Se congela (trainable=False) todo el modelo base para no modificar los pesos inicialmente aprendidos.
- Se añaden nuevas capas densas adaptadas al problema de clasificación de gastos y perros.

```
for layer in modelo_base.layers:
    layer.trainable = False
```

Aumento de Datos (Data Augmentation)

- Se aplica para **simular un dataset más grande** y **reducir overfitting**.

Aumentos aplicados:

- Rotaciones aleatorias
- Zoom aleatorio
- Desplazamientos horizontales/verticales
- Reflejo horizontal

- Brillo aleatorio

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

train_datagen = ImageDataGenerator(
   rescale=1./255,
   rotation_range=30,
   zoom_range=0.2,
   width_shift_range=0.2,
   height_shift_range=0.2,
   horizontal_flip=True,
   brightness_range=[0.8,1.2],
   validation_split=0.2
)
```

Arquitectura final

- modelo_base congelado como extractor.
- Flatten para aplanar salidas.
- Dense de 128 neuronas + ReLU.
- Dropout(0.5) para regularización.
- Capa de salida Dense(5) con softmax para clasificación multiclase.

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

modelo = Sequential([
   modelo_base,
   Flatten(),
   Dense(128, activation='relu'),
   Dropout(0.5),
   Dense(5, activation='softmax')
])
```

Entrenamiento inicial

- Se entrena solo la nueva cabeza de clasificación.
- Optimización con Adam.
- Función de pérdida categorical_crossentropy.

```
modelo.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
modelo.fit(train_generator, validation_data=val_generator, epochs=10)
```

Fine-Tuning

- **Después** de entrenar la cabeza, se descongelan **las últimas capas** del modelo base para afinar aún más.
- Se usa una tasa de aprendizaje baja (1e-5) para no destruir los pesos aprendidos.

```
for layer in modelo_base.layers[-30:]:
   layer.trainable = True

from tensorflow.keras.optimizers import Adam
modelo.compile(optimizer=Adam(learning_rate=1e-5),
loss='categorical_crossentropy', metrics=['accuracy'])
modelo.fit(train_generator, validation_data=val_generator, epochs=10)
```

Tabla 6. La tabla muestra técnicas adicionales sugeridas para optimizar el proceso de entrenamiento de redes neuronales, como el early stopping y el ajuste dinámico de la tasa de aprendizaje.

Técnica	**Propósito**	**Cómo aplicarla**
Early Stopping	Evitar entrenar demasiado tiempo si el modelo deja de mejorar.	EarlyStopping(monitor='val_loss', patience=5)
ReduceLROnPlateau	Reducir la tasa de aprendizaje automáticamente si no mejora.	ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=3)
ModelCheckpoint	Guardar el mejor modelo automáticamente.	ModelCheckpoint('mejor_modelo.h5', save_best_only=True)
Learning Rate Scheduler	Ajustar dinámicamente la tasa de aprendizaje	LearningRateScheduler(lambda epoch: 1e-3 * 0.95**epoch)

Técnica	Propósito	Cómo aplicarla
	durante el entrenamiento.	

FUENTE: (GERON, 2019)

Resultados esperados

- Precisión de validación superior al 90%.
- Reducción significativa de overfitting gracias al aumento de datos y dropout.
- Modelo liviano (menos de 20 MB) listo para ser exportado a dispositivos móviles o servicios web.

Conclusiones

El uso de modelos preentrenados combinado con técnicas modernas como **Transfer Learning**, **Fine-Tuning** y **Data Augmentation** permite construir modelos potentes y eficientes incluso con datasets de tamaño medio. Además, estas estrategias reducen los costos de entrenamiento, mejoran la capacidad de generalización y acortan los tiempos de desarrollo de soluciones basadas en Deep Learning.

Será un flujo completo:

1. Carga del modelo preentrenado
2. Aumento de datos
3. Transfer Learning
4. Fine-Tuning
5. Técnicas extra (EarlyStopping y ModelCheckpoint)
6. Visualización de resultados

Código Completo del Caso de Estudio

Código

```
import tensorflow as tf

from tensorflow.keras.applications import MobileNetV2
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D
```

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt
import numpy as np
# --- PARÁMETROS ---
IMG_SIZE = (224, 224)
BATCH_SIZE = 32
EPOCHS_TL = 5  # Transfer Learning
EPOCHS_FT = 5  # Fine-Tuning
LR_TL = 1e-3
LR_FT = 1e-4
```

```
# --- DATA AUGMENTATION ---
train_gen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True
)

val_gen = ImageDataGenerator(rescale=1./255)

train_data = train_gen.flow_from_directory(
    '/content/drive/MyDrive/practicasMachineL/DEEPLEARNING/GATOSPERROS/DATASET/dogscats/train',
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='binary'
)

val_data = val_gen.flow_from_directory(
    '/content/drive/MyDrive/practicasMachineL/DEEPLEARNING/GATOSPERROS/DATASET/dogscats/valid',
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='binary',
    shuffle=False
)
```

```
# --- TRANSFER LEARNING ---
base_model = MobileNetV2(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
base_model.trainable = False
```

```
x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(128, activation='relu')(x)
output = Dense(1, activation='softmax')(x)

model = Model(inputs=base_model.input, outputs=output)
model.compile(optimizer=Adam(learning_rate=LR_TL),
              loss='binary_crossentropy',
              metrics=['accuracy'])

early_stop = EarlyStopping(patience=3, restore_best_weights=True)

print("🔵 Fase 1: Transfer Learning")
history_tl = model.fit(train_data, validation_data=val_data,
epochs=EPOCHS_TL, callbacks=[early_stop])
```

```
# --- FINE-TUNING ---
print("🟢 Fase 2: Fine Tuning")
base_model.trainable = True
for layer in base_model.layers[:-20]:  # solo últimas 20 capas se entrenan
    layer.trainable = False

model.compile(optimizer=Adam(learning_rate=LR_FT),
              loss='binary_crossentropy',
              metrics=['accuracy'])

history_ft = model.fit(train_data, validation_data=val_data,
epochs=EPOCHS_FT, callbacks=[early_stop])
```

```
# --- GUARDAR MODELO ---
model.save("/content/drive/MyDrive/practicasMachineL/DEEPLEARNING/G
ATOSPERROS/DATASET/modelo_gatos_perros2.h5")
print("✅ Modelo guardado como modelo_gatos_perros.h5")
```

```
# --- MATRIZ DE CONFUSIÓN ---
print("📊 Matriz de Confusión")
val_data.reset()
y_true = val_data.classes
y_pred = model.predict(val_data)
y_pred_classes = (y_pred > 0.5).astype(int).reshape(-1)

cm = confusion_matrix(y_true, y_pred_classes)
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
display_labels=val_data.class_indices)
```

```
disp.plot(cmap='Blues')
plt.title("Matriz de Confusión")
plt.show()

# --- GRÁFICAS DE ACCURACY Y LOSS ---
def plot_metrics(history1, history2):
    acc = history1.history['accuracy'] + history2.history['accuracy']
    val_acc = history1.history['val_accuracy'] + history2.history['val_accuracy']
    loss = history1.history['loss'] + history2.history['loss']
    val_loss = history1.history['val_loss'] + history2.history['val_loss']
    epochs = range(1, len(acc) + 1)

    plt.figure(figsize=(12, 5))

    # Accuracy
    plt.subplot(1, 2, 1)
    plt.plot(epochs, acc, label='Entrenamiento')
    plt.plot(epochs, val_acc, label='Validación')
    plt.title('Precisión (accuracy)')
    plt.xlabel('Épocas')
    plt.ylabel('Precisión')
    plt.legend()

    # Loss
    plt.subplot(1, 2, 2)
    plt.plot(epochs, loss, label='Entrenamiento')
    plt.plot(epochs, val_loss, label='Validación')
    plt.title('Pérdida (loss)')
    plt.xlabel('Épocas')
    plt.ylabel('Pérdida')
    plt.legend()

    plt.tight_layout()
    plt.show()

plot_metrics(history_tl, history_ft)
```

Explicacion del código

Importación de librerías

Este bloque carga todas las librerías necesarias:

- TensorFlow y Keras: para construir, entrenar y gestionar el modelo.
- MobileNetV2: modelo convolucional preentrenado eficiente.
- ImageDataGenerator: carga y transforma imágenes desde carpetas.
- Adam y EarlyStopping: optimizador y callback para evitar sobreentrenamiento.
- sklearn y matplotlib: evaluación y visualización de métricas.

Definición de parámetros

IMG_SIZE = (224, 224)
BATCH_SIZE = 32
EPOCHS_TL = 5
EPOCHS_FT = 5
LR_TL = 1e-3
LR_FT = 1e-4

Se definen los valores clave del flujo:

- Tamaño de imagen requerido por MobileNetV2 (224x224).
- Tamaño de lote (batch).
- Épocas para las dos fases del entrenamiento: Transfer Learning y Fine-Tuning.
- Tasas de aprendizaje diferentes para cada fase.

Aumento y carga de datos

Mediante ImageDataGenerator, se crean dos generadores:

- train_gen aplica **aumento de datos** para generalizar mejor.
- val_gen solo normaliza las imágenes.

Las imágenes se cargan desde carpetas locales divididas en **train** y **valid**, con clases **gatos** y **perros**.

Transfer Learning (fase 1)

Se reutiliza el modelo **MobileNetV2 preentrenado** (sin la cabeza de clasificación original) y se congela completamente.

Se añaden nuevas capas:

- GlobalAveragePooling2D para aplanar la salida.
- Dense(128) con ReLU para aprender nuevas representaciones.
- Dense(1) con softmax (aquí lo correcto sería ***sigmoid*** para binaria).

Luego se compila y entrena el modelo solo con las nuevas capas durante 5 épocas.

Fine-Tuning (fase 2)

Después de transfer learning:

- Se **descongela parcialmente** MobileNetV2 (últimas 20 capas).
- Se recompila el modelo con una tasa de aprendizaje más baja.
- Se entrena por otras 5 épocas ajustando tanto las capas nuevas como parte del modelo base.

Esto mejora el rendimiento al adaptar mejor el conocimiento preentrenado al nuevo dataset.

Guardar el modelo

model.save("...modelo_gatos_perros2.h5")

Se guarda el modelo entrenado en Google Drive para reutilizarlo sin volver a entrenar.

Matriz de confusión

- Se generan predicciones para el conjunto de validación.
- Se convierte cada salida probabilística en una clase (0 o 1).
- Se construye la **matriz de confusión** para ver aciertos y errores.
- Se visualiza con ConfusionMatrixDisplay.

Esto permite entender **cómo clasifica el modelo**, y si confunde más a una clase que a otra.

Gráfica de métricas (accuracy y loss)

La función plot_metrics:

- Combina los resultados de ambas fases (history_tl + history_ft).
- Muestra en dos subgráfico:
  - La evolución de la precisión (accuracy) en entrenamiento y validación.
  - La evolución de la pérdida (loss) en entrenamiento y validación.

Es una herramienta visual clave para analizar si el modelo mejora y si hay signos de sobreajuste.

¿Qué incluye este flujo completo?

Parte	**Acción Realizada**
Aumento de datos	Mejora generalización
Transfer Learning	Entrenamiento solo de nuevas capas
Fine-Tuning parcial	Descongelar últimas capas para afinar
EarlyStopping + ReduceLROnPlateau	Mejorar entrenamiento automáticamente
ModelCheckpoint	Guardar el mejor modelo
Visualización final	Analizar cómo evolucionó el aprendizaje

🎓 ¿Qué es Transfer Learning y fine tuning?

Transfer Learning es cuando usamos un modelo ya entrenado (por ejemplo, en millones de imágenes) como punto de partida para resolver **otro problema similar**, como clasificar perros y gatos.

🎯 ¿Por qué usarlo?

- Porque **ya aprendió características útiles** (como bordes, formas, texturas).

- Solo necesitas **ajustarlo un poco** para tu tarea (por ejemplo: gato vs. perro).
- Así puedes entrenar con **menos datos y más rápido**.

🎨 Analogía intuitiva

Imagina que un estudiante ya sabe **identificar animales salvajes** (leones, cebras, elefantes).
Ahora le pides que **identifique si una imagen tiene un perro o un gato**.

🔁 No necesita empezar desde cero.
Ya **sabe lo que son ojos, orejas, patas, pelaje**…
Solo necesita aprender a **diferenciar específicamente perro vs gato**.

🛠️ ¿Cómo se refleja eso en el código?

Paso a paso explicado del código con Transfer Learning y Fine-Tuning

Cargar modelo preentrenado (MobileNetV2)

base_model = MobileNetV2(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

¿Qué hace?

Carga el modelo MobileNetV2 ya entrenado con más de 1 millón de imágenes (ImageNet).
include_top=False significa que **no queremos su última capa de clasificación** (porque queremos entrenar la nuestra: gato vs perro).

📦 **MobileNetV2 sabe detectar cosas como ojos, hocicos, pelaje, forma general del animal.**

Congelar las capas preentrenadas

```
base_model.trainable = False
```

🔹 Así **no modificamos lo que ya aprendió** (evitamos sobreentrenamiento innecesario al inicio). Solo vamos a entrenar la parte nueva que añadimos.

Añadir nuevas capas para nuestro problema

```
x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(128, activation='relu')(x)
output = Dense(1, activation='sigmoid')(x)
model = Model(inputs=base_model.input, outputs=output)
```

🔹 Aquí construimos nuestro propio "clasificador final":

- GlobalAveragePooling2D: reduce la información a lo más importante.
- Dense(128): capa oculta para aprendizaje.
- Dense(1, sigmoid): salida binaria → 0 = gato, 1 = perro.

Compilar y entrenar solo la nueva parte

```
model.compile(optimizer=Adam(learning_rate=0.001), ...)
history_tl = model.fit(...)
```

🔹 Entrenamos **solo la parte nueva**. Las capas profundas no se modifican aún.

Fine-tuning: desbloquear capas profundas

```
base_model.trainable = True
for layer in base_model.layers[:-20]:
    layer.trainable = False
```

🔹 Ahora que nuestro modelo está "calentado", desbloqueamos las **últimas 20 capas** para que **se adapten mejor a perros y gatos**. Esto es el famoso:

🛠️ **Fine-Tuning** = ajustar finamente algunas partes del modelo base.

Reentrenar con learning rate más bajo

```
model.compile(optimizer=Adam(learning_rate=0.0001), ...)
history_ft = model.fit(...)
```

🔹 Ahora el modelo ajusta detalles **de todo el conocimiento aprendido** para especializarse.

VISUALIZAR RESULTADOS

- Matriz de confusión: muestra los aciertos y errores.
- Gráficos de accuracy y loss: muestran cómo aprendió en cada época.

RESUMEN CLARO

Fase	**Qué hace**
Transfer Learning	Usa un modelo entrenado como base
Congelar capas	No se tocan las capas base al inicio
Añadir nuevas capas	Se entrena solo lo nuevo (tu clasificador)
Fine-Tuning	Se desbloquean capas profundas para especializar el modelo
Resultado	Modelo adaptado a tus datos con menos esfuerzo

¿QUÉ HACE EARLYSTOPPING?

```
early_stop = EarlyStopping(patience=3, restore_best_weights=True)
```

Esta línea **crea un mecanismo para detener automáticamente el entrenamiento del modelo** cuando **ya no mejora** en la validación, y así evitar que el modelo aprenda de más y se sobreentrene (overfitting).

🔍 ¿Qué significan los argumentos?

🔸 *PATIENCE=3*

Espera **3 épocas sin mejora** en la validación antes de detener el entrenamiento.

Ejemplo:

- Si la precisión (val_accuracy) no mejora durante 3 épocas seguidas, se detiene.

🔸 *RESTORE_BEST_WEIGHTS=TRUE*

Al detener el entrenamiento, el modelo vuelve automáticamente a los **pesos que lograron la mejor precisión** durante las épocas previas.

🔁 Así, no te quedas con el modelo "de la última época", sino con el **mejor modelo validado**.

🎯 ¿Por qué es útil?

Problema	**Solución con EarlyStopping**
El modelo entrena muchas épocas innecesarias	Se detiene solo cuando ya no mejora
El modelo se sobreentrena (memoriza en vez de generalizar)	Se detiene antes y guarda el mejor estado

Resultado:

- Menos tiempo de entrenamiento
- Mejores resultados en datos reales
- Modelo más generalizable

RESUMEN FINAL DEL CAPÍTULO 2

En este capítulo se exploraron los **modelos preentrenados** como una herramienta clave dentro del deep learning moderno, destacando su capacidad de reutilizar representaciones previamente aprendidas para resolver nuevas tareas con mayor eficiencia. Se analizó el enfoque de **transfer learning**, el cual permite adaptar estos modelos a dominios específicos, reduciendo el costo computacional y el tiempo de entrenamiento.

Asimismo, se profundizó en la técnica de **fine-tuning**, resaltando cómo pequeños ajustes en las capas finales del modelo mejoran su rendimiento en problemas concretos sin necesidad de entrenar desde cero. Finalmente, se incluyó un ejercicio práctico en **Google Colab** que permitió aplicar de manera experimental estos conceptos, brindando a los estudiantes una experiencia directa con la implementación y adaptación de arquitecturas preentrenadas a diferentes contextos.

En conjunto, el capítulo evidencia la importancia de los modelos preentrenados y de las técnicas de transferencia como pilares fundamentales para optimizar el uso de redes neuronales profundas en aplicaciones reales, constituyendo un puente entre la teoría y la práctica en el aprendizaje profundo.

AUTOEVALUACIÓN

Instrucciones: Reflexiona sobre cada pregunta y verifica si puedes responder con claridad. Si no, revisa las secciones correspondientes del capítulo.

1. ¿Qué es un modelo preentrenado y en qué tipo de dataset suele entrenarse originalmente?
2. ¿Qué diferencia hay entre usar un modelo preentrenado "congelado" y aplicar ***fine-tuning***?
3. ¿Qué ventajas ofrece el ***transfer learning*** frente al entrenamiento desde cero?
4. ¿Cómo seleccionas qué capas ajustar durante el ***fine-tuning***?
5. ¿Qué pasos se deben seguir para adaptar un modelo como MobileNetV2 a un nuevo conjunto de clases?
6. ¿Qué papel juega la normalización y el ***data augmentation*** al reutilizar modelos preentrenados?
7. ¿Cómo se evalúa si el modelo está mejorando luego del fine-tuning?
8. ¿Cuáles son los riesgos de sobreajuste al aplicar transfer learning con pocos datos?

EJERCICIOS DE APLICACIÓN DE CONTENIDOS

Ejercicio 1: Implementar transferencia de aprendizaje

- Utiliza un modelo preentrenado como **ResNet50** o **EfficientNet** para una nueva tarea de clasificación (por ejemplo, flores o señales de tránsito).
- Sustituye las capas superiores por nuevas capas densas adaptadas a tu problema.
- Entrena el modelo y documenta los resultados.

Ejercicio 2: Comparación de estrategias

- Compara dos enfoques sobre el mismo dataset: (a) modelo entrenado desde cero y (b) modelo preentrenado con fine-tuning.
- Registra precisión, pérdida y tiempos de entrenamiento.
- Analiza qué enfoque fue más eficiente y por qué.

Ejercicio 3: Visualización de predicciones

- Muestra imágenes de validación con su clase verdadera y la clase predicha por el modelo.
- Comenta algunos aciertos y errores, explicando posibles causas desde el punto de vista del aprendizaje transferido.

SUGERENCIA DE PROYECTO AUTÓNOMO

Título: *Clasificador personalizado con modelos preentrenados*

1. Elige un nuevo conjunto de imágenes (por ejemplo: frutas, expresiones faciales, animales salvajes).
2. Crea una estructura de carpetas compatible (train/ y valid/)
3. Utiliza un modelo preentrenado como **MobileNetV2** o **VGG16** con ***transfer learning***.
4. Aplica ***fine-tuning*** a las últimas capas.
5. Evalúa el desempeño, genera gráficas y discute los resultados.
6. Crea un pequeño informe explicando las decisiones técnicas tomadas.

REFERENCIAS BIBLIOGRÁFICAS

Tan, M., & Le, Q. V. (2019). EfficientNet: Rethinking model scaling for convolutional neural networks. In ***Proceedings of the 36th International Conference on Machine Learning*** (pp. 6105–6114). https://arxiv.org/abs/1905.11946

Zhuang, F., Qi, Z., Duan, K., Xi, D., Zhu, Y., Zhu, H., ... & He, Q. (2020). A comprehensive survey on transfer learning. ***Proceedings of the IEEE, 109***(1), 43–76. https://doi.org/10.1109/JPROC.2020.3004555

Tan, C., Sun, F., Kong, T., Zhang, W., Yang, C., & Liu, C. (2021). A survey on deep transfer learning. In ***International Conference on Artificial Neural Networks*** (pp. 270–279). Springer. https://doi.org/10.1007/978-3-030-86383-8_22

Wightman, R., Touvron, H., & Jegou, H. (2021). ResNet strikes back: An improved training procedure in timm. ***arXiv preprint*** arXiv:2110.00476. https://arxiv.org/abs/2110.00476

Liu, Z., Lin, Y., Cao, Y., Hu, H., Wei, Y., Zhang, Z., ... & Guo, B. (2022). A ConvNet for the 2020s. In ***Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*** (pp. 11976–11986). https://arxiv.org/abs/2201.03545

Géron, A. (2019). ***Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*** (2nd ed.). O'Reilly Media.

Kornblith, S., Shlens, J., & Le, Q. V. (2019). Do better ImageNet models transfer better? Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2661–2671. https://doi.org/10.1109/CVPR.2019.00277

Tan, M., & Le, Q. V. (2021). EfficientNetV2: Smaller models and faster training. Proceedings of the International Conference on Machine Learning (ICML), 10096–10106. https://arxiv.org/abs/2104.00298

Minaee, S., Boykov, Y., Porikli, F., Plaza, A., Kehtarnavaz, N., & Terzopoulos, D. (2021). Image segmentation using deep learning: A survey. IEEE Transactions on Pattern Analysis and Machine Intelligence, 44(7), 3523–3542. https://doi.org/10.1109/TPAMI.2021.3059968

Wightman, R., Touvron, H., & Jégou, H. (2021). ResNet strikes back: An improved training procedure in timm. arXiv preprint. https://arxiv.org/abs/2110.00476

CAPÍTULO 3. DETECCIÓN DE OBJETOS CON YOLOV8

RESUMEN DEL CAPÍTULO

En este capítulo se abordan los fundamentos y aplicaciones de los modelos de detección de objetos en imágenes, con un enfoque en la arquitectura YOLOv8. Se explica su funcionamiento, ventajas frente a versiones anteriores, y el proceso de entrenamiento con datos personalizados. A través de un caso de estudio en el ámbito agrícola, se entrena un modelo YOLOv8 para detectar enfermedades en mazorcas de cacao, demostrando su utilidad práctica en la automatización del diagnóstico visual.

INTRODUCCIÓN

La detección de objetos es una de las tareas más avanzadas en visión por computadora. A diferencia de la clasificación, este tipo de modelos localiza y clasifica múltiples objetos dentro de una misma imagen. YOLO (You Only Look Once) ha sido una de las arquitecturas más populares por su eficiencia y velocidad. En este capítulo nos centraremos en YOLOv8, la versión más reciente y optimizada, explorando su estructura, ventajas, y cómo puede aplicarse a problemas del mundo real como el monitoreo agrícola. A partir de un dataset de mazorcas de cacao, mostraremos cómo preparar los datos, entrenar el modelo y evaluar su desempeño.

PREGUNTAS DE ENFOQUE

- ¿Qué diferencia existe entre clasificación de imágenes y detección de objetos?
- ¿Cómo funciona internamente la arquitectura YOLOv8?
- ¿Qué ventajas ofrece YOLOv8 frente a versiones anteriores o modelos similares?
- ¿Qué se necesita para entrenar un modelo YOLOv8 con datos propios?
- ¿Cómo se evalúa el desempeño de un modelo de detección de objetos?
- ¿Cómo puede aplicarse la detección de objetos en contextos agrícolas?

OBJETIVOS DEL CAPÍTULO

- Explicar el concepto y aplicación de la detección de objetos en imágenes.
- Comprender la arquitectura y funcionamiento de YOLOv8.
- Preparar un dataset personalizado para tareas de detección.
- Entrenar y evaluar un modelo YOLOv8 para detectar enfermedades en mazorcas de cacao.
- Interpretar visualmente los resultados obtenidos a través de predicciones.

RESULTADOS DE APRENDIZAJE ESPERADOS

Al finalizar este capítulo, el lector será capaz de:

- Diferenciar entre clasificación, segmentación y detección de objetos.
- Describir la arquitectura YOLOv8 y sus componentes clave.
- Usar herramientas como Ultralytics YOLO para entrenar modelos con datos propios.
- Aplicar un modelo entrenado para detectar múltiples objetos en tiempo real.
- Evaluar y mejorar el rendimiento de un modelo de detección en un contexto aplicado.

PROBLEMAS A RESOLVER

- ¿Cómo adaptar un modelo de detección de objetos a un nuevo dominio como la agricultura?
- ¿Qué se debe tener en cuenta para etiquetar correctamente un dataset de detección?
- ¿Cómo mitigar el sobreajuste en un modelo YOLO entrenado con pocos datos?
- ¿Qué métricas permiten validar la calidad de las detecciones?
- ¿Cómo automatizar procesos de monitoreo visual con deep learning?

TRABAJOS PREVIOS Y APLICABILIDAD EN LA INDUSTRIA

Las arquitecturas **YOLOv8** representan el estado del arte en detección de objetos en tiempo real, integrando mejoras significativas en precisión, velocidad y facilidad de uso respecto a versiones anteriores. Su diseño modernizado incluye un backbone mejorado, un cuello FPN + PAN para detección en múltiples escalas y enfoque *anchor-free*, lo que permite un rendimiento optimizado tanto en estándar de benchmarks como COCO o Roboflow 100 (Yaseen, 2024). Los modelos pre-entrenados permiten aplicar **transfer learning con facilidad**, incluso con pocas líneas de código, facilitando su adaptación a conjuntos de datos específicos mediante fine-tuning controlado por capas congeladas (Ultralytics, 2024; StackOverflow, 2024). En el ámbito industrial, YOLOv8 ya se utiliza en una variedad de aplicaciones reales: desde conteo de personas en entornos urbanos y gestión de inventarios en retail, hasta monitoreo de fugas y tanques en la industria petrolera, e implementaciones de seguridad en sites de construcción (Vina, 2024; Solawetz & Francesco, 2024). Además, investigaciones recientes han demostrado la posibilidad de optimizar YOLOv8 para detección de objetos peligrosos en línea (Jahan et al., 2025), conducción distraída (Elshamy et al., 2024) o detección de caídas en entornos industriales (Pereira, 2024), consolidando su relevancia en sistemas de seguridad y automatización.

DEFINICIÓN.

YOLO (siglas de **You Only Look Once**) es un modelo de **detección de objetos** en imágenes y video que puede decirte:

1. **Qué objetos hay** en una imagen (por ejemplo: perro, semáforo, monilia…)
2. **Dónde están** esos objetos (marcándolos con un recuadro)
3. **Con qué certeza** fueron detectados (confianza del modelo)

YOLO es una Red Neuronal Convolucional (CNN) especializada en **detección de objetos**, está basada en una arquitectura de **convoluciones**, igual que otros modelos como ResNet o VGG, pero con un diseño **optimizado para detectar objetos** en una sola pasada (one shot) (Bochkovskiy et al., 2020; Jocher et al., 2023).

¿Por qué se llama "You Only Look Once"?

Porque, a diferencia de otros métodos más lentos, YOLO **analiza la imagen solo una vez** y:

- La **divide en una cuadrícula**
- En cada celda predice:
  - Si hay un objeto
  - Qué objeto es
  - Y las coordenadas del cuadro que lo contiene

🎯 ¿Qué lo hace especial?

Característica	**Explicación breve**
🔁 Rápido	Detecta objetos **en tiempo real**
🧠 Todo en uno	Predice **clase + posición** al mismo tiempo
📦 Preciso	Funciona bien con objetos múltiples

📷 Ejemplo de salida YOLO:

Imagina esta imagen:

Foto de una mazorca de cacao con enfermedad

YOLO detecta:

- "monilia" con 94% de confianza → dibuja un cuadro verde en esa zona
- "phytophthora" con 82% → otro cuadro
- "sano" con 99% → otro cuadro

🔧 ¿Dónde se usa?

- 🚗 Autos autónomos (detectar peatones, señales, autos)
- 🛡️ Cámaras de seguridad (reconocer intrusos)
- 🧪 Agricultura (detectar enfermedades en plantas)

- 🐾 Detección de animales salvajes

📦 Versiones de YOLO

- YOLOv3, YOLOv4, YOLOv5 (PyTorch no oficial), YOLOv6
- **YOLOv8** (actual, desarrollado por Ultralytics)

Resumen simple:

YOLO es una técnica inteligente de visión por computadora que permite encontrar rápidamente **qué hay y dónde está** en una imagen, **todo en una sola pasada**.

⚙️ ¿Cómo funciona como CNN?

1. **Entrada**: Imagen (por ejemplo, de 640x640)
2. **Capas convolucionales**:
    - Extraen características como bordes, texturas, formas
    - Estas capas aprenden **qué cosas parecen un perro, una hoja enferma, etc.**
3. **Capa de detección**:
    - Divide la imagen en una cuadrícula
    - En cada celda predice:
        - Si hay un objeto
        - Qué clase es
        - Las coordenadas del bounding box

📦 ¿Qué arquitectura usa?

Depende de la versión de YOLO:

Versión	Backbone (CNN base)
YOLOv3	Darknet-53 (Redmon et al., 2018)
YOLOv5	CSPDarknet + Focus Layer
YOLOv8	Nueva arquitectura basada en **CNN + módulos tipo Transformer (C2f)** para mejorar velocidad y precisión

📦 ¿QUÉ VERSIONES HAY EN YOLOV8?

YOLOv8 tiene **varias versiones del mismo modelo** que varían en **tamaño**, **velocidad** y **precisión**

Tabla 7. versiones de yolov8
FUENTE: (JOCHER ET AL., 2023)

Versión	**Significado**	**Tamaño (↓)**	**Velocidad (↑)**	**Precisión (↑)**	**Ideal para...**
yolov8n.pt	Nano	Muy bajo	Muy alta	Menor	Celulares, apps ligeras, IoT
yolov8s.pt	Small	Bajo	Alta	Media	Proyectos rápidos o medianos
yolov8m.pt	Medium	Medio	Media	Buena	Buen balance de rendimiento
yolov8l.pt	Large	Alto	Más lento	Muy buena	Detectar objetos con alta precisión
yolov8x.pt	Extra Large	Muy alto	Más lento	Excelente	GPU potente, tareas complejas

¿Qué es yolov8n.pt?

- **yolov8** → es la versión **8** de YOLO (la más reciente desarrollada por Ultralytics).
- **n** → significa **Nano**, es decir, la versión más **ligera y rápida**, ideal para dispositivos con poca capacidad.
- **.pt** → indica que el archivo es un **modelo preentrenado en formato PyTorch**.

Entonces:

🔹 yolov8n.pt = **YOLO versión 8, modelo Nano, entrenado previamente, en formato PyTorch**

✅ Si lo descargas con este código el modelo es preentrenado:

```
from ultralytics import YOLO
model = YOLO('yolov8n.pt')
```

✔️ Sí, **es un modelo preentrenado en el dataset COCO** (80 clases como 'person', 'dog', 'car', etc.).

¿Cuál deberías usar?

- Si vas a **probar rápido o usar en dispositivos normales** → yolov8n.pt o yolov8s.pt
- Si tienes buena GPU y quieres más precisión → yolov8m.pt, yolov8l.pt, o yolov8x.pt

También existen variantes:

- yolov8n-cls.pt → para clasificación de imágenes
- yolov8n-seg.pt → para segmentación de objetos
- yolov8n-pose.pt → para estimación de poses humanas

📦 **¿Y si quiero cargar un modelo de Yolo** NO PREENTRENADO**?**

Puedes crear uno desde cero con:

```
model = YOLO('yolov8n.yaml')  # Define la arquitectura, pero sin pesos preentrenados
model.train(data='tu_data.yaml', epochs=50)
```

Aquí, estás entrenando **desde cero**.

✅ Resumen

Archivo	**¿Preentrenado?**	**¿Incluye clases COCO?**
yolov8n.pt	✅ Sí	✅ Sí (80 clases)
yolov8n.yaml	❌ No (solo arquitectura)	❌ (tú lo entrenas)

🧰 Requisitos previos

1. **Instalar Ultralytics YOLOv8**:

   pip install ultralytics

   Si la instalación la estas ejecutando en google colab y la estructura de carpetas que se generan en el entrenamiento no se pierdan sugiero primero cambiarte a una ruta del drive que esté utilizando, por ejemplo

   ```
   %cd /content/drive/MyDrive/practicasMachineL/DEEPLEARNING/yolov8
   !pip install ultralytics
   ```

2. **Estructura del dataset**: Asegúrate de que tu conjunto de datos siga la estructura esperada por YOLOv8:

```
dataset/
├── images/
│   ├── train/
│   └── val/
├── labels/
│   ├── train/
│   └── val/
├── data.yaml
```

Ilustración 16. estructura de carpetas para un modelo yolo
FUENTE: ELABORACION PROPIA

Cada imagen debe tener un archivo de etiqueta correspondiente en formato YOLO (.txt) con las anotaciones de los objetos.

3. **Archivo de configuración (cacao.yaml)**: Crea un archivo YAML que describa tu conjunto de datos:

   path: /ruta/a/tu/dataset
   train: images/train
   val: images/val
   nc: 3
   names: ['monilia', 'phytophthora', 'sano']

```
train: /content/drive/MyDrive/yolov8/yolov5/data/images/train
val: /content/drive/MyDrive/yolov8/yolov5/data/images/valid

nc: 3 # Número de clases
names: ['monilia', 'fito', 'sano']  # Nombres de las clases
```

Reemplaza /ruta/a/tu/dataset con la ruta real a tu conjunto de datos.

🚀 ENTRENAMIENTO DEL MODELO

Utiliza el siguiente código en Python para entrenar tu modelo:

```
from ultralytics import YOLO

# Cargar el modelo preentrenado
model = YOLO('yolov8n.pt')  # Puedes usar 'yolov8s.pt' o 'yolov8m.pt' según tus necesidades

# Entrenar el modelo
model.train(data='cacao.yaml', epochs=50, imgsz=640, batch=16)
```

📊 EVALUACIÓN DEL MODELO

Después del entrenamiento, evalúa el rendimiento del modelo:

```
# Evaluar el modelo
metrics = model.val()
print(metrics)
```

Esto proporcionará métricas como precisión, recall y mAP (mean Average Precision).

🔍 Inferencia en nuevas imágenes

Para realizar predicciones en nuevas imágenes:

```
# Realizar inferencia
results = model.predict(source='ruta/a/imagen.jpg', save=True, conf=0.25)
```

Esto generará una imagen con las detecciones y la guardará en el directorio de resultados.

🖼️ Visualización de resultados

Si deseas visualizar los resultados directamente en tu script:

```
import cv2
from matplotlib import pyplot as plt

# Cargar la imagen con las detecciones
img = cv2.imread('runs/detect/predict/imagen.jpg')
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

# Mostrar la imagen
plt.imshow(img)
plt.axis('off')
plt.show()
```

📂 ESTRUCTURA GENERADA DE LOS RESULTADOS DEL ENTRENAMIENTO: RUNS/DETECT/TRAIN/

Cuando ejecutas el entrenamiento con YOLOv8 usando automáticamente **crea una estructura de carpetas** para guardar **todos los resultados del entrenamiento**, métricas, pesos, gráficas, etc.

```
runs/
└── detect/
    └── train/
        ├── weights/
        │   ├── best.pt
        │   └── last.pt
        ├── results.png
        ├── confusion_matrix.png
        ├── P_curve.png
        ├── PR_curve.png
        ├── labels.jpg
        ├── F1_curve.png
        ├── opt.yaml
        ├── args.yaml
        ├── train_batch0.jpg
        └── ...
```

¿Qué contiene cada archivo?

Tabla 8. contenido de los archivos generados al entrenar yolo

Archivo / Carpeta	**¿Qué es?**
weights/best.pt	✅ Modelo con mejor precisión en validación (¡el que debes usar!)
weights/last.pt	Último modelo al final del entrenamiento
results.png	Gráfica combinada de precisión, recall, loss, etc.
confusion_matrix.png	Matriz de confusión (predicciones vs verdaderos)
train_batch0.jpg	Vista previa de algunas imágenes + predicciones durante entrenamiento
labels.jpg	Distribución de clases en tus datos (cuántas veces aparece cada una)
P_curve.png / R_curve.png	Curvas de Precisión / Recall
F1_curve.png	Curva F1 (balance entre precisión y recall)
opt.yaml / args.yaml	Parámetros del entrenamiento (dataset, epochs, tamaño de imagen, etc.)

FUENTE: (JOCHER ET AL., 2023)

📦 Carpeta weights/

Aquí están los modelos entrenados:

- best.pt → úsalo para predicción final
- last.pt → puedes retomarlo si detuviste el entrenamiento

👁️ ***PARA VISUALIZAR RESULTADOS:***

```
from IPython.display import Image
Image('runs/detect/train/results.png')
```

🎯 Resumen

YOLOv8 genera automáticamente una carpeta con:

- 🧠 Modelos entrenados
- 📊 Gráficas de desempeño
- 🔍 Ejemplos de inferencia
- 📝 Parámetros del experimento

Todo centralizado en runs/detect/train/

¿Quieres que te explique cómo usar best.pt para hacer predicciones personalizadas ahora?

✅ **PASO A PASO PARA HACER UNA PREDICCIÓN**

🔹 1. Importar la librería

from ultralytics import YOLO

2. Cargar el modelo entrenado

Asegúrate de haber guardado tu modelo como best.pt (dentro de runs/detect/train/weights/):

model = YOLO('runs/detect/train/weights/best.pt')

🔹 3. Hacer la predicción sobre una imagen nueva

Solo reemplaza 'ruta/a/tu/imagen.jpg' con la ruta real de la imagen que quieres analizar:

results = model.predict(source='ruta/a/tu/imagen.jpg', save=True, conf=0.3)

- save=True → Guarda la imagen con los resultados
- conf=0.3 → Umbral de confianza (ajústalo si quieres que sea más o menos exigente)

🔹 4. Ver la imagen con detecciones

Después de ejecutar, YOLO guarda la imagen con las cajas en:

runs/detect/predict/imagen.jpg

Puedes mostrarla así (en Jupyter o Colab):

```
from PIL import Image
import matplotlib.pyplot as plt

# Cargar y mostrar la imagen
img = Image.open('runs/detect/predict/imagen.jpg')
plt.imshow(img)
plt.axis('off')
plt.show()
```

🎯 ¿Y si quieres hacer predicción en una carpeta de imágenes?

Solo cambia source='ruta/a/carpeta/', y YOLO procesará todas las imágenes dentro.

Cuando ejecutas esta línea:

```
results = model.predict(source='ruta/a/tu/imagen.jpg', save=True, conf=0.3)
```

🟩 **YOLOv8 guarda automáticamente la imagen con los resultados** en la siguiente ruta:

runs/detect/predict/

¿Cómo encontrar el archivo?

Si tu imagen original se llama foto1.jpg, YOLO generará una imagen editada con cajas de detección en:

runs/detect/predict/foto1.jpg

🔁 Si ejecutas varias veces el mismo comando, y no cambias el nombre del directorio con save_dir=..., YOLO sobreescribirá ese contenido.

📌 Si quieres definir otra carpeta personalizada:

Puedes agregar el parámetro save_dir='ruta/salida':

```
results = model.predict(source='ruta/a/tu/imagen.jpg', save=True, save_dir='mis_resultados/')
```

Eso guardará la imagen procesada en mis_resultados/.

La carpeta runs/detect/predict/ **no necesita existir previamente**. YOLOv8 la **crea automáticamente** la **primera vez que haces una predicción**.

¿QUÉ HACE YOLO AL PREDECIR?

Cuando ejecutas:

results = model.predict(source='ruta/a/imagen.jpg', save=True)

YOLOv8:

1. Busca si ya existe la carpeta runs/detect/predict/
2. Si **no existe**, la **crea automáticamente**
3. Si **ya existe**, guarda los resultados dentro **y sobrescribe** los archivos anteriores (a menos que uses save_dir)

📁 *SI PREDICES VARIAS VECES SEGUIDAS...*

- **Primera vez**: crea runs/detect/predict/
- **Segunda vez**: sobreescribe el contenido anterior
- Si quieres mantener los resultados anteriores separados, haz esto:

model.predict(source='ruta/a/imagen.jpg', save=True, save_dir='runs/detect/mi_prediccion1')

👁️ ¿DÓNDE VER LOS RESULTADOS?

Navega a runs/detect/predict/ en tu explorador de archivos o usa Python para mostrarlo:

```
from PIL import Image
import matplotlib.pyplot as plt

img = Image.open('runs/detect/predict/imagen.jpg')
plt.imshow(img)
plt.axis('off')
plt.show()
```

🏁 DIFERENCIA ENTRE BEST.PT Y LAST.PT

Archivo	¿Qué es?
best.pt	✅ Es el modelo que tuvo **mejor desempeño en validación** (mayor mAP o menor loss según el caso). Es el que **deberías usar para hacer predicciones**.
last.pt	📦 Es el modelo **al final del entrenamiento**, después de la última época. Puede o no ser mejor que el best.pt.

🔍 ¿Cómo se decide cuál es el "mejor"?

Durante el entrenamiento, YOLO evalúa el modelo en el conjunto de validación y mide métricas como:

- **mAP50** o **mAP50-95**
- **Precisión** y **Recall**

Si en alguna época se mejora el mejor valor hasta ese momento, YOLO guarda una copia en best.pt.

¿Cuál deberías usar?

| Para predicción/inferencia | Usa **best.pt** ✅ | | Para continuar entrenamiento (resume) | Puedes usar **last.pt** |

EJEMPLO:

```
model = YOLO("runs/detect/train/weights/best.pt")
results = model.predict(source='imagen.jpg')
```

✅ CÓDIGO PARA CONTINUAR EL ENTRENAMIENTO DESDE LAST.PT

```
from ultralytics import YOLO

# Cargar el modelo desde la última época
model = YOLO('runs/detect/train/weights/last.pt')

# Continuar entrenamiento (resume)
model.train(resume=True)
```

🧠 ¿Qué hace resume=True?

- Carga automáticamente:
  - los **pesos** (last.pt)
  - el **opt.yaml** (parámetros anteriores del entrenamiento)
  - el progreso anterior (épocas ya entrenadas)
- Continúa desde la siguiente época

📝 ¿Qué debes tener?

En la carpeta runs/detect/train/ deben estar:

- weights/last.pt
- opt.yaml
- Historial del entrenamiento

🔄 Si quieres cambiar alguna configuración al reanudar (ej. más épocas):

```
model.train(resume=True, epochs=100)
```

Esto continuará entrenando hasta completar 100 épocas (no suma, sino reemplaza el total).

✅ 🔁 Reentrenar desde best.pt con nuevos parámetros o datos

```
from ultralytics import YOLO

# Cargar el modelo ya entrenado
model = YOLO('runs/detect/train/weights/best.pt')

# Reentrenar con nuevos datos
model.train(
   data='nuevo_dataset.yaml',  # tu nuevo conjunto de datos
   epochs=50,
   imgsz=640,
   batch=16,
   project='runs',
   name='reentreno_con_best',
   exist_ok=True  # para sobrescribir si ya existe esa carpeta
)
```

✅ ¿Qué hace este código?

- Toma el modelo best.pt como punto de partida
- Entrena **desde cero en el nuevo dataset**

- Guarda los nuevos resultados en runs/detect/reentreno_con_best/

📝 Notas:

- **No uses resume=True**, porque eso busca continuar exactamente el experimento anterior.
- Puedes usar esto también si simplemente quieres **seguir entrenando con el mismo dataset** pero con otras configuraciones.

¿QUÉ HACE YOLO DURANTE EL ENTRENAMIENTO?

YOLO (**You Only Look Once**) es un modelo que aprende a detectar objetos (como mazorcas de cacao con Monilia) **viendo imágenes y sabiendo dónde están los objetos dentro de ellas**.

Paso 1: Entrada

- YOLO recibe muchas imágenes 📸
- Cada imagen tiene **anotaciones**: cajas que indican **qué objeto hay** y **dónde está**
    - Ejemplo:
        - imagen_1.jpg
        - Anotación: en (x, y, ancho, alto) hay un objeto de clase "monilia"

Paso 2: División en cuadrícula

- La imagen se **divide en una cuadrícula** (como 7x7 o 20x20)
- En cada celda, YOLO intenta predecir:
    - Si hay un objeto (sí o no)
    - Qué clase es (monilia, sano…)
    - Las coordenadas del objeto (x, y, ancho, alto)

Paso 3: Cálculo de error (loss)

- YOLO compara lo que **predijo** con lo que **debería haber dicho**
- Si se equivoca:
    - Aprende ajustando sus parámetros internos
    - Repite esto con muchas imágenes hasta que mejora

Paso 4: Entrenamiento en múltiples épocas

- Una **época** = ver todo el dataset una vez

- El entrenamiento repite muchas épocas para ir mejorando poco a poco

Paso 5: Guarda el modelo

- Durante el proceso, YOLO evalúa qué tan bien va detectando
- Guarda dos archivos importantes:
  - last.pt: modelo después de la última época
  - best.pt: el que logró **mejor precisión** en validación

¿Qué aprende YOLO?

- Aprende a detectar **formas, texturas, bordes, patrones**
- Aprende **qué combinación de cosas significa "monilia" o "sano"**

Metáfora sencilla:

Es como enseñar a alguien a identificar frutas:

- Le muestras una fruta y le dices "esto es un plátano"
- Luego otra: "esto es una manzana"
- Después le das frutas nuevas, y le preguntas qué son y dónde están

RESUMEN FINAL DEL CAPÍTULO 3

Este capítulo se centró en la **arquitectura YOLOv8**, una de las versiones más avanzadas de la familia YOLO (You Only Look Once) para la detección de objetos. Se describieron las distintas **variantes de modelos preentrenados** que ofrece YOLOv8 (Nano, Small, Medium, Large, XLarge), enfatizando sus diferencias en términos de capacidad de cómputo, velocidad de inferencia y precisión, lo que permite elegir el modelo más adecuado según los recursos y necesidades del proyecto.
Además, se abordó la organización de los archivos generados durante el entrenamiento, particularmente el rol de los archivos best.pt (modelo con mayor desempeño validado) y last.pt (última iteración entrenada). Se explicó la importancia de comprender esta estructura para facilitar tanto la evaluación como la implementación del modelo en aplicaciones reales.
En la sección práctica, se expuso el proceso de **entrenamiento personalizado de YOLOv8**, desde la preparación del dataset hasta la ejecución del entrenamiento y validación. Finalmente, los estudiantes realizaron un **ejercicio aplicado en Google Colab**, donde implementaron un flujo completo de

entrenamiento, guardado de resultados y verificación de predicciones, consolidando los conocimientos adquiridos en un entorno práctico y accesible.

En conclusión, este capítulo integró los fundamentos teóricos de YOLOv8 con su aplicación práctica, brindando a los lectores herramientas para comprender, entrenar y adaptar modelos de detección de objetos en escenarios reales.

AUTOEVALUACIÓN

Marca las respuestas que puedas contestar con seguridad. Si tienes dudas, repasa el contenido correspondiente del capítulo.

1. ¿Qué diferencia hay entre clasificación de imágenes y detección de objetos?
2. ¿Qué componentes principales tiene la arquitectura YOLOv8?
3. ¿Cómo debe estructurarse un dataset para entrenar un modelo YOLOv8?
4. ¿Qué función cumple el archivo data.yaml en el entrenamiento de YOLOv8?
5. ¿Qué métricas se utilizan para evaluar el desempeño de un modelo de detección?
6. ¿Cómo se visualizan las predicciones realizadas por YOLOv8 sobre una imagen?
7. ¿Qué técnicas puedes aplicar si el modelo no detecta correctamente las clases?
8. ¿Cuál es la ventaja de usar modelos preentrenados para detección de objetos?
9. ¿Qué aplicaciones tiene la detección de objetos en el sector agrícola?
10. ¿Qué desafíos técnicos enfrentaste al entrenar YOLOv8 con tu propio dataset?

EJERCICIOS DE APLICACIÓN DE CONTENIDOS

Ejercicio 1: Crear tu propio dataset anotado

- Elige un nuevo objeto (por ejemplo, frutas, insectos, herramientas).
- Toma o reúne al menos 30 imágenes.
- Anótalas con una herramienta como LabelImg en formato YOLO.
- Estructura la carpeta en train/, valid/ y genera el archivo data.yaml.

Ejercicio 2: Entrenamiento con YOLOv8

- Entrena un modelo YOLOv8n o YOLOv8s con tu dataset personalizado.
- Ajusta los parámetros de entrenamiento (épocas, batch size).
- Evalúa el modelo y visualiza ejemplos de predicción sobre nuevas imágenes.

Ejercicio 3: Comparación de resultados

- Prueba al menos dos tamaños de modelo (por ejemplo, YOLOv8n y YOLOv8m).
- Compara el tiempo de entrenamiento, precisión y calidad de las detecciones.
- Discute cuál modelo se adapta mejor al problema y por qué.

🎯 PROYECTO AUTÓNOMO SUGERIDO

Título: *Detección de objetos aplicada a la agricultura de precisión*

1. Identifica un problema agrícola (ej. plagas, maduración de frutas, enfermedades foliares).
2. Reúne y etiqueta un conjunto mínimo de 50 imágenes para al menos 2 clases.
3. Entrena un modelo YOLOv8 con ese dataset.
4. Evalúa los resultados e identifica mejoras posibles (más datos, anotaciones precisas).
5. Documenta el proceso con capturas de resultados, métricas y lecciones aprendidas.

REFERENCIAS BIBLIOGRÁFICAS

Bochkovskiy, A., Wang, C. Y., & Liao, H. Y. M. (2020). ***YOLOv4: Optimal speed and accuracy of object detection*** [Preprint]. arXiv. https://arxiv.org/abs/2004.10934

Jocher, G., Chaurasia, A., Qiu, J., & Stoken, A. (2023). ***YOLO by Ultralytics*** [Computer software]. GitHub. https://github.com/ultralytics/ultralytics

Redmon, J., & Farhadi, A. (2018). ***YOLOv3: An incremental improvement*** [Preprint]. arXiv. https://arxiv.org/abs/1804.02767

Elshamy, M. R., Emara, H. M., Shoaib, M. R., & Badawy, A.-H. A. (2024). P-YOLOv8: Efficient and Accurate Real-Time Detection of Distracted Driving. arXiv preprint.

Jahan, M. K., Bhuiyan, F. I., Amin, A., Mridha, M. F., Safran, M., Alfarhood, S., … Che, D. (2025). Enhancing the YOLOv8 model for realtime object detection to ensure online platform safety. Scientific Reports, 15, Article 21167.

Pereira, G. A. (2024). Fall Detection for Industrial Setups Using YOLOv8 Variants. arXiv preprint.
Solawetz, J., & Francesco, (2024, October 23). What is YOLOv8? A Complete Guide. varo.ai (Blog post).

Ultralytics. (2024, March 25). Object Detection with a pre-trained Ultralytics YOLOv8 Model. Ultralytics Blog.

Vina, A. (2024, June 6). AI in oil and gas: Refining innovation. Ultralytics Blog.

Yaseen, M. (2024). What is YOLOv8: An In-Depth Exploration of the Internal Features of the Next-Generation Object Detector. arXiv preprint.

CAPÍTULO 4. RECONOCIMIENTO FACIAL CON DEEP LEARNING

RESUMEN DEL CAPÍTULO

Este capítulo explora el uso de redes neuronales profundas para el reconocimiento facial, una de las aplicaciones más representativas del deep learning moderno. Se abordan los fundamentos del reconocimiento facial, los tipos de tareas involucradas (verificación, identificación), y los modelos más utilizados como DeepFace y ArcFace. A través de un caso práctico, se construye un sistema funcional que incluye: la recolección de datos faciales personalizados, el procesamiento de imágenes con DeepFace, y el despliegue de una aplicación web con Flask que detecta rostros desde la cámara y los compara con una base de datos local.

INTRODUCCIÓN

El reconocimiento facial se ha convertido en una herramienta clave en aplicaciones de seguridad, autenticación y análisis de imágenes. Gracias a modelos basados en deep learning, como ArcFace y DeepFace, hoy es posible alcanzar niveles de precisión cercanos al humano en tareas de identificación y verificación. Este capítulo guía al lector en el desarrollo de un sistema completo de reconocimiento facial, desde la construcción del dataset de entrenamiento hasta la implementación de una app web que captura rostros en tiempo real y los reconoce utilizando embeddings faciales. Se destacan tanto los aspectos técnicos como los retos prácticos al trabajar con datos biométricos.

TRABAJOS PREVIOS Y APLICABILIDAD EN LA INDUSTRIA

Las técnicas de **reconocimiento facial basadas en deep learning** han alcanzado avances notables en precisión y robustez gracias a arquitecturas diseñadas para aprender representaciones faciales profundas. Estas tecnologías se estructuran normalmente en cuatro etapas clave: detección facial, alineación, extracción de características y comparación de rostros, siendo la representación facial el componente crucial que determina la exactitud del sistema (Deng, 2025). Revisiones recientes destacan cómo estas arquitecturas han mejorado la eficiencia y estabilidad del reconocimiento facial, especialmente en condiciones adversas como variaciones de iluminación, pose y oclusión (Amirgaliyev et al., 2025). Adicionalmente, se han

explorado enfoques innovadores como las redes cápsula (capsule networks), que modelan relaciones espaciales jerárquicas en las caras para lograr detecciones más resilientes y precisas (Elian, 2025). En el ámbito industrial, estas soluciones alimentan aplicaciones reales como autenticación biométrica en seguridad, control de acceso en instalaciones críticas, verificación en entornos bancarios y análisis de interacción en retail, consolidando su relevancia práctica y capacidad de adaptación (Amirgaliyev et al., 2025).

Preguntas de enfoque

- ¿Cómo se diferencia la verificación facial de la identificación facial?
- ¿Qué papel juegan los ***face embeddings*** en los modelos de reconocimiento?
- ¿Cómo funciona el modelo ArcFace y por qué es uno de los más precisos?
- ¿Qué ventajas ofrece DeepFace al integrar múltiples modelos y backends?
- ¿Cómo se construye una base de datos personalizada de rostros?
- ¿Qué pasos son necesarios para implementar un sistema de reconocimiento facial en una aplicación web?

Objetivos del capítulo

- Explicar los fundamentos del reconocimiento facial basado en deep learning.
- Comprender la arquitectura y funcionamiento de modelos como ArcFace y DeepFace.
- Crear un dataset personalizado de imágenes faciales.
- Implementar un sistema de reconocimiento facial usando DeepFace y Flask.
- Detectar y reconocer rostros capturados desde una cámara en tiempo real.

Resultados de aprendizaje esperados

Al finalizar este capítulo, el lector será capaz de:

- Describir el flujo completo de un sistema de reconocimiento facial con deep learning.
- Generar y utilizar embeddings faciales para comparar rostros.

- Configurar DeepFace con modelos como ArcFace para tareas de identificación.
- Integrar captura en tiempo real con Flask y OpenCV en una app web.
- Evaluar la precisión y robustez de su sistema ante distintas condiciones.

PROBLEMAS A RESOLVER

- ¿Cómo manejar correctamente la variabilidad de poses, iluminación y calidad en imágenes faciales?
- ¿Qué hacer si un rostro no es detectado correctamente o se confunde con otro?
- ¿Cómo evitar el sobreentrenamiento o sesgo en la base de datos facial?
- ¿Qué implicaciones éticas y de privacidad deben considerarse en el reconocimiento facial?
- ¿Cómo garantizar que el sistema funcione en tiempo real con precisión aceptable?

El rostro humano es una de las señales biométricas más naturales y utilizadas para identificar a una persona. Desde desbloquear un teléfono hasta verificar la identidad en un aeropuerto, el reconocimiento facial se ha convertido en una tecnología cotidiana, pero detrás de esta aparente simplicidad se encuentra un campo complejo y fascinante.

En este capítulo exploraremos cómo los modelos de deep learning han revolucionado el reconocimiento facial, superando las limitaciones de los enfoques clásicos y logrando niveles de precisión comparables al reconocimiento humano. Analizaremos conceptos clave como embeddings faciales, técnicas de verificación e identificación, y conoceremos arquitecturas populares como ArcFace, FaceNet y VGGFace. Además, implementaremos casos prácticos con DeepFace para que el lector pueda experimentar directamente con estas herramientas.

A continuación, se muestran los principales modelos de Deep Learning que puedes usar para **reconocimiento facial**, junto con su descripción, ventajas y las bibliotecas recomendadas. Todos son aplicables a proyectos reales y didácticos.

	Modelo	Descripción	Ventajas	Biblioteca / Framewor
1	FaceNet	Genera embeddings faciales a partir de imágenes. Muy útil para verificación y reconocimiento.	Preciso, permite comparar rostros con distancia euclidiana	TensorFlow / Keras, OpenFace
2	VGG-Face	Modelo basado en VGG16 entrenado en millones de rostros.	Fácil de implementar con keras-vggface, compatible con transfer learning	Keras, keras-vggface
3	Dlib Face Recognition	Utiliza redes convolucionales para generar embeddings faciales. Muy	Ligero, muy usado en aplicaciones reales	Dlib + face_recognition
4	DeepFace	Framework de alto nivel que integra varios modelos: VGG-Face, Facenet, OpenFace, Dlib,	Fácil de usar, soporte para verificación y reconocimiento	deepface
5	ArcFace (InsightFace)	Modelo moderno y robusto para clasificación y verificación facial.	Alta precisión en benchmarks, ideal para producción	PyTorch / InsightFace

Ilustración 17. modelo de deep learning para reconocimienteo facial
FUENTE: DENG, J., GUO, J., XUE, N., & ZAFEIRIOU, S. (2019)

MEJOR OPCIÓN PARA PROYECTOS REALES: **ARCFACE (INSIGHTFACE)**

¿Por qué?

- **Alta precisión**: ArcFace es uno de los modelos más precisos en benchmarks como LFW, MegaFace, etc.
- **Robustez**: funciona muy bien en condiciones variadas (luz, expresión, ángulos).
- **Producción**: usado en sistemas comerciales de seguridad, biometría, verificación de identidad, etc.
- **Escalable**: permite entrenar con millones de imágenes y comparar millones de rostros de forma eficiente.

📦 Se implementa con InsightFace, que ofrece modelos listos para usar y versiones optimizadas para GPU.

🔧 ¿Cuándo elegir otros?

Tabla 9. Modelos y cuando usarlos

Modelo	**Cuándo usarlo**
DeepFace	Prototipos rápidos, proyectos académicos o educativos.
Dlib + face_recognition	Proyectos ligeros, sin GPU, en CPU o en Raspberry Pi.
VGG-Face	Si ya trabajas en Keras y necesitas una base sencilla.
FaceNet	Si quieres entrenar tu propio modelo y generar embeddings.

FUENTE: FUENTE: DENG, J., GUO, J., XUE, N., & ZAFEIRIOU, S. (2019)

Recomendación final

- Si vas a producción (aplicaciones reales, empresas, seguridad): **ArcFace con InsightFace**.
- Si estás enseñando o haciendo una demo rápida: **DeepFace** es ideal (fácil y funcional).

MODELOS DE DEEP LEARNING VS MODELOS CLÁSICOS

LOS MODELOS MODERNOS DE DEEP LEARNING (COMO ARCFACE, FACENET, VGG-FACE) ***SUPERAN AMPLIAMENTE A LOS MÉTODOS CLÁSICOS COMO*** *LBPH, FISHERFACE Y EIGENFACES* ***EN PRECISIÓN, ROBUSTEZ Y ESCALABILIDAD.***

COMPARATIVA: DEEP LEARNING VS MÉTODOS CLÁSICOS

Tabla 10. comparativa: deep learning vs metodos clasicos

Criterio	**LBPH / Fisher / Eigenfaces**	**ArcFace / FaceNet / VGG-Face (Deep Learning)**
Precisión	Moderada (60–80% aprox.)	Alta (>98% en LFW, MegaFace, etc.)
Robustez a luz, ángulo, edad	Baja a media	Alta (entrenados con variaciones reales)

Criterio	LBPH / Fisher / Eigenfaces	ArcFace / FaceNet / VGG-Face (Deep Learning)
Generalización	Mala con nuevos rostros	Muy buena (embeddings + distancia)
Escalabilidad	Limitada	Alta (base de datos grande, búsqueda eficiente)
Uso en producción real	Casi obsoleto	Estándar actual en biometría, seguridad, fintech
Requerimiento computacional	Muy bajo (funciona sin GPU)	Medio-alto (mejor con GPU, pero se puede en CPU)

FUENTE: WANG, H., WANG, Y., ZHOU, Z., JI, X., GONG, D., ZHOU, J., ... & LIU, W. (2018)

🎯 **DIFERENCIA ENTRE MODELOS DE DEEP LEARNING Y MODELOS CLÁSICOS EN** RECONOCIMIENTO FACIAL

Tabla 11. diferencia entre modelos de deep learning y modelos clasicos en reconocimiento facial

Característica	Modelos Clásicos (LBPH, Eigenfaces, etc.)	Modelos de Deep Learning (ArcFace, FaceNet, etc.)
🔍 ¿Qué analizan?	Rasgos simples y visibles (bordes, texturas)	Representaciones abstractas aprendidas de los datos
📐 Tipo de características extraídas	Lineales y manuales (intensidad, distancias)	Profundas, no lineales, jerárquicas
🧠 ¿Aprenden de los datos?	No. Usan reglas predefinidas	Sí. Aprenden patrones complejos directamente de imágenes
🔁 Generalización	Baja. Requiere condiciones controladas	Alta. Funciona con distintos ángulos, luces, expresiones

Característica	Modelos Clásicos (LBPH, Eigenfaces, etc.)	Modelos de Deep Learning (ArcFace, FaceNet, etc.)
⚙️ Entrenamiento	No requiere (modelo ya viene definido)	Requiere millones de imágenes para entrenarse bien
🧾 Precisión	Aceptable en entornos simples	Muy alta incluso en entornos reales
💡 Ejemplos de modelos	Haar Cascade, LBPH, Eigenfaces, Fisherfaces	VGG-Face, ArcFace, FaceNet, DeepFace
📷 Resistencia a cambios en la imagen	Baja (pose, luz, fondo afectan)	Alta (pose, luz, fondo son tolerados)
💻 Requerimientos de cómputo	Bajos (CPU)	Altos (GPU recomendado)

FUENTE: WANG, H., WANG, Y., ZHOU, Z., JI, X., GONG, D., ZHOU, J., ... & LIU, W. (2018)

- Modelos clásicos:

Es como un **policía que compara fotos** mirando si los ojos, la boca y la nariz están en la misma posición.

- Necesita que las fotos sean bien iluminadas y de frente.
- Si alguien gira la cara o hay sombra, **puede fallar**.

¿Por qué los métodos clásicos ya no se usan en sistemas reales?

- Son muy **sensibles** a condiciones como iluminación, expresiones o fondo.
- No extraen **características profundas**: solo transforman la imagen linealmente.

- No escalan bien con muchos rostros (ej. más de 1000).

¿Y cuándo se usan aún?

Los métodos clásicos como **LBPH** a veces se usan:

- En dispositivos **sin GPU**, como cámaras embebidas.
- En **proyectos educativos** o introducción a visión por computadora.
- Cuando se necesita algo **ultraligero** y no se requiere precisión alta.

🔹 **Modelos de Deep Learning:**

Es como un **experto que aprendió miles de caras** y **entiende patrones profundos** como estructura ósea, forma general, texturas de piel, etc.

- Aunque cambies el fondo, el peinado o la expresión, **aún te reconoce**.

✅ ¿Cuándo usar uno u otro?

Situación	**Modelo recomendado**
Recursos limitados (Arduino, IoT)	Modelo clásico (LBPH)
Aplicación seria (seguridad, app)	Deep Learning (ArcFace, FaceNet)
Proyecto educativo simple	Modelos clásicos para comenzar
Alta precisión requerida	Modelos profundos entrenados en GPU

✅ Conclusión

Si estás trabajando en reconocimiento facial **serio o aplicado en producción**, debes usar **modelos basados en Deep Learning** (como ArcFace, FaceNet o DeepFace). Son **el nuevo estándar**.

RECOMENDACIÓN PARA TI: USAR DEEPFACE CON EL BACKEND ARCFACE

Tabla 12. usar deepface con el backend arcface
FUENTE: WANG, H., WANG, Y., ZHOU, Z., JI, X., GONG, D., ZHOU, J., ... & LIU, W. (2018)

Ventaja	Explicación
✅ **DeepFace es fácil de usar**	No necesitas configurar nada complejo. Solo instalar y usar.
✅ **Puedes elegir ArcFace como backend**	DeepFace permite cambiar el motor interno (ArcFace, FaceNet, etc.)
✅ **ArcFace es el más preciso**	Es el backend más potente dentro de DeepFace (mejor que VGG, Dlib, etc.)
✅ **Código en pocas líneas**	Ideal para prototipos, clases, pruebas y hasta producción ligera

🧠 ¿QUÉ ES ARCFACE?

ArcFace es un modelo que **aprende a reconocer rostros comparando su "esencia" matemática**, no la imagen en sí. Esa "esencia" se llama un **embedding facial**.

🧬 ¿QUÉ ES UN EMBEDDING?

Un **embedding** es como el **ADN matemático del rostro**: un vector de números que describe las características únicas de una cara.

Por ejemplo, supón que tenemos dos personas:

- 👩 Ana → embedding: [0.2, -0.4, 0.9, ...]
- 👨 Juan → embedding: [-0.1, 0.3, 0.5, ...]

Aunque no se parezcan visualmente, sus rostros están convertidos en vectores, y lo que importa ahora es qué tan **cercanos o lejanos** están **esos vectores** entre sí.

🧭 ¿Cómo decide si dos rostros son de la misma persona?

ArcFace no compara pixeles, **compara ángulos** entre embeddings, como si estuviera midiendo la separación entre dos flechas desde el mismo punto.

Ejemplo visual:

Imagina un círculo. Cada rostro está representado como una flecha (vector) desde el centro.

- ✅ Si dos vectores (embeddings) **apuntan en direcciones similares**, son de la misma persona.
- ❌ Si apuntan en direcciones muy distintas, son personas diferentes.

¿Qué hace especial a ArcFace?

ArcFace **hace más estricta la separación** entre personas distintas al agregar un **"margen angular"**.

👉 En vez de decir "quiero que Ana y Juan estén lejos", ArcFace dice:

"Quiero que estén separados **por al menos 30 grados** en el espacio de vectores."

Esto **evita que dos personas parecidas se confundan**.

📸 Ejemplo paso a paso

1. Tienes fotos de Ana y Juan. Las pasas por una red neuronal que genera estos **embeddings**:
    - Ana (foto 1): [0.1, 0.9]
    - Ana (foto 2): [0.12, 0.88]
    - Juan: [-0.8, 0.2]
2. El sistema mide el **ángulo** entre estos vectores:
    - Ana y Ana: ángulo pequeño → ✅ misma persona
    - Ana y Juan: ángulo grande → ❌ personas distintas
3. **Durante el entrenamiento**, ArcFace ajusta los pesos del modelo para que:
    - Los ángulos entre fotos del **mismo rostro** sean **más cercanos**

- Y los ángulos entre fotos de **personas distintas** sean **mayores que un margen fijo**

¿Y para qué sirve eso?

Gracias a esta estrategia, ArcFace **puede reconocer un rostro aunque cambie el peinado, la luz o la expresión**, porque **aprende lo que realmente te hace único**, no los detalles visuales superficiales.

Resumen en simple

- **ArcFace convierte rostros en vectores numéricos únicos (embeddings).**
- **Compara estos vectores midiendo el ángulo entre ellos.**
- **Aprende a separar bien los rostros usando un margen angular para más precisión.**

Ejemplo de cómo Comparar ángulos entre embeddings

Usando un ejemplo visual e intuitivo con flechas y un plano cartesiano. Imagina que estamos trabajando en un espacio 2D para que sea más fácil de entender (en realidad, ArcFace usa espacios de cientos de dimensiones).

🔵 Paso 1: Imagina un plano cartesiano

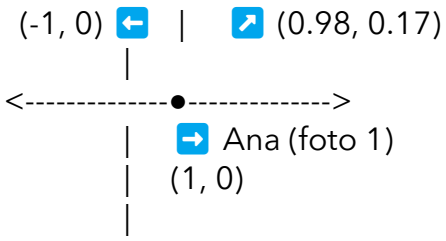
Como cuando dibujas ejes X e Y. En el centro está el punto (0, 0). Desde allí vamos a dibujar **flechas** que representan a diferentes rostros.

Supón que tienes dos personas: Ana y Juan

Cada rostro se convierte en un **embedding**, que es un vector como una flecha:

- Ana → embedding 1: (1, 0) → ➡️ una flecha apuntando a la derecha
- Ana → embedding 2: (0.98, 0.17) → ↗️ casi igual, un poco arriba
- Juan → embedding: (-1, 0) → ⬅️ una flecha apuntando a la izquierda

Dibujo imaginario:



🧠 ¿Qué mide ArcFace?

✅ **Mide el ángulo** entre las flechas. En nuestro ejemplo:

- Ángulo entre **Ana (foto 1)** y **Ana (foto 2)**: pequeño → ✅ mismo rostro
- Ángulo entre **Ana y Juan**: 180° → ❌ diferentes personas

🧲 ¿Y qué hace especial a ArcFace?

ArcFace dice:

"Quiero que **todas las fotos del mismo rostro apunten en casi la misma dirección** (ángulo pequeño), y que **las de personas distintas estén separadas por un ángulo mínimo**, por ejemplo 45° o más."

🔍 ¿Por qué no comparar distancia?

Porque **dos vectores pueden estar cerca pero no en la misma dirección**. ArcFace **normaliza todos los vectores** para que vivan en un círculo, y compara **solo el ángulo**, que es más robusto ante cambios de luz, expresión, etc.

📐 Fórmula usada internamente

ArcFace usa una fórmula que incluye:

$\cos(\theta + m)$

Donde:

- θ es el ángulo entre dos vectores.
- m es el **margen angular extra** que obliga al modelo a separar bien las clases (rostros).

Esto hace que el modelo **aprenda embeddings que no solo sean distintos, sino bien separados**.

✅ Resumen visual

Persona	Embedding (2D)	Dirección	Comparación Angular
Ana foto 1	(1, 0)	➡️	
Ana foto 2	(0.98, 0.17)	↗️	Ángulo pequeño ✅
Juan	(-1, 0)	⬅️	Ángulo grande ❌

comparar **dos embeddings faciales** usando la **distancia coseno**, que es **otra forma popular de medir similitud entre rostros**, especialmente cuando los vectores están normalizados.

🔷 ¿Qué es la distancia coseno?

La **distancia coseno** mide qué **tan alineados están dos vectores**. No le importa su magnitud, solo la dirección.

- Si dos vectores **apuntan igual** (misma dirección): distancia coseno = 0 → ✅ misma persona
- Si dos vectores **son ortogonales** (forman 90°): distancia coseno = 1 → ❌ diferentes
- Si apuntan en **direcciones opuestas**: distancia coseno = 2

🧱 Fórmula:

🧮 **Fórmula:**

$$\text{distancia\_coseno} = 1 - \frac{\vec{a} \cdot \vec{b}}{||\vec{a}|| \cdot ||\vec{b}||}$$

✅ Ejemplo numérico

Supongamos los siguientes embeddings:

- 📸 Ana (foto 1):

$$\vec{a} = [1, 0]$$

- 📸 Ana (foto 2):

$$\vec{b} = [0.98, 0.17]$$

- 📸 Juan:

$$\vec{c} = [-1, 0]$$

Paso a paso en Python

```
import numpy as np
from numpy.linalg import norm

# Vectores
a = np.array([1, 0])          # Ana foto 1
b = np.array([0.98, 0.17])    # Ana foto 2
c = np.array([-1, 0])         # Juan

# Función para calcular distancia coseno
def cosine_distance(v1, v2):
    dot = np.dot(v1, v2)
    return 1 - (dot / (norm(v1) * norm(v2)))

print("Distancia coseno Ana1 - Ana2:", cosine_distance(a, b))  # ≈ 0.015
print("Distancia coseno Ana1 - Juan:", cosine_distance(a, c))  # = 2.0
```

✅ Resultado:

- Ana1 vs Ana2: distancia coseno ≈ **0.015** → **muy parecidos**
- Ana1 vs Juan: distancia coseno = **2.0** → **completamente distintos**

🧠 ¿Cuándo se usa?

La distancia coseno se usa mucho en:

- Reconocimiento facial (FaceNet, ArcFace normaliza los embeddings, por eso se usan ángulos o distancia coseno)
- Sistemas de recomendación (similitud de textos, perfiles, etc.)

Comparación con otras métricas:

Métrica	**Qué mide**	**Rango típico**	**Bueno si...**
Euclidiana	Distancia lineal	0 a ∞	Distancia baja
Coseno	Dirección entre vectores	0 (igual) a 2 (opuesto)	Distancia baja
Ángulo (ArcFace)	Grado entre vectores	0° a 180°	Ángulo pequeño

Caso de Estudio: Reconocimiento Facial con DeepFace y ArcFace en Google Colab

🎯 Objetivo

Desarrollar un sistema de **reconocimiento facial** utilizando la biblioteca DeepFace y el modelo **ArcFace**, capaz de:

1. 📸 Capturar y almacenar rostros en una base de datos (mediante imágenes subidas).
2. 🔍 Comparar nuevos rostros contra esa base y verificar si ya existen.
3. 📊 Visualizar resultados y analizar la similitud facial usando la **distancia de embeddings**.

🛠️ Herramientas y tecnologías

- Python 3 en Google Colab
- Librería DeepFace
- Modelo de reconocimiento **ArcFace**
- OpenCV para visualización
- Matplotlib y Pandas para análisis
- Subida de imágenes a Colab manualmente

🧪 Pasos del caso de estudio

🔹 1. Captura y registro de rostros

Escenario: Se solicita al usuario subir imágenes individuales de personas para crear la base de datos de rostros.

- Cada imagen debe contener un solo rostro visible.
- Las imágenes se almacenan en la carpeta /base.

Resultado esperado: Una base de rostros lista para ser consultada.

🔹 2. Subida de una imagen nueva

Escenario: Se sube una nueva imagen con un rostro desconocido (ej. una cámara de seguridad o una selfie).

- Se compara este rostro con todos los que están en la base usando el modelo **ArcFace**.
- Se mide la **distancia de embeddings** para encontrar similitudes.

Resultado esperado: Se muestra el rostro más parecido, la distancia y se informa si es la misma persona (si la distancia está bajo cierto umbral, por ejemplo 0.6).

🔹 3. Comparación y verificación facial

Escenario adicional (opcional): El sistema puede también verificar si dos imágenes específicas son del **mismo individuo** usando DeepFace.verify().

CAPTURA Y REGISTRO DE ROSTROS

El siguiente script está diseñado para **extraer rostros desde un video** y guardarlos como imágenes, uno por uno, usando OpenCV y Haar cascades. Ideal para construir una **base de datos de rostros** para entrenamiento o prueba en modelos como ArcFace o DeepFace.

🔹 1. Importación de librerías

```
import cv2                  # Librería de visión por computador
import os as os             # Manejo de archivos y carpetas
```

```
import imutils              # Utilidades para redimensionar imágenes fácilmente
from google.colab.patches import cv2_imshow  # Mostrar imágenes en Google Colab
```

🔹 2. Definir el nombre de la persona y ruta de guardado

```
personName = 'jessica'

personPath = '/content/drive/MyDrive/practicasMachineL/DEEPLEARNING/Rfacial/data' + '/' + personName

if not os.path.exists(personPath):
    print('carpeta creada: ',personPath)
    os.makedirs(personPath)
```

- Aquí defines el nombre de la persona (jessica).
- Se crea una carpeta personalizada para ella si no existe aún.
- Se usará para guardar los rostros extraídos del video.

🔹 3. Cargar el video y el detector de rostros

```
cap = cv2.VideoCapture('/content/drive/MyDrive/practicasMachineL/DEEPLEARNING/Rfacial/test2.mp4')

faceClassif = cv2.CascadeClassifier('/content/drive/MyDrive/practicasMachineL/DEEPLEARNING/Rfacial/haarcascade_frontalface_default.xml')
```

- cv2.VideoCapture: abre el video que contiene los rostros.
- CascadeClassifier: carga el modelo Haar cascade preentrenado para detectar rostros.

🔹 4. Inicializar contador

```
count = 0
```

- Este contador se usará para numerar los archivos de rostro guardados (rostro_0.jpg, rostro_1.jpg, ...).

🔹 5. Bucle principal: procesar cada frame del video

```
while True:
```

```
  ret, frame = cap.read()           # Leer siguiente frame del video
  if ret == False: break            # Si no se pudo leer (fin del video), salimos

  frame = imutils.resize(frame, width=640)        # Redimensionar para facilitar procesamiento
  gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)     # Convertir a escala de grises (requerido por Haar)
  auxFrame = frame.copy()                         # Copia del frame original para recortar rostros
```

- 6. Detección y recorte de rostros

```
  faces = faceClassif.detectMultiScale(gray, 1.3, 5)  # Detecta rostros en escala de grises

  for (x, y, w, h) in faces:
    cv2.rectangle(frame, (x,y),(x+w,y+h),(0,255,0),2)                  # Dibuja un rectángulo
    rostro = auxFrame[y:y+h, x:x+w]                                    # Recorta el rostro
    rostro = cv2.resize(rostro, (150, 150), interpolation=cv2.INTER_CUBIC)  # Redimensiona rostro
    cv2.imwrite(personPath + '/rostro_{}.jpg'.format(count), rostro)           # Guarda la imagen
    count = count + 1                                                  # Aumenta contador
```

- 7. Mostrar el frame actual

```
  cv2_imshow(frame)
```

  - Muestra el frame actual con rectángulos en Colab.

- 8. Control de salida

```
  k = cv2.waitKey(1)
  if k == 27 or count >= 300:
    break
```

  - waitKey(1) permite avanzar cuadro a cuadro.
  - Si se presiona Esc (27) o se capturan 300 rostros, termina el bucle.

- 9. Liberar recursos

```
cap.release()
cv2.destroyAllWindows()
```

- Libera la cámara o video y cierra todas las ventanas de OpenCV.

🧠 ¿Qué hace en resumen?

1. Abre un video que contiene una o más personas.
2. Detecta rostros en cada frame.
3. Recorta y guarda los rostros detectados como imágenes de 150x150.
4. Los guarda en una carpeta con el nombre de la persona.
5. Se detiene al llegar a 300 imágenes.

Una vez que tenemos una data con la estructura de carpetas y archivos realizamos el proyecto completo de reconocimiento facial en Google Colab, utilizando DeepFace con el modelo **ArcFace**. Vamos paso a paso para entenderlo de forma clara.

🔁 Resumen general del flujo

1. Se carga una base de datos de rostros desde Drive.
2. Se genera el **embedding** de cada rostro usando el modelo ArcFace.
3. Se sube una imagen de prueba.
4. Se detectan los rostros en esa imagen.
5. Se genera su embedding y se compara con los de la base usando **distancia coseno**.
6. Si la distancia es menor al umbral, se identifica; si no, se marca como **Desconocido**.

🔍 Paso a paso

📁 Cambiar directorio a tu carpeta en Drive

%cd /content/drive/MyDrive/practicasMachineL/DEEPLEARNING/Rfacial

- %cd: cambia el directorio de trabajo en Colab.
- Te ubicas en la carpeta donde está tu base de datos y archivos relacionados.

⚙️ Instalar deepface

!pip install deepface

- Instala la librería DeepFace, que incluye modelos preentrenados para reconocimiento facial como ArcFace, VGG-Face, etc.

📚 Importación de librerías necesarias

```
from deepface import DeepFace
from google.colab import files
import zipfile, os
import cv2
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics.pairwise import cosine_distances
```

- DeepFace: para análisis y embeddings.
- files: para subir imágenes desde tu computadora.
- cv2: para manipular imágenes.
- cosine_distances: calcula qué tan similares son dos vectores.

📂 Definir ruta de base de datos

```
db_path = "/content/drive/MyDrive/practicasMachineL/DEEPLEARNING/Rfacial/data"
```

- Es la carpeta que contiene **subcarpetas por persona**, y dentro de cada subcarpeta hay imágenes.

🔄 Cargar y representar la base de datos

```
db_embeddings = []
db_names = []

for person in os.listdir(db_path):
    person_folder = os.path.join(db_path, person)
    if os.path.isdir(person_folder):
        for img in os.listdir(person_folder):
            full_path = os.path.join(person_folder, img)
            try:
                emb = DeepFace.represent(img_path=full_path, model_name="ArcFace", enforce_detection=False)[0]['embedding']
                db_embeddings.append(emb)
```

```
        db_names.append(person)
    except:
        continue
```

- Recorre cada imagen en tu base.
- Extrae el **embedding** usando ArcFace.
- Lo guarda en una lista db_embeddings junto con el nombre de la persona.

Este es el paso que **convierte cada cara en una huella digital numérica** para comparar luego.

📤 Subir imagen nueva

```
uploaded = files.upload()
img_path = list(uploaded.keys())[0]
```

- Se sube una imagen para comparar.
- img_path tendrá el nombre del archivo subido.

🧠 Detección de rostros y extracción de áreas

```
img = cv2.imread(img_path)
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
face_info = DeepFace.extract_faces(img_path=img_path, detector_backend="retinaface", enforce_detection=False)
```

- Carga la imagen y la convierte a RGB para visualizar con matplotlib.
- Usa el detector de rostros retinaface para localizar rostros en la imagen.

🔁 Para cada rostro detectado:

```
threshold = 0.68  # Umbral para ArcFace (cuanto menor, más parecido)

for face in face_info:
    box = face['facial_area']
    x, y, w, h = box['x'], box['y'], box['w'], box['h']
    roi = img_rgb[y:y+h, x:x+w]
```

- Extrae la región del rostro (ROI) usando coordenadas del detector.

📐 Generar embedding del rostro de prueba y comparar con base

```
test_emb = DeepFace.represent(img_path=roi, model_name="ArcFace", enforce_detection=False)[0]['embedding']
```

```
distances = cosine_distances([test_emb], db_embeddings)[0]
min_dist = np.min(distances)
best_idx = np.argmin(distances)
identity = db_names[best_idx] if min_dist < threshold else "Desconocido"
```

- Obtiene el **embedding del rostro detectado**.
- Lo compara con todos los de la base usando **distancia coseno**.
- Si la distancia mínima es menor al umbral (0.68), lo reconoce.

🖼️ Dibujar resultados sobre la imagen

```
color = (0, 255, 0) if identity != "Desconocido" else (255, 0, 0)
cv2.rectangle(img_rgb, (x, y), (x+w, y+h), color, 2)
cv2.putText(img_rgb, identity, (x, y-10), cv2.FONT_HERSHEY_SIMPLEX, 0.9, color, 2)
```

- Dibuja un rectángulo y el nombre de la persona.
- Verde si se reconoce, rojo si no.

🖼️ Mostrar resultado

```
plt.imshow(img_rgb)
plt.axis("off")
plt.title("Resultado del reconocimiento")
plt.show()
```

- Muestra la imagen final con las predicciones encima.

✅ Resultado final

- Si el rostro aparece en la base, te dirá el nombre.
- Si no, lo marcará como **"Desconocido"**.
- Puedes repetir el proceso con más imágenes.

Ilustración 18. prediccion con el modelo deepface y arcface
FUENTE: ELABORACION PROPIA

🎓 Caso de Estudio: Sistema Web de Reconocimiento Facial en Tiempo Real con Flask, DeepFace y FaceNet

🧠 Objetivo

Desarrollar una aplicación web que:

1. 📸 **Capture rostros desde una cámara en tiempo real**
2. 🧠 Genere **embeddings faciales con FaceNet**
3. 🗃️ Compare el rostro capturado con una **base de datos local de usuarios**
4. ✅ Identifique a la persona o indique si es **desconocida**

🛠️ Tecnologías y herramientas

- Python 3.x
- Flask (microframework web)
- DeepFace (FaceNet como backend de embeddings)
- OpenCV (captura y procesamiento de video)
- HTML/CSS (interfaz simple)

📁 Estructura de archivos sugerida

```
proyecto_reconocimiento/
├── main.py
├── base_rostros/
│   ├── Wilmer/
│   │   └── rostro_001.jpg
│   ├── Jessica/
│   │   └── rostro_002.jpg
├── templates/
│   └── index.html
├── static/
│   └── script.js
```

🔁 Flujo del sistema

1. El usuario accede a la web.
2. La cámara se activa para capturar un rostro.
3. El rostro se extrae y se guarda temporalmente.
4. Se genera un embedding con FaceNet.
5. Se compara con todos los embeddings de la base.
6. Se muestra el nombre de la persona más parecida (si la distancia es menor a un umbral) o "Desconocido".

🧪 Casos de uso

- 🔐 Control de acceso por rostro en oficinas.
- 🧾 Registro automático de asistencia.
- 🎓 Identificación de estudiantes en evaluaciones virtuales.

🔍 Consideraciones técnicas

- Umbral recomendado para FaceNet: **0.6** (distancia euclidiana)
- Es importante normalizar las imágenes (alineación + 160x160 px)
- Para mayor rendimiento, los embeddings de la base se **precargan** en memoria

📌 Requisitos de aprendizaje

- Comprender cómo DeepFace abstrae el uso de modelos complejos como FaceNet
- Integrar visión por computador con una interfaz web
- Aplicar medidas de distancia (euclidiana o coseno) para reconocimiento
- Procesar y visualizar video en tiempo real con OpenCV en Flask

✅ Requisitos y pasos para ejecutar una app Flask con DeepFace y FaceNet

🔧 1. Instalar Python 3.10

DeepFace requiere una versión compatible de Python (preferiblemente 3.9 o 3.10).
🔽 Descárgalo desde: https://www.python.org/downloads/release/python-3100/

Durante la instalación, asegúrate de marcar la opción **"Add Python to PATH"**.

🧪 2. Crear un entorno virtual con Python 3.10

Abre una terminal (CMD o PowerShell) y ejecuta:

```
# Verifica que Python 3.10 esté disponible
py -3.10 --version

# Crea un entorno virtual con Python 3.10
py -3.10 -m venv venv

# Activa el entorno virtual
# En Windows:
venv\Scripts\activate
```

📦 3. Instalar Flask y DeepFace

Una vez activado el entorno virtual, instala las dependencias necesarias:

```
pip install flask deepface==0.0.79
```

⚠️ Usa deepface==0.0.79 o similar que sea compatible con Python 3.10. También puedes fijar las versiones en un requirements.txt.

▶️ 4. Ejecutar la aplicación

En la terminal con el entorno activo:

```
python main.py
```

Abre en el navegador: http://127.0.0.1:5000

Solución.

1. Archivo main.py Script crea una aplicación **Flask** para **reconocimiento facial en tiempo real** con el modelo **FaceNet** de DeepFace, capturando

imágenes desde la cámara web del usuario mediante la interfaz web y comparándolas con una base local de rostros.

🧱 Estructura General del Código

1. Se define una app con Flask.
2. Se precargan los **embeddings** de una base de datos de rostros (/data/persona/*.jpg).
3. Se expone una ruta /reconocer que recibe una imagen desde el navegador, detecta rostros y los compara.
4. Devuelve como respuesta el nombre de la persona o "Desconocido".

🔍 EXPLICACIÓN POR PARTES

📦 1. Importaciones y configuración inicial

```
from flask import Flask, render_template, request, jsonify
import cv2, numpy as np, os, base64
from deepface import DeepFace
from sklearn.metrics.pairwise import cosine_distances
from PIL import Image
```

- Se importan todas las librerías necesarias para la app web, tratamiento de imágenes, embeddings y comparación de vectores.

🚀 2. Inicialización de la app y variables globales

```
app = Flask(__name__)
db_path = "data"  # carpeta base de rostros
db_embeddings = []  # vectores faciales
db_names = []       # nombres asociados
threshold = 0.6     # umbral de similitud (para FaceNet recomendado)
```

- Aquí se preparan las listas y el umbral que se usará para verificar si el rostro detectado es **suficientemente similar** a alguno en la base.

🧠 3. Función cargar_base(): precarga los embeddings de la base

```
def cargar_base():
    for person_name in os.listdir(db_path):
        ...
        for img_name in os.listdir(person_folder):
            ...
```

```
    img = cv2.imread(img_path)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    img = cv2.resize(img, (224, 224))

    embedding = DeepFace.represent(
      img_path=img,
      model_name="Facenet",
      detector_backend="opencv",
      enforce_detection=False
    )[0]['embedding']

    db_embeddings.append(embedding)
    db_names.append(person_name)
```

- Carga todas las imágenes de la carpeta data.
- Convierte cada rostro en un **embedding con FaceNet (**Schroff, et al 2015).
- Los guarda en db_embeddings junto al nombre de la persona.

Esto es clave: convierte las imágenes de referencia en vectores matemáticos.

🌐 4. Ruta / – Página principal

```
@app.route('/')
def index():
  return render_template('index.html')
```

- Carga una interfaz HTML que probablemente incluye la cámara web y un botón para capturar.

📸 5. Ruta /reconocer – Proceso de reconocimiento facial

```
@app.route('/reconocer', methods=['POST'])
def reconocer():
  cargar_base()  # (puedes moverlo a una inicialización única)

  # 1. Recibir imagen (en base64 desde JS)
  data = request.json['image'].split(',')[1]
  img_bytes = base64.b64decode(data)
  np_arr = np.frombuffer(img_bytes, np.uint8)
  img = cv2.imdecode(np_arr, cv2.IMREAD_COLOR)
```

- Recibe la imagen capturada desde la cámara web del usuario.
- Decodifica el string base64 (usado por JavaScript/HTML) en una imagen OpenCV.

🔍 6. Detección y comparación de rostros

```
detections = DeepFace.extract_faces(img_path=img, detector_backend="opencv", enforce_detection=False)
```

- Usa el detector para localizar los rostros dentro de la imagen enviada.

```
for face in detections:
    x, y, w, h = face['facial_area'].values()
    roi = img[y:y+h, x:x+w]
    emb = DeepFace.represent(img_path=roi, model_name="Facenet", detector_backend="opencv", enforce_detection=False)[0]['embedding']
```

- Extrae cada rostro (ROI) y obtiene su embedding.

🧪 7. Comparación del embedding con los almacenados

```
distances = cosine_distances([emb], db_embeddings)[0]
min_dist = np.min(distances)
best_idx = np.argmin(distances)
identity = db_names[best_idx] if min_dist < threshold else "Desconocido"
```

- Compara el embedding con todos los de la base usando **distancia coseno**.
- Si la distancia es menor al umbral, lo reconoce. Si no, lo marca como "Desconocido".

📩 8. Devolver resultados al navegador

```
return jsonify({"resultados": resultados})
```

- Devuelve un JSON con los nombres identificados al cliente (usualmente el navegador).

🚀 9. Ejecutar la aplicación Flask

```
if __name__ == '__main__':
    app.run(debug=True, port=5050)
```

- Ejecuta el servidor web en localhost:5050.

2. Archivo static/script.js código en JavaScript es el que se encarga de:

1. **Activar la cámara del usuario**
2. **Capturar una imagen cuando se presiona un botón**
3. **Enviar esa imagen al backend Flask** para que el sistema haga el reconocimiento facial con DeepFace
4. **Mostrar el resultado en pantalla**

Te lo explico paso a paso:

🔧 1. Acceso a elementos del DOM

```
const video = document.getElementById('video');
const canvas = document.getElementById('canvas');
const resultado = document.getElementById('resultado');
const context = canvas.getContext('2d');
```

- Obtiene referencias a los elementos HTML:
  - <video> donde se muestra la cámara
  - <canvas> donde se dibuja el frame capturado
  - (o <p>) donde se mostrará el resultado
  - context permite dibujar sobre el canvas (como una hoja en blanco)

🎥 2. Activar la cámara del usuario

```
navigator.mediaDevices.getUserMedia({ video: true }).then(stream => {
  video.srcObject = stream;
});
```

- Solicita acceso a la cámara.
- Si el usuario acepta, el video de la cámara se muestra en el <video> en tiempo real.

📸 3. Función capturar() – Captura un frame y lo envía al servidor

```
function capturar() {
  context.drawImage(video, 0, 0, canvas.width, canvas.height);
```

- Toma una **foto** de lo que se ve en ese momento en el video y la dibuja en el canvas.

🖼️ 4. Convertir la imagen a base64

```
  let dataUrl = canvas.toDataURL('image/jpeg');
  document.getElementById("img_preview").src = dataUrl;
```

- Convierte el contenido del canvas en una **imagen JPEG en base64**.
- Muestra esa imagen en un <img id="img_preview"> para que el usuario vea la captura.

📤 5. Enviar la imagen al backend Flask

```
fetch('/reconocer', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ image: dataUrl })
})
```

- Hace una petición POST a la ruta /reconocer del backend Flask.
- Envía el string base64 dentro del cuerpo JSON ({ image: dataUrl }).

📬 6. Recibir la respuesta del servidor y mostrar el nombre

```
.then(res => {
  return res.json();  // Convierte la respuesta en un objeto JS
})
.then(data => {
  document.getElementById("resultado").innerText = data.resultados.join(', ');
})
```

- El servidor responde con un JSON del tipo: { "resultados": ["Jessica"] }
- El nombre se muestra en pantalla usando innerText.

⚠️ 7. Manejo de errores

```
.catch(error => {
  alert("❌ Error en res.json() o fetch:" + error);
});
```

- Muestra una alerta si ocurre un error al convertir la respuesta o al hacer la solicitud.

3. Archivo /templates/index.html archivo HTML básico que forma parte del **frontend** de tu sistema de reconocimiento facial. Está diseñado para **mostrar la cámara, capturar una imagen del rostro del usuario y enviar esa imagen al backend (Flask)** para reconocimiento. Vamos paso a paso:

🧱 Estructura general

```
<!DOCTYPE html>
<html>
<head>
  <title>Reconocimiento Facial</title>
</head>
<body>
  ...
</body>
</html>
```

- Es un documento HTML estándar.
- El <title> define el título que aparece en la pestaña del navegador.

📸 Interfaz de captura de rostro

```
<h2>📸 Captura tu rostro</h2>
<video id="video" width="320" height="240" autoplay></video>
```

- Muestra un **elemento de video** que se conectará a la cámara del usuario.
- El atributo autoplay hace que el video se reproduzca automáticamente cuando se recibe el stream.

⭕ Botón para tomar una foto

```
<br>
<button onclick="capturar()">Reconocer</button>
```

- Este botón ejecuta la función capturar() (definida en un archivo JS externo).
- Al hacer clic, se toma una **foto del video actual** y se envía al servidor.

🎨 Canvas oculto para tomar la imagen

```
<canvas id="canvas" width="320" height="240" style="display:none;"></canvas>
```

- El <canvas> es una zona gráfica oculta.
- Se usa para "dibujar" el frame capturado del video, y luego convertirlo en una imagen.

🖼️ Mostrar imagen capturada

- Aquí se muestra una **vista previa** de la imagen que el usuario acaba de capturar con la cámara.

🧾 Resultado del reconocimiento

Resultado: ...

- Aquí se mostrará el resultado que devuelve Flask (por ejemplo, "Jessica" o "Desconocido").

🔌 Carga del script JavaScript

<script src="/static/script.js"></script>

- Se importa el archivo script.js ubicado en la carpeta /static del proyecto Flask.
- Este archivo contiene toda la lógica para:
  - Activar la cámara
  - Capturar la imagen
  - Convertirla a base64
  - Enviarla por fetch() al servidor
  - Mostrar el resultado

✅ ¿Cómo se conecta con Flask?

Cuando haces clic en el botón, el script llama a /reconocer en el backend Flask (como te mostré en el código anterior), y Flask responde con el nombre reconocido.

RESUMEN FINAL DEL CAPÍTULO 4

En este capítulo se abordaron los modelos de **Deep Learning aplicados al reconocimiento facial**, destacando sus ventajas frente a modelos simples tradicionales. Se explicó cómo las arquitecturas profundas permiten una representación más robusta y precisa de los rasgos faciales, incrementando la capacidad de identificación incluso en condiciones complejas como variaciones de iluminación, expresiones o ángulos de captura.

Asimismo, se realizó una comparación entre modelos básicos y los basados en redes neuronales profundas, evidenciando la superioridad de estos últimos en términos de exactitud y escalabilidad. Finalmente, se incluyó una **sección práctica** que permitió a los lectores implementar un ejercicio de reconocimiento facial, consolidando así los conceptos teóricos y mostrando su aplicación en un entorno real.

AUTOEVALUACIÓN

Reflexiona sobre estas preguntas. Si puedes responderlas con claridad, has comprendido los conceptos clave del capítulo.

1. ¿Qué diferencia hay entre detección facial y reconocimiento facial?
2. ¿Qué es un ***embedding facial*** y qué representa?
3. ¿Qué ventajas ofrece el modelo ArcFace frente a otros enfoques tradicionales?
4. ¿Qué funcionalidades ofrece la librería DeepFace?

5. ¿Cómo se construye una base de datos personalizada de rostros para un sistema de reconocimiento?
6. ¿Qué técnicas pueden mejorar la precisión del reconocimiento facial?
7. ¿Cómo se integran OpenCV y Flask para capturar y procesar imágenes desde la cámara?
8. ¿Cuál es el flujo general de comparación entre una imagen capturada y una base de datos de rostros?
9. ¿Qué implicaciones éticas existen en el uso de reconocimiento facial?
10. ¿Qué errores comunes pueden presentarse al implementar un sistema como este?

EJERCICIOS DE APLICACIÓN DE CONTENIDOS

Ejercicio 1: Captura de rostros personalizada

- Diseña un pequeño sistema que permita registrar los rostros de varias personas usando la cámara de tu computador.
- Almacena las imágenes en carpetas con el nombre de cada persona.

Ejercicio 2: Reconocimiento facial en tiempo real

- Crea un programa en Flask que capture una imagen desde la cámara.
- Usa DeepFace y ArcFace para comparar esa imagen con tu base de rostros.
- Muestra el nombre de la persona reconocida o "Desconocido" si no hay coincidencia.

Ejercicio 3: Comparación entre modelos

- Modifica tu código para comparar el rendimiento de diferentes modelos disponibles en DeepFace (VGG-Face, Facenet, ArcFace).
- Evalúa cuál funciona mejor en tu dataset considerando precisión y tiempo de respuesta.

Ejercicio 4: Evaluación de precisión

- Crea un conjunto de pruebas con imágenes conocidas y desconocidas.
- Mide cuántas veces el sistema reconoce correctamente y cuántas se equivoca.
- Presenta los resultados en una matriz de confusión.

PROYECTO AUTÓNOMO SUGERIDO

Título: *Sistema de control de acceso por reconocimiento facial*

1. Diseña una aplicación web con Flask que permita reconocer rostros en tiempo real desde una cámara.
2. Si el rostro coincide con un registro autorizado, muestra "Acceso permitido". Si no, "Acceso denegado".
3. Usa ArcFace como modelo de embeddings, y guarda los resultados en un registro con fecha y hora.
4. Implementa una interfaz visual sencilla (HTML + JS) para mostrar la cámara, el estado del acceso y el rostro capturado.
5. Documenta tu implementación con capturas y una reflexión sobre las limitaciones del sistema.

REFERENCIAS BIBLIOGRÁFICAS

Deng, J., Guo, J., Xue, N., & Zafeiriou, S. (2019). ArcFace: Additive angular margin loss for deep face recognition. In ***Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*** (pp. 4690–4699). https://doi.org/10.1109/CVPR.2019.00482

Schroff, F., Kalenichenko, D., & Philbin, J. (2015). FaceNet: A unified embedding for face recognition and clustering. In ***Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*** (pp. 815–823). https://doi.org/10.1109/CVPR.2015.7298682

Wang, H., Wang, Y., Zhou, Z., Ji, X., Gong, D., Zhou, J., ... & Liu, W. (2018). CosFace: Large margin cosine loss for deep face recognition. In ***Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*** (pp. 5265–5274). https://doi.org/10.1109/CVPR.2018.00552

Amirgaliyev, B., Mussabek, M., & Rakhimzhanova, T. (2025). A Review of Machine Learning and Deep Learning Methods for Person Detection, Tracking and Identification, and Face Recognition with Applications. ***Sensors, 25***(5), 1410. https://doi.org/10.3390/s25051410 Biz4Group+15MDPI+15ResearchGate+15

Deng, C. (2025). A review of face recognition technologies based on deep learning. ***Proceedings of the 4th International Conference on Signal Processing and Machine Learning***, 1–10. ResearchGate

Elian, H. M. (2025). Facial recognition technique leveraging capsule networks for efficient recognition. ***Egyptian Journal of Remote Sensing and Space Sciences***, 93.5% accuracy, capsule networks (CapsNets) hierarchically model facial features.

Wilmer Braulio Rivas Asanza https://orcid.org/0000-0002-2239-3664
Es Ingeniero en Informática por la Universidad Católica de Cuenca, con una formación académica que incluye un Diplomado en Auditoría de TI por la Escuela Superior Politécnica del Litoral (ESPOL), un Diplomado en Inteligencia Artificial, una Maestría en Docencia y Gestión en la Educación Superior por la Universidad Estatal de Guayaquil y una Maestría en Gestión Estratégica de las Tecnologías de la Información por la Universidad Estatal de Cuenca. Además, es Doctor en Tecnologías de la Información y las Comunicaciones por la Universidad de La Coruña, España. Cuenta con experiencia profesional tanto en el sector público como en el privado. Es profesor titular en la Universidad Técnica de Machala durante aproximadamente 14 años, impartiendo asignaturas como Sistemas Operativos, Gestión de Centros de Cómputo e Inteligencia Artificial. Actualmente, cuenta con varias publicaciones y libros indexados que reflejan su trayectoria académica y profesional.

Bertha Eugenia Mazón Olivo https://orcid.org/0000-0002-2749-8561
Es docente investigadora titular en la UniversidadTécnica de Machala (UTMACH), Ecuador. Doctora en Tecnologías de la Información y las Comunicaciones por la Universidad de A Coruña, España. Es Magíster en Informática Aplicada e Ingeniera en Sistemas por la Escuela Superior Politécnica de Chimborazo (ESPOCH) y posee un Diplomado Internacional en Ingeniería de Datos por la Universidad Nacional de Ucayali, Perú. Actualmente cursa una Maestría en Big Data y Data Science en la Universidad Internacional de Valencia. Participa como docente de posgrado en UTMACH, Universidad de los Hemisferios y ESPOCH. Forma parte del grupo de investigación AutoMathTIC y dirige el proyecto sobre adopción de IA e IOT en el sector agropecuario de El Oro. Contribuyó al diseño de la carrera de Ciencia de Datos en UTMACH. Sus intereses incluyen IOT, Ciencia de Datos, Big Data, Machine Learning y Deep Learning. Ha publicado diversos artículos científicos, libros y capítulos.

Joffre Jeorwin Cartuche Calva https://orcid.org/0000-0002-1633-2291
Ecuatoriano, Analista e Ingeniero en Sistemas Informáticos por la ESPOCH, con una Maestría en Ingeniería de Software por la ESPE. Candidato a Doctor en Dirección de Proyectos por la Universidad de Investigación e Innovación de México. Docente titular agregado en la Universidad Técnica de Machala, donde imparte asignaturas de pregrado como Programación Orientada a Objetos, Programación Avanzada, Programación Móvil con Despliegues en la Nube, y en posgrado Gestión de Proyectos de Software. Cuenta con experiencia en desarrollo de software, soluciones móviles e integradas, y gestión tecnológica. Sus intereses incluyen arquitectura de software, metodologías ágiles, inteligencia artificial aplicada, microservicios, contenedores, computación en la nube (AWS y Azure) y despliegue de datos. Su enfoque está orientado a la innovación tecnológica, el desarrollo de software escalable y la gestión eficiente de proyectos de TI.

ISBN: 978-9942-53-028-8
9 789942 530288

Compás
capacitación e investigación