

# UNIDAD 6

## VISIÓN ARTIFICIAL

### Programación de Inteligencia Artificial

Curso de Especialización en Inteligencia Artificial y Big Data



CHATGPT prompt: A futuristic digital illustration of computer vision technology: a close-up of a detailed human eye with vibrant blue and orange iris, surrounded by digital holograms and interface elements. A humanoid robot interacts with a glowing 3D data interface, while scientists work at computer stations in a high-tech lab. The scene includes elements like neural networks, data flow diagrams, and surveillance cameras. Blue-toned lighting, sleek and modern atmosphere, educational and visually striking.

**Carlos M. Abrisqueta Valcárcel**  
IES Ingeniero de la Cierva 2024/25

## ÍNDICE

<b>1. INTRODUCCIÓN .....</b>	<b>4</b>
<b>2. TECNOLOGÍAS DE VISIÓN ARTIFICIAL .....</b>	<b>5</b>
2.1. OPENCV: LA BASE DE LA VISIÓN ARTIFICIAL CLÁSICA.....	5
2.1.1. <i>Principales Funcionalidades de OpenCV</i> .....	6
2.1.2. <i>Ventajas y aplicaciones de OpenCV</i> .....	7
2.1.3. <i>Desafíos y futuro de OpenCV</i> .....	8
2.2. YOLO: DETECCIÓN DE OBJETOS EN TIEMPO REAL .....	8
2.2.1. <i>Ventajas de YOLO</i> .....	9
2.2.2. <i>Evolución de YOLO</i> .....	9
2.2.3. <i>Aplicaciones de YOLO</i> .....	10
2.2.4. <i>Ejemplo de uso de YOLO</i> .....	10
2.2.5. <i>Limitaciones de YOLO</i> .....	11
2.3. MEDIPIPE: ANÁLISIS DE CUERPO, MANOS Y CARA.....	11
2.4. SEGMENTACIÓN CON SAM DE META .....	12
2.5. OTRAS TECNOLOGÍAS .....	12
<b>3. TRATAMIENTO DE IMÁGENES CON OPENCV .....</b>	<b>12</b>
3.1. CARGAR IMÁGENES .....	13
3.2. ROTACIÓN DE IMÁGENES USANDO NUMPY .....	13
3.3. AJUSTE DE BRILLO UTILIZANDO NUMPY .....	14
3.4. TRASLACIÓN DE IMÁGENES .....	15
3.5. FLIPS .....	15
3.5.1. <i>Redimensionamiento de imágenes</i> .....	17
3.6. OTRAS LIBRERÍAS – LA LIBRERÍA PILLOW .....	18
3.7. CAPTURA DE VÍDEO DESDE LA CÁMARA CON OPENCV .....	18
3.8. LOS FPS .....	20
3.9. CAPTURA DE CÁMARA Y PROCESAMIENTO DE FRAMES EN OPENCV .....	21
<b>4. LAS REDES NEURONALES CONVOLUCIONALES.....</b>	<b>23</b>
4.1. LAS CONVOLUCIONES .....	23
4.1.1. <i>Propiedades de la Convolución</i> .....	24
4.1.2. <i>Aplicaciones de la Convolución</i> .....	24
4.1.3. <i>Ejemplo de Convolución Discreta</i> .....	24
4.2. ESTRUCTURA DE UNA CNN .....	25
4.3. ENTRENAMIENTO DE UNA CNN.....	26
4.3.1. <i>Kernels de una CNN</i> .....	26
4.4. CABECERA DE CLASIFICACIÓN EN UNA CNN.....	28
4.4.1. <i>Ventajas de la Cabecera de Clasificación</i> .....	29
4.4.2. <i>Aplicaciones</i> .....	29
<b>5. APRENDIZAJE POR TRANSFERENCIA.....</b>	<b>30</b>
5.1. TIPOS DE APRENDIZAJE POR TRANSFERENCIA .....	31
5.1.1. <i>Extracción de Características</i> .....	31
5.1.2. <i>Ajuste Fino (fine-tuning)</i> .....	32
<b>6. VISIÓN ARTIFICIAL AVANZADA .....</b>	<b>32</b>
6.1. INSTALACIÓN DE YOLO Y MEDIPIPE.....	32
6.2. USO DE YOLO PARA DETECCIÓN DE OBJETOS .....	32
6.3. CLASES DE OBJETOS EN YOLO .....	34
6.4. FILTRADO DE CLASES EN YOLOv8 .....	35
6.5. USO DE MEDIPIPE PARA DETECCIÓN DE POSES Y GESTOS .....	37
6.6. MEDIPIPE MODEL MAKER.....	38
6.6.1. <i>¿Qué es MediaPipe Model Maker?</i> .....	38
6.6.2. <i>Casos de Uso</i> .....	38

6.6.3.	<i>Características Principales</i> .....	39
6.6.4.	<i>Flujo de Trabajo</i> .....	39
6.6.5.	<i>Instalación</i> .....	39
6.6.6.	<i>Ejemplo en Python: Clasificación de Imágenes</i> .....	39
6.7.	TENSORFLOW LITE (TFLITE).....	40

## 1. INTRODUCCIÓN

Imagina que tienes una cámara que puede ver todo a su alrededor. Ahora piensa: ¿qué pasaría si pudiéramos enseñarle a esa cámara no solo a capturar imágenes, sino también a entenderlas? Eso es exactamente lo que busca la visión artificial, una rama de la inteligencia artificial que se encarga de enseñar a las máquinas a ver como lo hacemos los humanos.

Cuando miramos una imagen, nuestro cerebro automáticamente reconoce lo que hay en ella: un gato, un árbol, una señal de tráfico o incluso si alguien está feliz o triste. La visión artificial intenta replicar esta habilidad en las computadoras, usando programas y algoritmos avanzados para interpretar fotos, videos y datos visuales en general.

Pero ¿por qué es tan importante que las máquinas aprendan a ver? Bueno, esta tecnología se utiliza en muchas cosas que usamos o vemos todos los días, como:

- Los sistemas que desbloquean un teléfono con tu cara (reconocimiento facial).
- Los coches autónomos que necesitan detectar peatones, otros vehículos y señales de tráfico.
- Las aplicaciones de seguridad que identifican objetos sospechosos en cámaras de vigilancia.
- Incluso las herramientas médicas que analizan radiografías o tomografías para encontrar enfermedades.

**¿Qué tiene que ver la percepción computacional?** La percepción computacional es como una versión más grande de la visión artificial. No solo se trata de enseñar a las máquinas a ver, sino también a escuchar, sentir y entender todo tipo de datos que obtienen de su entorno. Por ejemplo, un robot podría usar visión artificial para reconocer un objeto en una mesa, pero también percepción computacional para combinar esa información con datos de un sensor táctil y saber si puede agarrar ese objeto sin dañarlo.

Podemos decir que la visión artificial es como un súper poder dentro de la percepción computacional, pero enfocado en todo lo visual.

**¿Cómo lo logran los ordenadores?** Gracias a las redes neuronales profundas, que son un tipo especial de programas de inteligencia artificial. Estas redes están diseñadas para aprender de manera parecida a como lo hace el cerebro humano. Por ejemplo, una red neuronal convolucional (o CNN, por sus siglas en inglés) puede analizar imágenes y descubrir patrones, como bordes, formas y colores, para después combinarlos y reconocer cosas más complejas, como un perro o un coche.

Por suerte, herramientas como *TensorFlow* y *PyTorch* hacen que trabajar con estas redes neuronales sea mucho más fácil. Con ellas, podemos entrenar a una computadora para que realice tareas increíbles, como:

- Detectar defectos en productos en una línea de ensamblaje.
- Clasificar fotos en álbumes automáticamente.
- Reconocer gestos de las manos para controlar un videojuego.

Aunque la visión artificial ha avanzado mucho, todavía se enfrenta a algunos desafíos. Por ejemplo, las computadoras a veces tienen problemas para reconocer cosas si cambian las condiciones de luz, el ángulo de la cámara o si hay mucho ruido en la imagen. Además, es importante que los sistemas sean rápidos, especialmente en aplicaciones donde el tiempo es clave, como los coches autónomos.

A pesar de esto, la visión artificial es un campo lleno de posibilidades. Cada vez más, está transformando la forma en que interactuamos con la tecnología y abriendo las puertas a un futuro donde las máquinas no solo vean, sino que también entiendan el mundo que nos rodea.

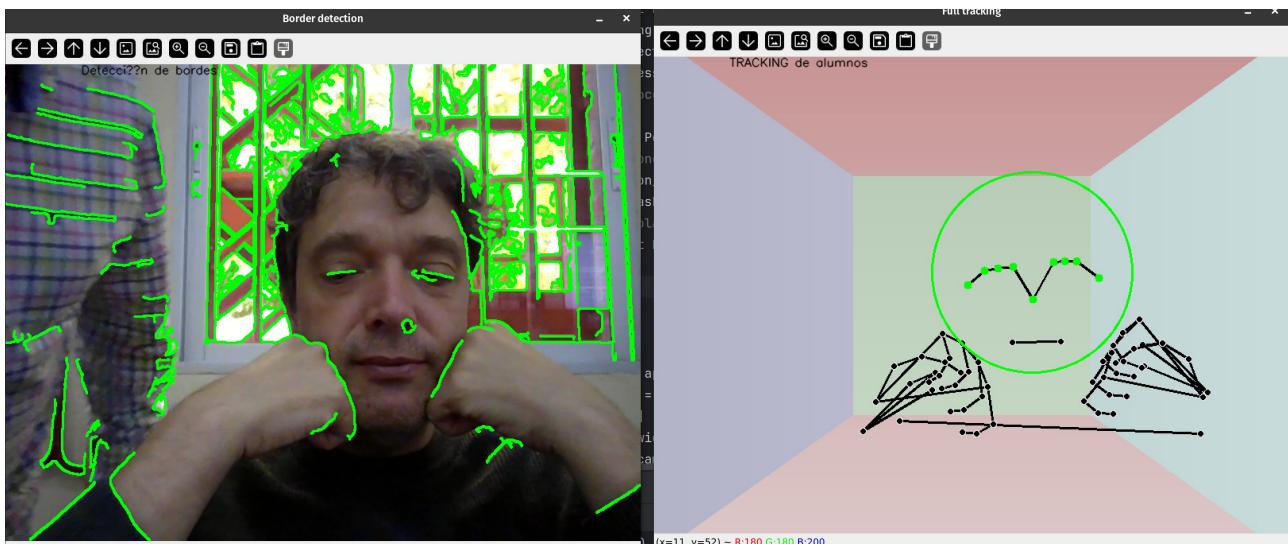


Figura 1: Ejemplo de tratamiento de imágenes con visión artificial

## 2. TECNOLOGÍAS DE VISIÓN ARTIFICIAL

La visión artificial ha avanzado enormemente gracias al desarrollo de herramientas y tecnologías que facilitan la implementación de sistemas capaces de analizar imágenes y videos. En esta sección exploraremos algunas de las tecnologías más importantes y cómo se utilizan en diferentes aplicaciones.

### 2.1. *OPENCV: LA BASE DE LA VISIÓN ARTIFICIAL CLÁSICA*

*OpenCV (Open Source Computer Vision Library)* es una biblioteca de código abierto que se ha convertido en un estándar para implementar soluciones de visión artificial. Ofrece una amplia gama de funciones para tareas como:

- Procesamiento básico de imágenes (filtros, transformaciones, detección de bordes).
- Reconocimiento de patrones y detección de objetos.
- Seguimiento de movimiento y análisis de video en tiempo real.

*OpenCV* es una excelente herramienta para proyectos que no requieren redes neuronales profundas, como el conteo de objetos en una cinta transportadora o la detección de colores específicos en una imagen.



Figura 2: La librería OpenCV

*OpenCV*, que significa *Open Source Computer Vision Library*, es una biblioteca de visión artificial y procesamiento de imágenes ampliamente utilizada en la universidad, la investigación y la industria. Fue desarrollada originalmente por Intel en el año 2000 y, desde entonces, ha evolucionado para ser compatible con múltiples plataformas y lenguajes de programación como *Python*, *C++* y *Java*.

La principal ventaja de *OpenCV* es que permite implementar soluciones eficientes y rápidas de visión artificial sin necesidad de usar redes neuronales profundas. Aunque en la actualidad se combina frecuentemente con técnicas modernas de aprendizaje profundo, *OpenCV* sigue siendo la base para muchas aplicaciones prácticas gracias a su conjunto de herramientas robustas y versátiles.

### 2.1.1. Principales Funcionalidades de *OpenCV*

*OpenCV* ofrece una amplia gama de funcionalidades que se agrupan en varias áreas de visión artificial. A continuación, se describen algunas de las más relevantes:

- **Procesamiento Básico de Imágenes:** *OpenCV* permite realizar operaciones fundamentales sobre imágenes, como:
  - Conversión entre espacios de color (RGB, HSV, escala de grises, etc.).
  - Filtrado y reducción de ruido (filtros Gaussianos, bilaterales, mediana, etc.).
  - Transformaciones geométricas (rotación, escalado, traslación, etc.).
  - Detección de bordes y contornos mediante algoritmos como *Canny* o *Sobel*.
- **Detección de Características:** *OpenCV* incluye herramientas para detectar puntos clave y características en imágenes, tales como:
  - **SIFT** (*Scale-Invariant Feature Transform*).
  - **SURF** (*Speeded-Up Robust Features*).
  - **ORB** (*Oriented FAST and Rotated BRIEF*), que es más rápido y eficiente.

Estas características se utilizan en aplicaciones como el registro de imágenes y la detección de objetos.

- **Detección de Objetos:** *OpenCV* implementa técnicas clásicas como los clasificadores en cascada de *Haar* o *HOG* (*Histogram of Oriented Gradients*) para detectar rostros, cuerpos, ojos, etc. Aunque estas técnicas son menos precisas que los modelos basados en redes neuronales profundas, son rápidas y útiles para aplicaciones ligeras.

- **Análisis de Video y Seguimiento de Objetos:** *OpenCV* se usa ampliamente para el análisis en tiempo real de secuencias de video. Algunas tareas típicas son:
  - Detección de movimiento y extracción de fondo.
  - Seguimiento de objetos con algoritmos como *KLT* (*Kanade-Lucas-Tomasi*) o *MeanShift*.
  - Análisis de flujo óptico para calcular el movimiento de los objetos entre cuadros.
- **Reconstrucción 3D:** *OpenCV* proporciona herramientas para tareas como la visión estéreo y la reconstrucción 3D a partir de imágenes o videos. Esto incluye:
  - Correspondencia estéreo para generar mapas de profundidad.
  - Estimación de la posición y orientación de objetos mediante detección de marcadores (por ejemplo *ArUco*).
- **Integración con Deep Learning:** Aunque originalmente *OpenCV* no estaba diseñado para trabajar con redes neuronales profundas, las versiones más recientes permiten integrar modelos preentrenados desarrollados en frameworks como *TensorFlow*, *PyTorch* o *Caffe*. *OpenCV* puede importar modelos en formatos estándar como *ONNX* y ejecutarlos eficientemente.

### 2.1.2. Ventajas y aplicaciones de *OpenCV*

#### Ventajas:

- **Código abierto y gratuito:** Al ser de código abierto, es accesible para todos y cuenta con una comunidad activa que contribuye con mejoras y nuevos módulos.
- **Portabilidad:** Compatible con *Windows*, *Linux*, *macOS*, y dispositivos móviles (*Android* e *iOS*).
- **Eficiencia:** Altamente optimizado para procesar imágenes y videos en tiempo real.
- **Facilidad de uso:** Su API es clara y amigable, especialmente en *Python*, lo que la hace ideal para principiantes y expertos.

#### Aplicaciones:

- **Seguridad:** Reconocimiento facial y de matrículas en sistemas de videovigilancia.
- **Industria:** Inspección de calidad en líneas de producción para detectar defectos.
- **Deportes:** Análisis del movimiento y estadísticas en transmisiones deportivas.
- **Salud:** Procesamiento de imágenes médicas para asistir en diagnósticos.
- **Entretenimiento:** Reconocimiento de gestos y efectos visuales en aplicaciones y videojuegos.

Ejemplo de *OpenCV* A continuación, se presenta un ejemplo sencillo en *Python* para cargar una imagen, convertirla a escala de grises y detectar bordes con el algoritmo de *Canny*:

```
import cv2

# Cargar una imagen
imagen = cv2.imread("imagen.jpg")
# Convertir a escala de grises
gris = cv2.cvtColor(imagen, cv2.COLOR_BGR2GRAY)
```

```
# Aplicar detección de bordes con Canny
bordes = cv2.Canny(gris, 100, 200)

# Mostrar la imagen original y los bordes detectados
cv2.imshow("Imagen Original", imagen) cv2.imshow("Bordes", bordes)

cv2.waitKey(0) cv2.destroyAllWindows()
```

Código 1: Ejemplo de *OpenCV*

Este ejemplo demuestra lo fácil que es implementar operaciones básicas de procesamiento de imágenes con *OpenCV*.

### 2.1.3. Desafíos y futuro de *OpenCV*

Aunque *OpenCV* es una herramienta poderosa, también enfrenta algunos desafíos:

- Las técnicas clásicas que implementa a menudo son menos precisas que las soluciones basadas en redes neuronales.
- El diseño de algoritmos avanzados puede requerir un conocimiento profundo de visión artificial.

Sin embargo, *OpenCV* sigue siendo una herramienta esencial en proyectos de visión artificial, y su integración con técnicas modernas asegura que continuará siendo relevante en el futuro.

## 2.2. YOLO: DETECCIÓN DE OBJETOS EN TIEMPO REAL

*YOLO* (*You Only Look Once*) es uno de los modelos más populares para la detección de objetos. A diferencia de métodos más tradicionales, *YOLO* procesa una imagen completa de una sola vez, lo que lo hace extremadamente rápido y eficiente. Sus aplicaciones incluyen:

- Sistemas de vigilancia que identifican personas o vehículos.
- Herramientas para detectar objetos en cámaras de drones.
- Soluciones de visión para el comercio minorista, como el monitoreo de estantes.

Con *YOLO*, es posible detectar múltiples objetos en tiempo real, lo que lo hace ideal para tareas que requieren una respuesta rápida.



Figura 3: La librería YOLO

*YOLO* es uno de los algoritmos más populares y revolucionarios para la detección de objetos en imágenes y videos. Su principal característica es la capacidad de realizar detecciones en tiempo real, lo que lo hace ideal para aplicaciones prácticas donde la velocidad es crucial.

*YOLO* es un modelo de detección de objetos basado en redes neuronales profundas que trata la detección como un problema de regresión. En lugar de analizar una imagen dividiéndola en múltiples regiones (como otros algoritmos tradicionales), *YOLO* procesa la imagen completa en un solo paso. Esto lo hace mucho más rápido y eficiente en comparación con modelos como *Faster R-CNN* o métodos basados en regiones.

El funcionamiento básico de *YOLO* consiste en dividir una imagen en una cuadrícula y predecir simultáneamente:

- Las coordenadas de los cuadros delimitadores (*bounding boxes*) alrededor de los objetos.
- Las clases de los objetos detectados (por ejemplo, perro, coche, persona).
- Un puntaje de confianza que indica qué tan seguro está el modelo de que el cuadro detectado contiene un objeto.

### 2.2.1. Ventajas de *YOLO*

*YOLO* se destaca por varias razones:

- **Velocidad:** Puede procesar imágenes en tiempo real, incluso en dispositivos con recursos limitados.
- **Eficiencia:** Realiza detección y clasificación de objetos en un solo paso, optimizando el uso de recursos.
- **Precisión:** A pesar de ser rápido, ofrece un rendimiento competitivo en comparación con modelos más complejos.
- **Generalización:** Funciona bien en imágenes no vistas anteriormente, lo que lo hace robusto para tareas del mundo real.

### 2.2.2. Evolución de *YOLO*

Desde su creación, *YOLO* ha pasado por múltiples versiones, cada una mejorando en términos de velocidad, precisión y facilidad de uso:

- ***YOLOv1*:** Introdujo el concepto de procesar toda la imagen de una vez, revolucionando la detección de objetos.
- ***YOLOv2* y *YOLOv3*:** Mejoraron la precisión al introducir anclas y detección en múltiples escalas, haciéndolo más efectivo para detectar objetos pequeños.
- ***YOLOv4*:** Se optimizó para trabajar en dispositivos más modestos mientras mantenía una alta precisión.
- ***YOLOv5*:** Aunque no es oficial, esta versión se ha convertido en una de las más populares por su facilidad de implementación con *PyTorch* y su soporte en la comunidad.
- ***YOLOv7*:** Una de las versiones más recientes y avanzadas, que combina velocidad y precisión a niveles sin precedentes.

- **YOLOv8:** Introducida por *Ultralytics*, esta versión incorpora técnicas avanzadas de aprendizaje automático para mejorar la precisión y la velocidad, consolidándose como una opción popular para tareas de reconocimiento de imágenes.
- **YOLOv9:** Marca un avance significativo en la detección de objetos en tiempo real, introduciendo técnicas revolucionarias como la Información de Gradiente Programable (*PGI*) y la Red de Agregación de Capas Eficiente Generalizada (*GELAN*).
- **YOLOv10:** Disponible en varias escalas de modelos para satisfacer distintas necesidades de aplicación, desde versiones nano para entornos con recursos limitados hasta versiones extragrandes para máxima precisión y rendimiento.
- **YOLOv11:** La última iteración de la serie *Ultralytics YOLO*, que redefine lo que es posible con una precisión, velocidad y eficacia de vanguardia, introduciendo mejoras significativas en la arquitectura y los métodos de entrenamiento.

### 2.2.3. Aplicaciones de *YOLO*

Gracias a su rapidez y precisión, *YOLO* se utiliza en una amplia variedad de aplicaciones:

- **Seguridad:** Detección de intrusos en sistemas de vigilancia en tiempo real.
- **Coches Autónomos:** Identificación de peatones, señales de tráfico y otros vehículos.
- **Drones:** Seguimiento y detección de objetos en imágenes aéreas.
- **Retail:** Monitoreo de inventarios y análisis de comportamiento de los clientes.
- **Medicina:** Identificación de anomalías en imágenes médicas como radiografías o tomografías.

### 2.2.4. Ejemplo de uso de *YOLO*

A continuación, se muestra un ejemplo en *Python* utilizando la implementación de *YOLOv5* en *PyTorch* para detectar objetos en una imagen:

```
# Importar la librería de YOLOv5
from yolov5 import YOLOv5

# Cargar el modelo preentrenado
model = YOLOv5('yolov5s', device='cpu')

# Cambiar a 'cuda' para usar GPU

# Detectar objetos en una imagen
results = model.predict('imagen.jpg')

# Mostrar los resultados
results.show()

# Mostrar la imagen con los cuadros detectados
# Guardar la imagen con los resultados
results.save('resultado.jpg')
```

Código 2: Ejemplo de *YOLO*

En este ejemplo, el modelo detectará objetos en una imagen y generará una salida con cuadros delimitadores, etiquetas y puntajes de confianza.

### 2.2.5. Limitaciones de *YOLO*

A pesar de sus grandes ventajas, *YOLO* también tiene algunas limitaciones:

- Puede tener dificultades para detectar objetos muy pequeños o solapados.
- Su rendimiento depende del tamaño y la calidad del conjunto de datos utilizado para el entrenamiento.
- Requiere cierto nivel de recursos computacionales para entrenar modelos desde cero.

*YOLO* sigue siendo una de las herramientas más poderosas en la visión artificial para tareas de detección de objetos. Su capacidad para realizar detecciones en tiempo real lo convierte en una solución ideal para una amplia variedad de aplicaciones prácticas. Con cada nueva versión, *YOLO* sigue mejorando y consolidándose como un estándar en la detección de objetos.

## 2.3. *MEDIPIPE: ANÁLISIS DE CUERPO, MANOS Y CARA*

*MediaPipe*, desarrollado por *Google*, es una biblioteca diseñada específicamente para tareas de análisis corporal, como el seguimiento de poses, manos y expresiones faciales. Sus aplicaciones prácticas incluyen:

- Reconocimiento de gestos para controlar dispositivos o videojuegos.
- Creación de efectos visuales en tiempo real en aplicaciones móviles.
- Estimación de pose para deportes o análisis biomecánico.

*MediaPipe* es ampliamente utilizado en aplicaciones que requieren un enfoque específico en interacciones humanas y movimientos detallados.

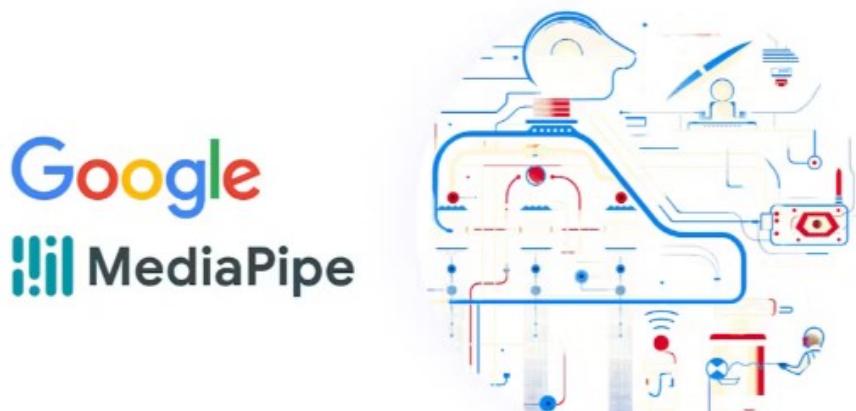


Figura 4: La librería MediaPipe

## 2.4. SEGMENTACIÓN CON SAM DE META

*SAM (Segment Anything Model)*, desarrollado por *Meta*, es un modelo avanzado diseñado para la segmentación de imágenes. La segmentación se refiere a dividir una imagen en diferentes regiones, identificando objetos y separándolos de su fondo. Las características principales de *SAM* incluyen:

- Capacidad de segmentar automáticamente cualquier objeto dentro de una imagen con un alto grado de precisión.
- Facilidad de uso, ya que no requiere un entrenamiento previo específico para cada objeto.
- Versatilidad para trabajar en tareas como segmentación médica, análisis de paisajes y edición de imágenes.

*SAM* representa un avance importante al simplificar tareas de segmentación que antes eran complejas y requerían modelos personalizados.



Figura 5: Segment Anything Model

## 2.5. OTRAS TECNOLOGÍAS

Además de las herramientas mencionadas, existen otras tecnologías que destacan en el campo de la visión artificial:

- ***Detectron2***: Una herramienta de código abierto para tareas avanzadas como detección de objetos, segmentación y *keypoints*. Es utilizada ampliamente en investigación y aplicaciones de la industria.
- ***OpenPose***: Especializada en el análisis de poses humanas, esta herramienta es ideal para el reconocimiento corporal en actividades deportivas y de entretenimiento.
- ***DeepLab***: Un modelo avanzado para la segmentación semántica que es particularmente útil en aplicaciones médicas y de mapeo geoespacial.

## 3. TRATAMIENTO DE IMÁGENES CON *OPENCV*

*OpenCV* es una biblioteca poderosa y versátil para el procesamiento de imágenes y visión por computadora en *Python*. Combinada con *NumPy*, ofrece una amplia gama de herramientas para manipular imágenes. A continuación, se presentan métodos para cargar imágenes en color y en escala de grises, rotar imágenes, ajustar el brillo, y trasladar imágenes utilizando *OpenCV* y *NumPy*.

### 3.1. CARGAR IMÁGENES

Para cargar una imagen en color o en escala de grises, utilizamos la función `cv2.imread()`. El segundo argumento de esta función determina el modo de color de la imagen cargada.

```
import cv2

# Cargar una imagen en color
image_color = cv2.imread('path/to/your/image.jpg', cv2.IMREAD_COLOR)

# Cargar una imagen en escala de grises
image_gray = cv2.imread('path/to/your/image.jpg', cv2.IMREAD_GRAYSCALE)
```

### 3.2. ROTACIÓN DE IMÁGENES USANDO NUMPY

Para rotar una imagen, podemos usar la funcionalidad de matrices de *NumPy*. Esto implica una rotación manual sin utilizar herramientas específicas de *OpenCV*, utilizando la manipulación de matrices para rotar la imagen en el plano.

```
import numpy as np

# Rotar una imagen 90 grados
height, width = image_color.shape[:2]
rotation_matrix = cv2.getRotationMatrix2D((width/2, height/2), 90, 1)
rotated_image = cv2.warpAffine(image_color, rotation_matrix, (width, height))
```

Primero, se extraen las dimensiones de la imagen original almacenadas en `image_color` mediante el acceso a las propiedades de forma de la matriz, obteniendo así el alto y ancho de la imagen. Luego, se crea una matriz de rotación utilizando la función `cv2.getRotationMatrix2D()`, donde el primer argumento es el centro de la imagen (calculado como la mitad del ancho y la altura), el segundo argumento es el ángulo de rotación de 90°, y el tercer argumento es el factor de escala, que aquí se mantiene en 1 para no alterar el tamaño de la imagen. Finalmente, la imagen rotada se genera mediante la función `cv2.warpAffine()`, que aplica la matriz de rotación a la imagen original, resultando en la imagen rotada que mantiene las dimensiones originales de ancho y alto.



Figura 6: original



Figura 7: rotada 45°

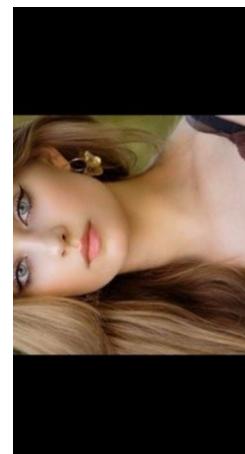


Figura 8: rotada 90°

### 3.3. AJUSTE DE BRILLO UTILIZANDO NUMPY

Ajustar el brillo de una imagen puede lograrse mediante operaciones aritméticas simples en *NumPy*.

```
# ajuste de brillo:  
import cv2  
import matplotlib.pyplot as plt  
  
# Cargar la imagen original en formato BGR->RGB  
image_color = cv2.imread('../imagenes/cara.jpg')  
image_rgb = cv2.cvtColor(image_color, cv2.COLOR_BGR2RGB)  
  
# Aumentar el brillo de la imagen  
brighter_image = cv2.add(image_rgb, 50)  
  
# Disminuir el brillo de la imagen  
darker_image = cv2.add(image_rgb, -50)  
  
fig, axes = plt.subplots(nrows=3, ncols=1, figsize=(6, 18))  
  
# Mostrar la imagen original  
axes[0].imshow(image_rgb)  
axes[0].set_title('Imagen Original')  
  
# Ocultar los ejes  
axes[0].axis('off')  
  
# Mostrar la imagen más brillante  
axes[1].imshow(brighter_image)  
axes[1].set_title('Imagen Más Brillante')  
axes[1].axis('off') # Ocultar los ejes  
  
# Mostrar la imagen más oscura  
axes[2].imshow(darker_image)  
axes[2].set_title('Imagen Más Oscura')  
  
# Ocultar los ejes  
axes[2].axis('off')  
  
plt.show()
```



Figura 9: Modificación del brillo de una imagen

### 3.4. TRASLACIÓN DE IMÁGENES

Trasladar una imagen significa moverla en el espacio  $XY$ . Esto se puede hacer creando una matriz de traslación y utilizando la función `cv2.warpAffine()`.



Figura 10: Traslación de una imagen

```
# Mover la imagen 100 píxeles hacia la derecha y 50 píxeles hacia abajo
translation_matrix = np.float32([[1, 0, 100], [0, 1, 50]])
translated_image = cv2.warpAffine(image_color, translation_matrix, (width, height))
```

### 3.5. FLIPS

Para hacer un *flip* (espejado) a una imagen utilizando la biblioteca *NumPy*, se puede emplear la funcionalidad de manipulación de *arrays*. A continuación, se describe cómo realizar *flips* horizontales y verticales a una imagen:

- **Flip Horizontal:** El *flip* horizontal se realiza invirtiendo el orden de las columnas de la imagen. Esto se logra con el siguiente código:

```
import cv2
import numpy as np

# Cargar la imagen
```

```
image = cv2.imread('path/to/your/image.jpg')

# Flip horizontal
image_flipped_horizontal = np.fliplr(image)

# Guardar o mostrar la imagen
cv2.imwrite('path/to/your/vertical_flipped.jpg', image_flipped_horizontal)
```

- **Flip Vertical:** De igual manera, el *flip* vertical se efectúa invirtiendo el orden de las filas de la imagen, como se muestra en el siguiente código:

```
# Flip vertical
image_flipped_vertical = np.flipud(image)

# Guardar o mostrar la imagen
cv2.imwrite('path/to/your/vertical_flipped.jpg', image_flipped_vertical)
```

### Explicación:

- **np.fliplr():** Esta función de NumPy invierte el orden de las columnas de un *array*. Cuando se aplica a una imagen, esto resulta en un *flip* horizontal (como mirar la imagen en un espejo de izquierda a derecha).
- **np.flipud():** Esta función invierte el orden de las filas del *array*. Al usarlo en imágenes, produce un *flip* vertical (como voltear la imagen de arriba hacia abajo).



Figura 11: Flipping de una imagen

### 3.5.1. Redimensionamiento de imágenes

Para escalar imágenes se puede usar la función `cv2.resize()`. Esta función toma como argumentos la imagen que se desea redimensionar, en este caso `image_rgb`, y una tupla que especifica las nuevas dimensiones, `(new_width, new_height)`. El tercer argumento, `interpolation=cv2.INTER_AREA`, es para determinar el método de interpolación usado durante el redimensionamiento. El uso de `cv2.INTER_AREA` es recomendado especialmente para reducir el tamaño de la imagen, ya que utiliza un método de interpolación de área que es eficaz para evitar el *aliasing* y proporciona resultados de alta calidad al minimizar el error de muestreo. Este método es particularmente útil en aplicaciones donde la precisión y la calidad de la imagen redimensionada son importantes.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Cargar la imagen original en formato BGR
image_color = cv2.imread('path/to/your/image.jpg')

# Convertir de BGR a RGB para la visualización correcta en Matplotlib
image_rgb = cv2.cvtColor(image_color, cv2.COLOR_BGR2RGB)

# Obtener las dimensiones originales de la imagen
height, width = image_rgb.shape[:2]

# Calcular las nuevas dimensiones
new_width = int(width * 0.5)
new_height = int(height * 0.5)

# Redimensionar la imagen
resized_image = cv2.resize(image_rgb, (new_width, new_height), interpolation=cv2.INTER_AREA)

# Configurar los subplots
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(10, 5))

# Mostrar la imagen original
axes[0].imshow(image_rgb)
axes[0].set_title('Imagen Original')

# Ocultar los ejes
axes[0].axis('off')

# Mostrar la imagen redimensionada
axes[1].imshow(resized_image)
axes[1].set_title('Imagen Escalada')

# Ocultar los ejes
axes[1].axis('off')
```

```
# Mostrar la figura  
plt.show()
```

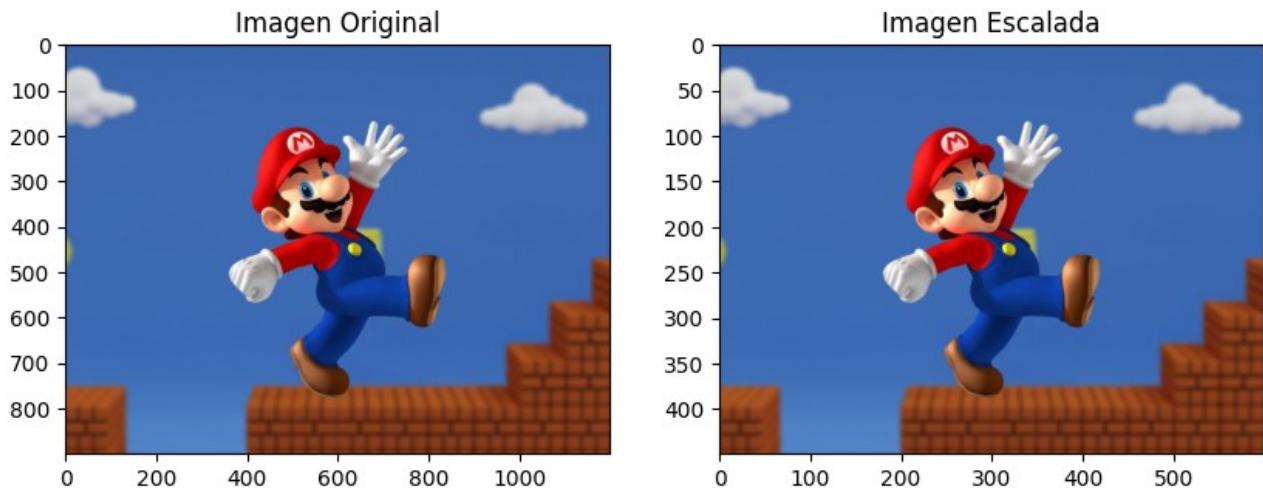


Figura 12: Modificación de la escala (tamaño) una imagen

### 3.6. OTRAS LIBRERÍAS – LA LIBRERÍA PILLOW

*Pillow* es una biblioteca de procesamiento de imágenes extensiva y fácil de usar en *Python*. Permite la manipulación de imágenes de varias maneras, incluyendo rotación, redimensionamiento y conversión de formatos. A continuación, se describen algunas de las funciones más utilizadas.

Por ejemplo, para rotar una imagen, primero debemos cargarla. Utilizamos la función *Image.open()* para abrir el archivo de imagen. Luego, podemos rotar la imagen utilizando el método *rotate()* especificando el ángulo en grados.

```
from PIL import Image  
  
# Cargar la imagen  
image = Image.open('path/to/your/image.jpg')  
  
# Rotar la imagen 90 grados  
rotated_image = image.rotate(90)  
rotated_image.save('path/to/your/rotated_image.jpg')
```

Código 3: Rotación de imágenes con *Pillow*

### 3.7. CAPTURA DE VÍDEO DESDE LA CÁMARA CON OPENCV

Un vídeo no es más que una secuencia de imágenes (denominadas *frames*) que se reproducen de forma continua a una velocidad determinada, comúnmente medida en fotogramas por segundo (FPS, por sus siglas en inglés). Por ejemplo, un vídeo con una tasa de 30 FPS significa que se muestran 30 imágenes individuales en un segundo, generando así la percepción de movimiento.

Cuando se utiliza una cámara web, el dispositivo captura imágenes en tiempo real y las envía a un software, como *OpenCV*. La cámara toma una imagen individual, la transmite, y casi

instantáneamente toma la siguiente, formando así un flujo continuo de imágenes. Este flujo se recibe y procesa *frame a frame* (imagen por imagen) en un bucle dentro del programa.

En *OpenCV*, la función *VideoCapture()* permite capturar estos *frames* de forma secuencial desde la cámara. En cada iteración de un bucle, se recibe un nuevo *frame* utilizando la función *read()*, que devuelve dos valores:

- Un valor booleano (*True* o *False*) que indica si el *frame* fue capturado correctamente.
- El *frame* actual, representado como una matriz de píxeles.

A medida que los *frames* son recibidos, pueden ser procesados, transformados o mostrados en pantalla usando *cv2.imshow()*. Este proceso continuará hasta que se detiene manualmente, permitiendo un flujo de vídeo en tiempo real.

El siguiente esquema describe este proceso:

- 1) La cámara toma un *frame* (imagen).
- 2) *OpenCV* recibe el *frame* usando la función *cap.read()*.
- 3) El *frame* se procesa o visualiza.
- 4) Se repite el proceso para el siguiente *frame*.

Este flujo continuo de imágenes forma el vídeo, dando la ilusión de movimiento cuando se reproduce a alta velocidad. *OpenCV* permite manipular cada uno de estos *frames* de manera individual, lo que es clave para aplicaciones como detección de movimiento, reconocimiento facial y filtros de imagen en tiempo real.

Capturar vídeo en tiempo real desde una cámara es una funcionalidad comúnmente utilizada en proyectos de visión artificial. *OpenCV* proporciona una interfaz sencilla para adquirir vídeo en tiempo real a través de cámaras conectadas al computador. El siguiente código en *Python* ilustra cómo utilizar *OpenCV* para capturar vídeo desde la cámara predeterminada del sistema.

```
import cv2
# Inicializar la captura de video
cap = cv2.VideoCapture(0) # El argumento 0 indica la primera cámara del sistema

# Verificar si la cámara se inicializó correctamente
if not cap.isOpened():
    print("No se pudo abrir la cámara")
    exit()

try:
    while True:
        # Capturar frame por frame
        ret, frame = cap.read()

        # Si frame se lee correctamente ret es True
        if not ret:
            print("No se puede recibir frame (stream end?). Exiting ...")
            break
```

```

# Mostrar el frame capturado
cv2.imshow('Frame', frame)

# Romper el bucle si se presiona 'q'
if cv2.waitKey(1) == ord('q'):
    break

finally:
    # Cuando todo esté hecho, liberar la captura
    cap.release()
    cv2.destroyAllWindows()

```

Este script comienza inicializando un objeto *VideoCapture* con el índice de la cámara (0 para la cámara predeterminada). La función *cap.isOpened()* verifica si la cámara se ha inicializado correctamente. Dentro del bucle, *cap.read()* captura cada *frame* de la cámara, y *cv2.imshow()* muestra el *frame*. El bucle se interrumpe si se presiona la tecla ‘q’. Finalmente, se liberan los recursos con *cap.release()* y se cierran todas las ventanas de *OpenCV* con *cv2.destroyAllWindows()*.

Esta sección provee una base sólida para aplicaciones que requieran captura de vídeo en tiempo real, como sistemas de seguridad, monitoreo en vivo o aplicaciones interactivas basadas en visión por computadora.

### 3.8. LOS FPS

Para capturar vídeo a un ratio fijo de fotogramas por segundo (FPS) en *OpenCV*, puedes controlar la tasa de adquisición de *frames* utilizando un temporizador o la función *cv2.waitKey()* en combinación con un cálculo del tiempo entre *frames*.

La idea es que si quieres capturar un video a, por ejemplo, 30 FPS, debes asegurarte de que cada *frame* se procese aproximadamente cada  $1/30$  segundos, lo que equivale a 33.33 milisegundos.

El siguiente código en *Python* utiliza *OpenCV* para capturar vídeo desde la cámara predeterminada del sistema y controla la tasa de adquisición de *frames* a un valor deseado, en este caso, **3 FPS**. Inicialmente, se configura el objeto *VideoCapture(0)* para acceder a la cámara web, verificando si la inicialización fue exitosa con *isOpened()*. El parámetro *target\_fps* determina la tasa de cuadros por segundo y se calcula la duración de cada *frame* como el inverso de los FPS (*frame\_duration*). Dentro de un bucle infinito, cada *frame* es capturado mediante *cap.read()* y se registra el tiempo al inicio del ciclo usando la función *time.time()*. Luego, se calcula el tiempo transcurrido y el tiempo restante para alcanzar la duración de *frame* deseada, introduciendo una pausa con *time.sleep()* si es necesario para mantener la tasa fija. La función *cv2.imshow()* se encarga de mostrar cada *frame* en una ventana. El bucle continuará ejecutándose hasta que se presiona la tecla ‘q’, momento en el que se liberan los recursos de la cámara con *cap.release()* y se cierran todas las ventanas usando *cv2.destroyAllWindows()*. Este enfoque asegura una tasa de adquisición de *frames* estable, incluso en sistemas con diferentes rendimientos.

```

import time
import cv2

# Inicializar la captura de video

```

```
cap = cv2.VideoCapture(0) # Captura desde la cámara predeterminada

# Verificar si la cámara se inicializó correctamente
if not cap.isOpened():
    print("No se pudo abrir la cámara")
    exit()

# Configurar los FPS deseados
target_fps = 3
frame_duration = 1 / target_fps # Duración de un frame en segundos

try:
    while True:
        start_time = time.time() # Marcar el inicio del frame

        # Capturar frame por frame
        ret, frame = cap.read()

        # Verificar si el frame fue capturado correctamente
        if not ret:
            print("No se pudo recibir el frame. Saliendo...")
            break

        # Mostrar el frame
        cv2.imshow('Frame a 30 FPS', frame)

        # Calcular el tiempo restante para alcanzar la duración del frame deseado
        elapsed_time = time.time() - start_time
        remaining_time = frame_duration - elapsed_time

        # Esperar el tiempo restante (si es positivo)
        if remaining_time > 0:
            time.sleep(remaining_time)

        # Romper el bucle si se presiona 'q'
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

finally:
    # Liberar recursos
    cap.release()
    cv2.destroyAllWindows()
```

### 3.9. CAPTURA DE CÁMARA Y PROCESAMIENTO DE FRAMES EN *OPENCV*

*OpenCV* permite capturar video desde la cámara web y realizar operaciones en tiempo real sobre cada *frame* antes de mostrarlo. Esto es útil para aplicaciones donde se requiere manipulación o análisis de video, como conversión a escala de grises, reflejo (*flip*), redimensionamiento o filtros personalizados. El siguiente código ejemplifica cómo capturar video desde la cámara predeterminada,

convertir cada *frame* a blanco y negro y mostrar tanto la imagen original como la transformada en ventanas separadas.

```
import cv2

# Inicializar la captura de video desde la cámara predeterminada
cap = cv2.VideoCapture(0)

# Verificar si la cámara se inicializó correctamente
if not cap.isOpened():
    print("No se pudo abrir la cámara")
    exit()

try:
    while True:
        # Capturar frame por frame
        ret, frame = cap.read()

        # Verificar si el frame fue capturado correctamente
        if not ret:
            print("No se pudo recibir el frame. Saliendo...")
            break

        # Convertir el frame a escala de grises
        gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

        # Mostrar la imagen original en una ventana
        cv2.imshow('Imagen Original', frame)

        # Mostrar la imagen en blanco y negro en otra ventana
        cv2.imshow('Imagen en Blanco y Negro', gray_frame)

        # Romper el bucle si se presiona 'q'
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

finally:
    # Liberar los recursos y cerrar todas las ventanas
    cap.release()
    cv2.destroyAllWindows()
```

### Descripción del Código:

El código inicia la captura de video utilizando `cv2.VideoCapture(0)`, que accede a la cámara predeterminada del sistema. Dentro de un bucle infinito, cada *frame* es capturado con `cap.read()` y verificado para asegurar que se haya recibido correctamente. A continuación, el *frame* es convertido a escala de grises utilizando la función `cv2.cvtColor()` con el parámetro `cv2.COLOR_BGR2GRAY`, que transforma una imagen a color BGR en una de escala de grises. Posteriormente, se muestran dos ventanas: una con la imagen original y otra con la imagen transformada. El bucle se mantiene activo

hasta que se presiona la tecla ‘*q*’, momento en el que se liberan los recursos de la cámara y se cierran todas las ventanas con *cv2.destroyAllWindows()*.

### Aplicaciones:

Este enfoque permite aplicar operaciones personalizadas a cada *frame* capturado en tiempo real, como:

- Conversión a escala de grises.
- Reflejo horizontal o vertical (*cv2.flip*).
- Redimensionamiento de la imagen (*cv2.resize*).
- Aplicación de filtros o efectos personalizados.

## 4. LAS REDES NEURONALES CONVOLUCIONALES

Las **Redes Neuronales Convolucionales** (CNN, por sus siglas en inglés) son una arquitectura de redes neuronales diseñada para procesar datos con estructura en forma de rejilla, como imágenes o señales temporales. Vamos a realizar una introducción a los conceptos fundamentales de las CNN, su arquitectura y aplicaciones principales en el ámbito de la inteligencia artificial.

En los últimos años, las Redes Neuronales Convolucionales (CNN) han revolucionado el campo de la inteligencia artificial, especialmente en tareas relacionadas con el procesamiento de imágenes y videos. Las CNN son una variante de las redes neuronales tradicionales que aprovechan la estructura espacial de los datos para extraer características jerárquicas, desde patrones simples hasta representaciones complejas.

Las Redes Neuronales Convolucionales son una herramienta poderosa para tareas que involucran datos con estructura espacial o temporal. Su capacidad para aprender características jerárquicas ha hecho posible avances significativos en áreas como visión por computadora, robótica y más.

### 4.1. LAS CONVOLUCIONES

La convolución es una operación fundamental en diversas áreas de la ingeniería y las matemáticas aplicadas, especialmente en el procesamiento de señales y sistemas lineales. Proporcionamos una introducción básica al concepto de convolución, sus propiedades y aplicaciones principales.

Las convoluciones tienen aplicaciones extensivas en una variedad de campos, entre ellas:

- **Procesamiento de Señales:** En este contexto, se utilizan para filtrar señales, calcular respuestas de sistemas lineales e implementar algoritmos de transformación.
- **Visión por Computadora:** Las convoluciones son esenciales en redes neuronales convolucionales (CNN) para extraer características de imágenes y datos visuales.
- **Procesamiento de Imágenes:** Permiten realizar operaciones como el suavizado, el resaltado de bordes y la detección de características.

La **convolución** de dos **funciones continuas**  $f(t)$  y  $g(t)$  se define como:

$$(f * g)(t) = \int_{-\infty}^{+\infty} f(\tau) \cdot g(t - \tau) d\tau$$

Para funciones **discretas**  $x[n]$  y  $h[n]$ , la convolución se expresa como:

$$(x * h)[n] = \sum_{k=-\infty}^{+\infty} x[k] \cdot h[n - k]$$

#### 4.1.1. Propiedades de la Convolución

La operación de convolución posee varias propiedades importantes:

- Comutatividad

$$f(t) * g(t) = g(t) * f(t)$$

- Asociatividad

$$f(t) * (g(t) * h(t)) = (f(t) * g(t)) * h(t)$$

- Distributividad

$$f(t) * (g(t) + h(t)) = f(t) * g(t) + f(t) * h(t)$$

- Existencia de elemento neutro

$$f(t) * \delta(t) = f(t)$$

donde  $\delta(t)$  es la delta de Dirac.

#### 4.1.2. Aplicaciones de la Convolución

La convolución es esencial en diversos campos, tales como:

- **Procesamiento de señales:** Permite determinar la respuesta de un sistema lineal e invariante en el tiempo a una entrada dada.
- **Análisis de sistemas:** Facilita el estudio de la respuesta impulsional de sistemas y su comportamiento ante diferentes señales de entrada.
- **Filtrado:** Se utiliza en el diseño de filtros para modificar o extraer información específica de una señal.

#### 4.1.3. Ejemplo de Convolución Discreta

Consideremos las siguientes secuencias discretas:

- Señal de entrada:  $x[n] = \{1, 2, 3\}$
- Respuesta al impulso:  $h[n] = \{0, 1, 0.5\}$

La convolución  $y[n] = x[n] * h[n]$  se calcula como:

$$y[0] = x[0] \cdot h[0] = 1 \cdot 0 = 0$$

$$y[1] = x[0] \cdot h[1] + x[1] \cdot h[0] = 1 \cdot 1 + 2 \cdot 0 = 1$$

$$y[2] = x[0] \cdot h[2] + x[1] \cdot h[1] + x[2] \cdot h[0] = 1 \cdot 0.5 + 2 \cdot 1 + 3 \cdot 0 = 2.5$$

$$y[3] = x[1] \cdot h[2] + x[2] \cdot h[1] = 2 \cdot 0.5 + 3 \cdot 1 = 4 \\ y[4] = x[2] \cdot h[2] = 3 \cdot 0.5 = 1.5$$

Por lo tanto, la señal resultante es  $y[n] = \{0, 1, 2.5, 4, 1.5\}$ .

## 4.2. ESTRUCTURA DE UNA CNN

Una **CNN**, o *Convolutional Neural Network*, está compuesta por capas especializadas que realizan diferentes operaciones para transformar y procesar los datos de entrada.

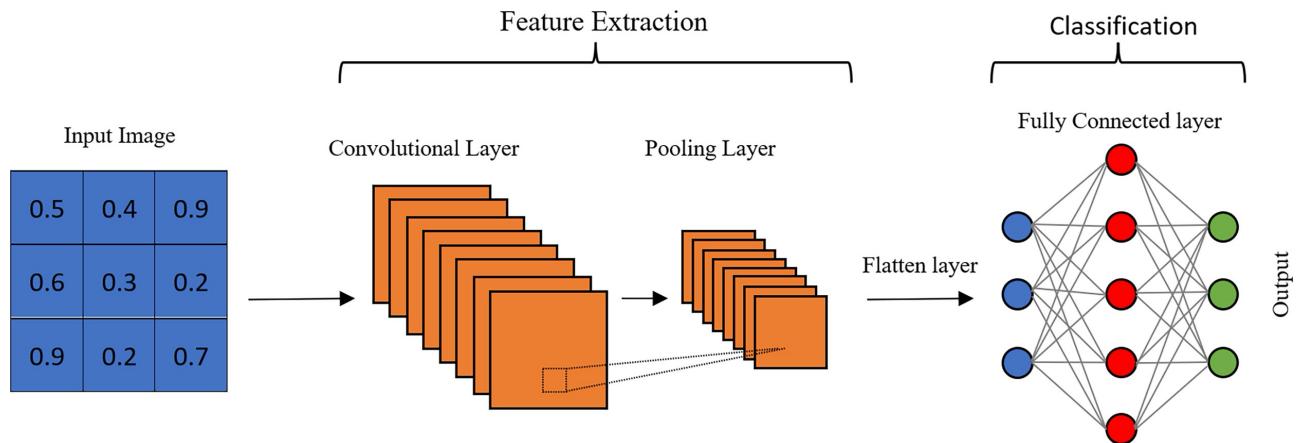


Figura 13: Arquitectura de una CNN

Las principales capas que constituyen una CNN son:

- **Capas Convolucionales**

La capa convolucional es el núcleo de una CNN. En esta capa, se aplican filtros (*kernels*) que recorren la entrada, realizando una operación de convolución para extraer características locales, como bordes, texturas y formas.

El cálculo de la convolución para un filtro  $W$  y una entrada  $X$  se define como:

$$Y[i, j] = \sum_m \sum_n X[i + m, j + n] \cdot W[m, n]$$

- **Capas de Pooling**

Las capas de *pooling* se utilizan para reducir la dimensionalidad de los datos, manteniendo las características más relevantes. Dos métodos comunes son:

- **Max Pooling:** Selecciona el valor máximo en cada región.
- **Average Pooling:** Calcula el promedio de los valores en cada región.

- **Capas Densas**

Al final de la arquitectura, las capas densas (*fully connected layers*) conectan todas las neuronas de una capa a las de la siguiente. Estas capas se utilizan para realizar tareas como clasificación o regresión.

- **Funciones de Activación**

Las funciones de activación introducen no linealidad en la red. Las funciones comunes en CNN incluyen:

- **ReLU (Rectified Linear Unit):**  
$$f(x) = \max(0, x)$$
- **Softmax:** Convierte los valores en probabilidades para tareas de clasificación.

### 4.3. ENTRENAMIENTO DE UNA CNN

Las CNN se entrena utilizando el algoritmo de retropropagación y un conjunto de datos etiquetados. El objetivo es minimizar una función de pérdida, como la entropía cruzada para clasificación. El proceso incluye:

- 1) **Cálculo hacia adelante:** Propagar los datos de entrada a través de la red.
- 2) **Cálculo de pérdida:** Evaluar el error entre la predicción y la etiqueta verdadera.
- 3) **Retropropagación:** Ajustar los pesos de la red utilizando gradientes.
- 4) **Actualización de pesos:** Aplicar un optimizador, como el descenso de gradiente estocástico (SGD).

#### 4.3.1. *Kernels* de una CNN

En el contexto de las Redes Neuronales Convolucionales, los **filtros** (o *kernels*) son matrices pequeñas que actúan como detectores de características específicas dentro de los datos de entrada. Estas características pueden ser bordes, texturas, patrones, o incluso representaciones más abstractas en capas profundas.

##### Definición de un Filtro

Un filtro es una matriz de pesos de tamaño fijo, por ejemplo,  $3 \times 3$ ,  $5 \times 5$ , o  $7 \times 7$ . Estos pesos se aprenden durante el proceso de entrenamiento y determinan qué características se extraen de la entrada. La operación de convolución aplica este filtro a regiones locales de los datos de entrada para generar una nueva representación, conocida como mapa de características (*feature map*).

##### Concepto de Receptivo Local

A diferencia de las capas densas, donde cada neurona se conecta a toda la entrada, los filtros operan en regiones locales de los datos. Esto se denomina campo receptivo, que representa el área de la entrada a la que un filtro es sensible. Por ejemplo, en una imagen, un filtro  $3 \times 3$  solo analiza una pequeña región de  $3 \times 3$  píxeles en cada paso.

##### Tipos de Características Extraídas

El comportamiento de un filtro depende de los valores de sus pesos. Algunos ejemplos de características que un filtro puede aprender incluyen:

- **Bordes horizontales:** Un filtro sensible a cambios en la intensidad entre filas.
- **Bordes verticales:** Detecta transiciones entre columnas de píxeles.
- **Texturas:** Captura patrones repetitivos como rayas o puntos.
- **Representaciones abstractas:** En capas más profundas, los filtros detectan características como formas completas u objetos.

## Importancia de Múltiples Filtros

En una CNN, cada capa convolucional utiliza múltiples filtros para extraer diferentes tipos de información de la entrada. Por ejemplo, si una capa tiene 64 filtros, producirá 64 mapas de características, cada uno destacando una característica única.

## Peso Compartido y Reducción de Parámetros

Un aspecto clave de los filtros es que los mismos pesos se utilizan en toda la entrada, desplazándose por ella. Esto reduce significativamente la cantidad de parámetros en comparación con las capas densas y ayuda a la red a generalizar mejor.

## Ejemplo Intuitivo

Imaginemos un filtro  $3 \times 3$  diseñado para detectar bordes verticales:

$$W = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Cuando este filtro se aplica a una imagen, las regiones donde hay un cambio brusco en la intensidad horizontal (bordes verticales) producen valores altos en el mapa de características, mientras que las regiones uniformes producen valores cercanos a cero.

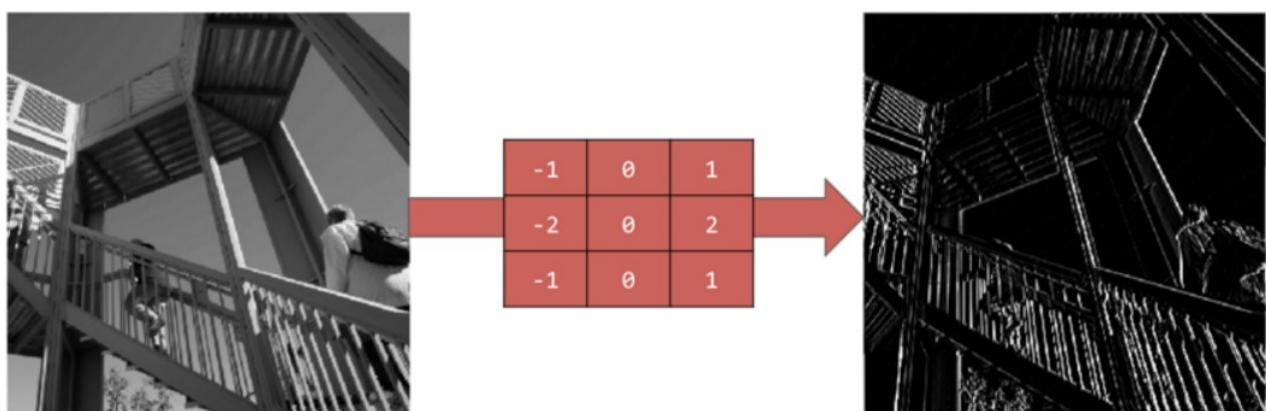


Figura 14: Filtro en una CNN

## Impacto del Tamaño del Filtro

El tamaño del filtro afecta la cantidad de detalle capturado. Filtros pequeños, como  $3 \times 3$ , se utilizan para detectar características locales, mientras que filtros más grandes capturan patrones más globales, a costa de mayor consumo computacional.

## Relación con el Aprendizaje

Durante el entrenamiento, los valores de los filtros se optimizan para que los mapas de características resalten las propiedades más relevantes para la tarea específica, como clasificación o detección de objetos. Esto permite que la red “aprenda” a identificar patrones en los datos.

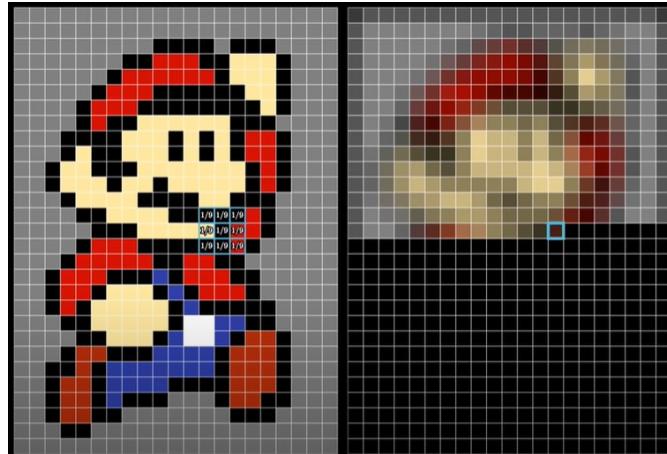


Figura 15: Aplicación del filtro a una imagen

## 4.4. CABECERA DE CLASIFICACIÓN EN UNA CNN

Una cabecera de clasificación es un componente que se añade al final de la arquitectura de una Red Neuronal Convolucional (CNN) para realizar tareas específicas, como clasificación o predicción. Este módulo transforma las características extraídas por las capas convolucionales y de *pooling* en una salida interpretable, como probabilidades asociadas a diferentes clases.

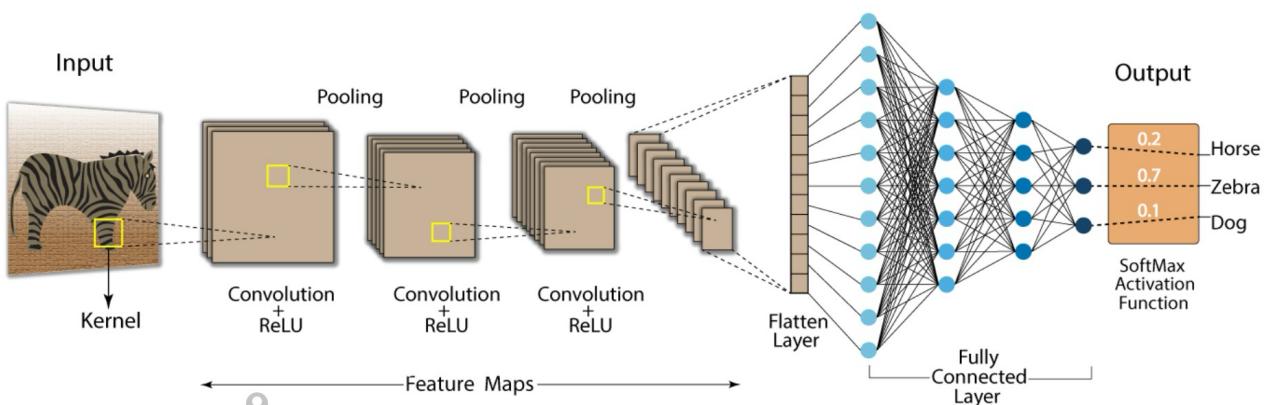


Figura 16: CNN con clasificador

### 4.4.1. Estructura de la Cabecera de Clasificación

La cabecera de clasificación típicamente incluye los siguientes elementos:

- **Capa de Aplanamiento (Flatten Layer):** Convierte las características multidimensionales obtenidas de las capas convolucionales y de *pooling* en un vector unidimensional. Esto facilita la entrada a las capas densas.

- **Capas Densas (Fully Connected Layers):** Consisten en una o más capas densas que procesan el vector aplanado para aprender combinaciones de características relevantes para la tarea de clasificación.
- **Capa de Salida:** Esta capa contiene tantas neuronas como clases en el problema de clasificación. Utiliza una función de activación, como *softmax*, para convertir las salidas en probabilidades normalizadas:

$$P(y_i|X) = \frac{e^{z_i}}{\sum_{j=1}^C e^{z_j}}$$

donde  $z_i$  representa la salida de la neurona  $i$ , y  $C$  es el número total de clases.

#### 4.4.2. Integración en la CNN

La cabecera de clasificación se integra directamente al final de las capas convolucionales y de *pooling*. En términos de diseño, la arquitectura típica es:

- **Capas iniciales:** Una serie de capas convolucionales y de pooling que actuán como extractoras de características.
- **Cabecera de clasificación:** Un módulo que toma las características extraídas y las transforma en predicciones de clase.

#### 4.4.3. Ejemplo de Implementación

Supongamos una CNN diseñada para clasificar imágenes en 10 clases diferentes. La cabecera de clasificación podría ser:

- 1) Una capa de aplanamiento para transformar las características 3D en un vector 1D.
- 2) Una capa densa con 128 neuronas y función de activación ReLU.
- 3) Una capa de salida con 10 neuronas y función de activación softmax.

La arquitectura de la cabecera se puede expresar como:

Flatten → Dense (128, ReLU) → Dense (10, Softmax).

#### 4.4.4. Ventajas de la Cabecera de Clasificación

- Permite utilizar las características aprendidas por la CNN para resolver problemas específicos.
- Es altamente flexible y puede adaptarse a diferentes tipos de tareas, como clasificación binaria, multiclasificación o incluso regresión.
- Facilita la interpretación del modelo al producir probabilidades asociadas a las diferentes clases.

#### 4.4.5. Aplicaciones

El uso de cabeceras de clasificación es común en tareas como:

- Clasificación de imágenes (detección de objetos, identificación facial).
- Análisis médico (clasificación de tejidos en imágenes médicas).

- Clasificación de texto a partir de imágenes (OCR con redes convolucionales).

Esta cabecera actúa como el puente entre las características extraídas por la CNN y el resultado final deseado, lo que la convierte en un componente esencial de muchas arquitecturas modernas.

## 5. APRENDIZAJE POR TRANSFERENCIA

El aprendizaje por transferencia es una técnica de aprendizaje automático en la que un modelo entrenado en una tarea se reutiliza como punto de partida para entrenar un modelo en una segunda tarea. Esta metodología es particularmente útil cuando los datos disponibles para la tarea objetivo son limitados, ya que permite aprovechar los conocimientos adquiridos previamente en un modelo preentrenado.

### Ventajas del Aprendizaje por Transferencia

- Reducción del tiempo de entrenamiento al utilizar modelos preentrenados.
- Mejora del rendimiento en tareas con datos limitados.
- Menor necesidad de recursos computacionales.

### Ejemplo en Python

A continuación, se muestra un ejemplo de cómo implementar el aprendizaje por transferencia utilizando un modelo preentrenado, como *ResNet50*, en un marco de aprendizaje profundo:

```
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.optimizers import Adam

# Cargar el modelo preentrenado ResNet50 sin la última capa
base_model = ResNet50(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

# Congelar las capas del modelo base
for layer in base_model.layers:
    layer.trainable = False

# Crear un nuevo modelo con capas personalizadas
model = Sequential([
    base_model,
    Flatten(),
    Dense(256, activation='relu'),
    Dense(10, activation='softmax')
    # Para una tarea de clasificación con 10 clases])

# Compilar el modelo
model.compile(optimizer=Adam(learning_rate=0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

```
# Entrenar el modelo
model.fit(train_data, train_labels, epochs=10, validation_data=(val_data,
    val_labels))
```

Código 4: Ejemplo de aprendizaje por transferencia en Python

En este ejemplo, se utiliza la red *ResNet50* preentrenada con el conjunto de datos *ImageNet* como base. Las capas del modelo base se congelan para conservar los pesos entrenados, y se añade una nueva cabeza personalizada para adaptar el modelo a la tarea específica.

## 5.1. TIPOS DE APRENDIZAJE POR TRANSFERENCIA

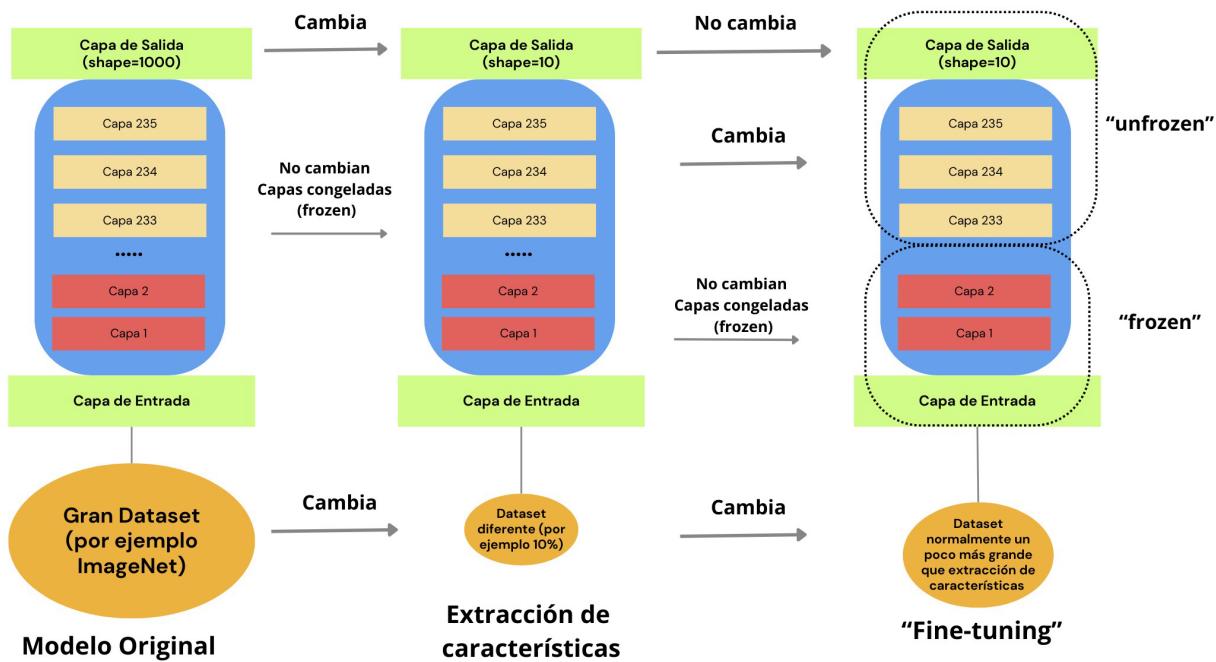


Figura 17: Transfer Learning

El aprendizaje por transferencia puede clasificarse en dos enfoques principales según el grado de ajuste realizado al modelo preentrenado: extracción de características y ajuste fino (*fine-tuning*). A continuación, se describen ambos métodos en detalle.

### 5.1.1. Extracción de Características

En este enfoque, se utiliza el modelo preentrenado como extractor de características, manteniendo congeladas (*frozen*) las capas ya entrenadas. Estas capas capturan representaciones generales del conjunto de datos original, como bordes, texturas y formas, que son útiles para tareas generales.

- **Capas congeladas:** Las capas inferiores del modelo original no se modifican, preservando el conocimiento adquirido en un gran conjunto de datos (por ejemplo, *ImageNet*).
- **Capas de salida:** Se reemplaza la capa de salida del modelo original (que podría tener una forma de salida específica, como 1000 clases en *ImageNet*) por una capa adaptada a la tarea objetivo, como 10 clases.

- **Dataset pequeño:** Este enfoque es ideal cuando el conjunto de datos de la tarea objetivo es pequeño, ya que reduce la necesidad de grandes volúmenes de datos para entrenar el modelo.

### 5.1.2. Ajuste Fino (*fine-tuning*)

El ajuste fino implica descongelar algunas capas superiores del modelo preentrenado para ajustarlas al conjunto de datos objetivo. Esto permite que el modelo refine su aprendizaje y capture características específicas de la tarea en cuestión.

- **Capas descongeladas (*unfrozen*):** En este caso, además de reemplazar la capa de salida, algunas de las capas superiores del modelo original se entranan nuevamente. Esto permite al modelo aprender características especializadas sin olvidar el conocimiento base.
- **Capas congeladas:** Las capas inferiores permanecen congeladas, preservando las representaciones generales aprendidas.
- **Dataset más grande:** El ajuste fino suele requerir un conjunto de datos más grande que el de extracción de características para evitar sobreajustes.

Mientras que la extracción de características es rápida y eficiente, el ajuste fino ofrece mayor flexibilidad y puede mejorar el rendimiento en tareas complejas, a costa de mayor tiempo de entrenamiento y requisitos de datos. Ambos enfoques son complementarios y su elección depende de la disponibilidad de datos y la naturaleza de la tarea objetivo.

## 6. VISIÓN ARTIFICIAL AVANZADA

Las herramientas modernas como *YOLO* (*You Only Look Once*) y *MediaPipe* permiten construir aplicaciones avanzadas de visión por computadora con gran eficiencia. Estas herramientas ofrecen modelos preentrenados que pueden detectar y procesar objetos, gestos, rostros, entre otros, con alta precisión. A continuación, se explica cómo instalarlas y utilizarlas para desarrollar aplicaciones prácticas.

### 6.1. INSTALACIÓN DE YOLO Y MEDIPIPE

Ambas herramientas se instalan fácilmente con *pip*:

```
# Para YOLO (usando la librería ultralytics):
pip install ultralytics

# Para MediaPipe:
pip install mediapipe
```

### 6.2. USO DE YOLO PARA DETECCIÓN DE OBJETOS

*YOLO* es una herramienta altamente eficiente para la detección de objetos en tiempo real. A continuación se muestra un fragmento de código que captura vídeo de la cámara y detecta objetos utilizando un modelo *YOLO* preentrenado:

```
from ultralytics import YOLO
import cv2
```

```
# Cargar el modelo YOLO preentrenado
model = YOLO('yolov8n.pt')

# Inicializar la cámara
cap = cv2.VideoCapture(0)

while True:
    ret, frame = cap.read()
    if not ret:
        break

    # Realizar detecciones
    results = model(frame)

    # Dibujar las detecciones en el frame
    for result in results:
        for box in result.boxes:
            x1, y1, x2, y2 = map(int, box.xyxy[0].tolist())
            label = model.names[int(box.cls[0])]
            conf = box.conf[0]
            cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 255, 0), 2)
            cv2.putText(frame, f'{label} {conf:.2f}', (x1, y1 - 10),
                       cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)

    cv2.imshow('YOLO Detection', frame)

    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

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

Código 5: Detección de objetos en tiempo real con *YOLO*

El código presentado muestra cómo utilizar *YOLO* junto con *OpenCV* para realizar detección de objetos en tiempo real. Primero, se carga el modelo preentrenado *yolov8n.pt* utilizando la clase *YOLO* de la librería *ultralytics*. Este modelo contiene la capacidad de detectar varios objetos en los *frames* que se le proporcionen.

A continuación, se inicializa la cámara mediante *OpenCV* con *cv2.VideoCapture(0)*, que captura *frames* de video en tiempo real. Estos *frames* son procesados dentro de un bucle que se ejecuta continuamente. En cada iteración, el código verifica que la captura del *frame* sea exitosa y, si lo es, se pasa al modelo *YOLO* para realizar las detecciones.

El modelo procesa el *frame* y genera resultados que incluyen información sobre los objetos detectados, como sus coordenadas, la clase a la que pertenecen y el nivel de confianza. Este es el paso donde los resultados pueden procesarse o visualizarse, dependiendo del requerimiento. En este ejemplo, los resultados se imprimen directamente en consola para su inspección.

Finalmente, opcionalmente se puede mostrar el video procesado en tiempo real utilizando `cv2.imshow()`, lo que permite visualizar los *frames* tal como los captura la cámara. El bucle se detiene al presionar la tecla ‘*q*’, y todos los recursos asociados a la cámara y las ventanas se liberan correctamente utilizando `cap.release()` y `cv2.destroyAllWindows()`.

### 6.3. CLASES DE OBJETOS EN YOLO

El modelo *YOLOv8*, utilizado en el siguiente código, está preentrenado con el conjunto de datos *COCO* (*Common Objects in Context*), un dataset ampliamente utilizado en tareas de visión artificial que contiene 80 clases diferentes de objetos comunes. El conjunto de datos *COCO* puede consultarse en su sitio oficial: <https://cocodataset.org/#home>. El modelo preentrenado incluye una lista de clases que puede ser accedida directamente después de cargar el modelo. A continuación, se presenta el código que permite listar y mostrar estas clases:

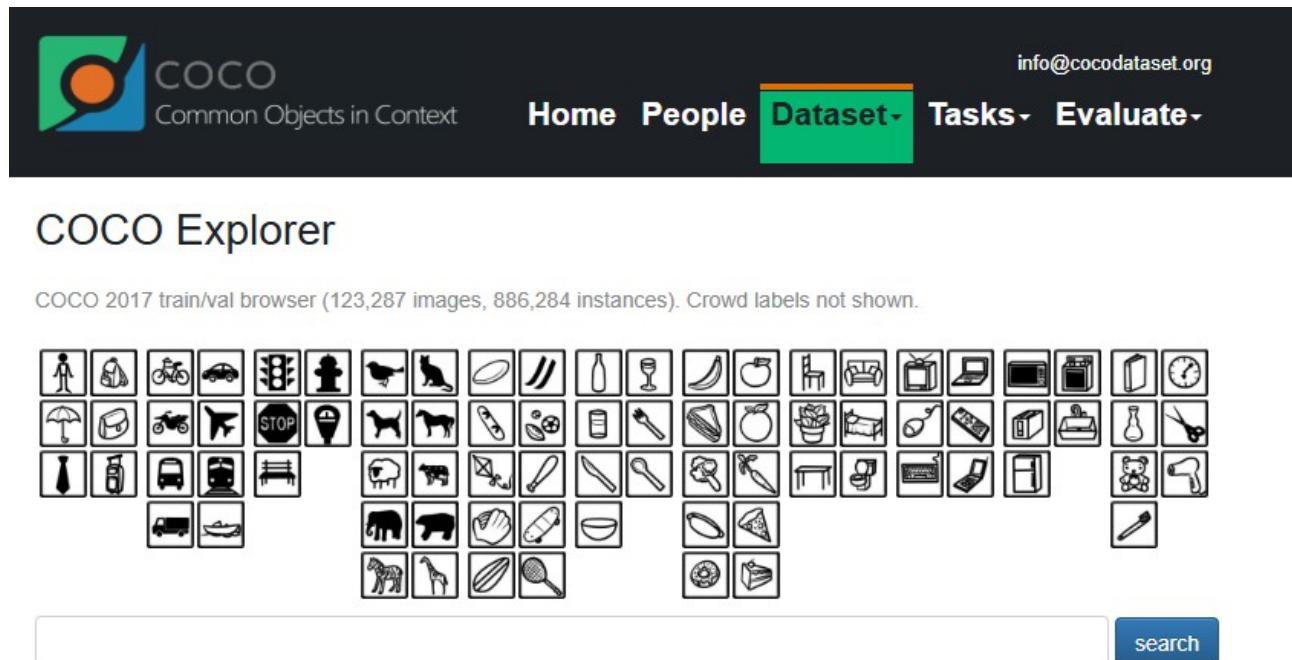


Figura 18: Coco dataset

```
from ultralytics import YOLO

# Cargar el modelo preentrenado con https://cocodataset.org/#home
model = YOLO('yolov8n.pt')

# Listar las clases detectadas
class_names = model.names # Diccionario con ID de clase como clave y nombre como valor

# Imprimir las clases
def print_classes():
    print("Clases detectadas por YOLOv8:")
    for class_id, class_name in class_names.items():
        print(f"{class_id}: {class_name}")
```

```
print_classes()
```

Código 6: Listado de clases detectadas por *YOLOv8*

Este código utiliza la librería *ultralytics* para cargar el modelo *YOLOv8* preentrenado desde el archivo *yolov8n.pt*. Una vez cargado, las clases que el modelo es capaz de detectar pueden obtenerse accediendo al atributo *names*, que devuelve un diccionario donde las claves representan el *ID* numérico de cada clase y los valores son los nombres de las clases correspondientes.

Al iterar sobre este diccionario, se imprimen las clases detectadas por el modelo junto con su identificador. Esto es útil para comprender qué objetos pueden ser detectados por el modelo en aplicaciones específicas.

El modelo preentrenado basado en *COCO* incluye clases como *person*, *bicycle*, *car*, entre muchas otras, cubriendo una amplia variedad de categorías comunes que hacen de *YOLO* una herramienta versátil para tareas de detección de objetos en diversos contextos.

#### 6.4. FILTRADO DE CLASES EN *YOLOV8*

Este código muestra cómo utilizar *YOLOv8* para detectar objetos en tiempo real y filtrar sólo aquellas clases de interés especificadas por el usuario. En este caso, se seleccionan dos clases: *person* y *cell phone*. La implementación se realiza utilizando la interfaz de estilo *PyTorch* que proporciona la librería *ultralytics*.

```
import cv2
from ultralytics import YOLO

# Cargar el modelo YOLOv8 (por ejemplo, YOLOv8n o YOLOv8s)
model = YOLO('yolov8n.pt') # Asegúrate de que tienes el archivo del modelo YOLOv8
descargado

# Inicializar la cámara
cap = cv2.VideoCapture(0) # 0 para la cámara principal

if not cap.isOpened():
    print("Error: No se pudo acceder a la cámara.")
    exit()

# Etiquetas de interés
target_classes = ['person', 'cell phone']

while True:
    # Capturar el frame de la cámara
    ret, frame = cap.read()
    if not ret:
        print("Error: No se pudo capturar el frame.")
        break

    # Realizar la detección
    results = model(frame)
```

```

# Iterar sobre las detecciones
for result in results:
    boxes = result.boxes # Coordenadas de las cajas
    for box in boxes:
        # Obtener información de la caja
        x1, y1, x2, y2 = box.xyxy[0].cpu().numpy()
        conf = box.conf[0].cpu().numpy()      # Confianza
        cls = box.cls[0].cpu().numpy()        # Clase detectada
        label = model.names[int(cls)]       # Nombre de la clase

        # Filtrar por clases de interés
        if label in target_classes:
            # Dibujar la caja en el frame
            cv2.rectangle(frame, (int(x1), int(y1)),
                          (int(x2), int(y2)), (0, 255, 0), 2)
            cv2.putText(frame, f"{label} {conf:.2f}",
                        (int(x1), int(y1) - 10),
                        cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)

    # Mostrar el frame con las detecciones
    cv2.imshow("Detcción YOLOv8", frame)

    # Presiona 'q' para salir
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

# Liberar los recursos
cap.release()
cv2.destroyAllWindows()

```

Código 7: Filtrado de clases de interés con *YOLOv8*

En este código, el modelo *YOLOv8* se carga utilizando la clase *YOLO* de *ultralytics*. Esta clase sigue un estilo similar al de *PyTorch*, lo que facilita el acceso a las detecciones y la manipulación de los resultados en tiempo real.

El modelo se inicializa con el archivo *yolov8n.pt*, que contiene un modelo preentrenado basado en el conjunto de datos *COCO*. Una vez cargado el modelo, se define una lista de clases de interés, en este caso *person* y *cell\_phone*. Durante la ejecución del bucle, cada *frame* capturado de la cámara es procesado por el modelo para realizar las detecciones.

Las detecciones se iteran a nivel de cajas delimitadoras, extrayendo información clave como las coordenadas de las cajas, el nivel de confianza y la clase detectada. Para filtrar las detecciones, se compara el nombre de la clase detectada con las clases de interés definidas. Si el nombre coincide, se dibuja un rectángulo alrededor del objeto detectado y se añade una etiqueta con el nombre de la clase y la confianza correspondiente.

## 6.5. USO DE MEDIPIPE PARA DETECCIÓN DE POSES Y GESTOS

*MediaPipe* es ideal para tareas como el reconocimiento de poses humanas y gestos en tiempo real. A continuación se presenta un ejemplo de código para detectar poses humanas:

```
import cv2
import mediapipe as mp
mp_pose = mp.solutions.pose
pose = mp_pose.Pose()
mp_drawing = mp.solutions.drawing_utils

# Inicializar la cámara
cap = cv2.VideoCapture(0)

while cap.isOpened():
    ret, frame = cap.read()
    if not ret:
        break

    # Convertir a RGB
    rgb_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

    # Procesar el frame
    results = pose.process(rgb_frame)

    # Dibujar las poses detectadas
    if results.pose_landmarks:
        mp_drawing.draw_landmarks(frame, results.pose_landmarks,
                                  mp_pose.POSE_CONNECTIONS)

    cv2.imshow('MediaPipe Pose', frame)

    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

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

Código 8: Detección de poses humanas con MediaPipe

Este ejemplo se basa en la solución *mp.solutions.pose*, que proporciona un modelo preentrenado optimizado para la estimación de poses.

Para comenzar, se importa el módulo *mediapipe* y se inicializa el componente de detección de poses con *mp.solutions.pose.Pose()*. Además, se utiliza *mp.solutions.drawing\_utils* para dibujar las marcas de las articulaciones y las conexiones detectadas por el modelo en la imagen.

La captura de video se realiza mediante *OpenCV* utilizando *cv2.VideoCapture(0)*, que abre la cámara predeterminada del dispositivo. En cada iteración del bucle, el código verifica si el *frame* fue

capturado correctamente. Una vez que el *frame* es capturado, se convierte de formato *BGR* a *RGB*, ya que *MediaPipe* requiere que las imágenes de entrada estén en formato *RGB*.

El modelo procesa el *frame* convertido mediante el método *pose.process()*, el cual devuelve las marcas de las articulaciones y las conexiones detectadas. Si el modelo detecta con éxito las poses humanas, estas se dibujan sobre el *frame* original utilizando *mp.solutions.drawing\_utils.draw\_landmarks()*. Este paso visualiza las articulaciones y las conexiones entre ellas, permitiendo un seguimiento claro y en tiempo real de las poses.

El *frame* con las poses dibujadas se muestra en una ventana de *OpenCV* utilizando *cv2.imshow()*, que permite al usuario observar el procesamiento en tiempo real. El bucle continuará hasta que se presiona la tecla *q*, momento en el cual se liberan los recursos de la cámara con *cap.release()* y se cierran todas las ventanas creadas con *cv2.destroyAllWindows()*.

## 6.6. MEDIPIPE MODEL MAKER

*MediaPipe Model Maker* es una herramienta de *MediaPipe* diseñada para entrenar modelos personalizados de aprendizaje automático con facilidad, centrándose en aplicaciones de visión artificial y otros tipos de datos multimedia.

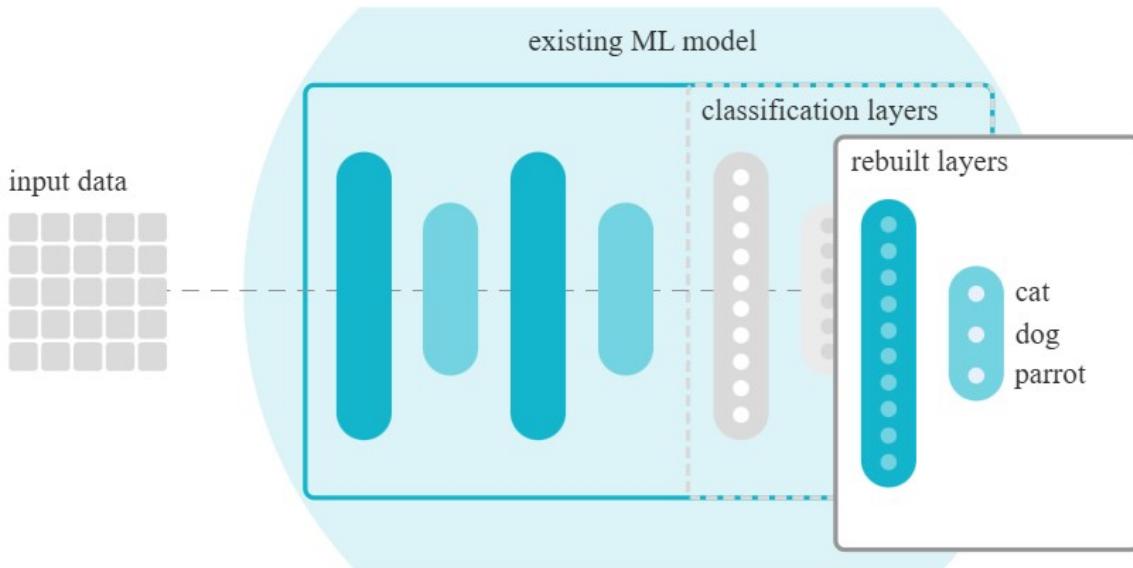


Figura 19: Transfer Learning con Mediapipe Model Maker

### 6.6.1. ¿Qué es MediaPipe Model Maker?

Esta herramienta facilita el proceso de entrenamiento y personalización de modelos preexistentes mediante aprendizaje por transferencia. Permite a los desarrolladores personalizar soluciones para aplicaciones específicas, utilizando datos propios, sin necesidad de crear un modelo desde cero.

### 6.6.2. Casos de Uso

MediaPipe Model Maker es ideal para tareas como:

- **Clasificación de Imágenes:** Entrenar un modelo para clasificar imágenes en diferentes categorías personalizadas (por ejemplo, identificar especies de plantas o clasificar productos).
- **Detección de Objetos:** Ajustar un modelo para detectar objetos específicos en imágenes o videos.
- **Segmentación de Imágenes:** Realizar segmentación semántica para aplicaciones como mapeo o procesamiento de imágenes médicas.
- **Reconocimiento de Poses:** Personalizar un modelo para reconocer poses o gestos específicos.

#### 6.6.3. Características Principales

- **Facilidad de Uso:** Simplifica el proceso de entrenamiento con una API intuitiva y modular.
- **Compatibilidad con Aprendizaje por Transferencia:** Utiliza modelos preentrenados optimizados, lo que permite personalizarlos con datasets específicos de manera rápida.
- **Optimización para Dispositivos Móviles y en Tiempo Real:** Los modelos producidos están optimizados para ejecutarse en dispositivos móviles (*Android*, *iOS*) y en aplicaciones de baja latencia.
- **Integración con MediaPipe Solutions:** Los modelos personalizados pueden integrarse fácilmente con las soluciones de *MediaPipe*, como la detección de manos, caras o poses.

#### 6.6.4. Flujo de Trabajo

- 1) **Preparar un Dataset:** Reunir datos (imágenes o videos) relevantes para tu aplicación.
- 2) **Configurar el Entrenamiento:** Usar *MediaPipe Model Maker* para cargar los datos y entrenar un modelo basado en un modelo preentrenado.
- 3) **Exportar el Modelo:** Guardar el modelo entrenado en un formato optimizado (como *TFLite*) para su implementación.
- 4) **Implementación:** Integrar el modelo con aplicaciones en tiempo real usando la librería *MediaPipe*.

#### 6.6.5. Instalación

Para empezar con *MediaPipe Model Maker*, puedes instalarlo usando:

```
pip install mediapipe-model-maker
```

#### 6.6.6. Ejemplo en Python: Clasificación de Imágenes

```
from mediapipe_model_maker import image_classifier
from mediapipe_model_maker.image_classifier import Dataset

# Cargar el dataset
data = Dataset.from_folder("ruta/a/dataset")

# Dividir en entrenamiento y validación
train_data, validation_data = data.split(0.8)
```

```
# Entrenar el modelo
model = image_classifier.create(train_data)

# Evaluar el modelo
loss, accuracy = model.evaluate(validation_data)

# Exportar el modelo como TFLite
model.export(export_dir="modelo_exportado")
```

Código 9: Ejemplo de clasificación de imágenes con MediaPipe Model Maker

## 6.7. TENSORFLOW LITE (TFLITE)

*TensorFlow Lite*, conocido como *TFLite*, es una herramienta diseñada para ejecutar modelos de aprendizaje automático en dispositivos con recursos limitados, como teléfonos móviles, dispositivos embebidos y microcontroladores. Forma parte del ecosistema de *TensorFlow* y se centra en la optimización de modelos para permitir su uso en tiempo real, con baja latencia y alta eficiencia energética.

El objetivo principal de *TFLite* es llevar las capacidades de aprendizaje profundo a dispositivos fuera de los entornos tradicionales de computación en la nube o servidores de alto rendimiento. Esto permite implementar aplicaciones inteligentes directamente en dispositivos personales o *IoT*, como asistentes virtuales, cámaras inteligentes o wearables.

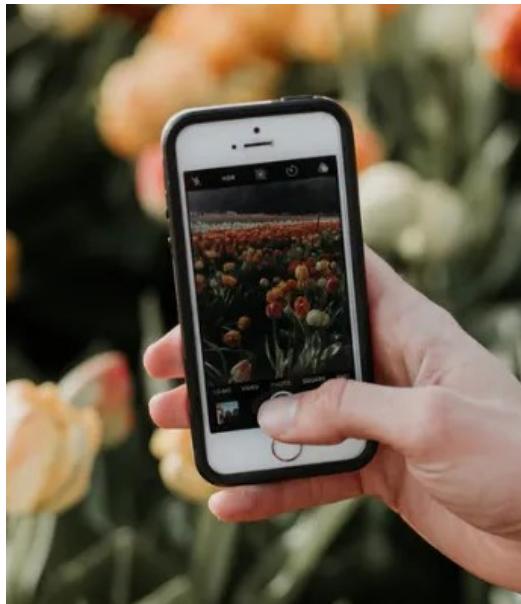


Figura 20: TFLite para móviles

*TFLite* utiliza un formato optimizado que reduce el tamaño de los modelos y su consumo computacional. Los modelos se convierten desde el formato estándar de *TensorFlow* a un formato compacto y eficiente que puede ser ejecutado por el intérprete de *TensorFlow Lite*. Este intérprete es liviano y está diseñado para funcionar en arquitecturas de hardware variadas, incluyendo CPUs, GPUs, y aceleradores especializados como TPUs (Tensor Processing Units).

Una de las características más destacadas de *TFLite* es su capacidad para soportar técnicas de optimización avanzadas como la cuantización, la poda y la fusión de operaciones. Por ejemplo, la cuantización reduce el tamaño del modelo y mejora la velocidad de inferencia al representar los pesos y las activaciones con precisión reducida, como enteros en lugar de números en punto flotante.

Otra ventaja de *TFLite* es su compatibilidad multiplataforma. Los modelos pueden integrarse fácilmente en aplicaciones móviles a través de frameworks como *Android Studio* o en aplicaciones iOS con *Core ML*. Además, el soporte para dispositivos embebidos permite su uso en plataformas como *Raspberry Pi* y microcontroladores basados en *ARM*.

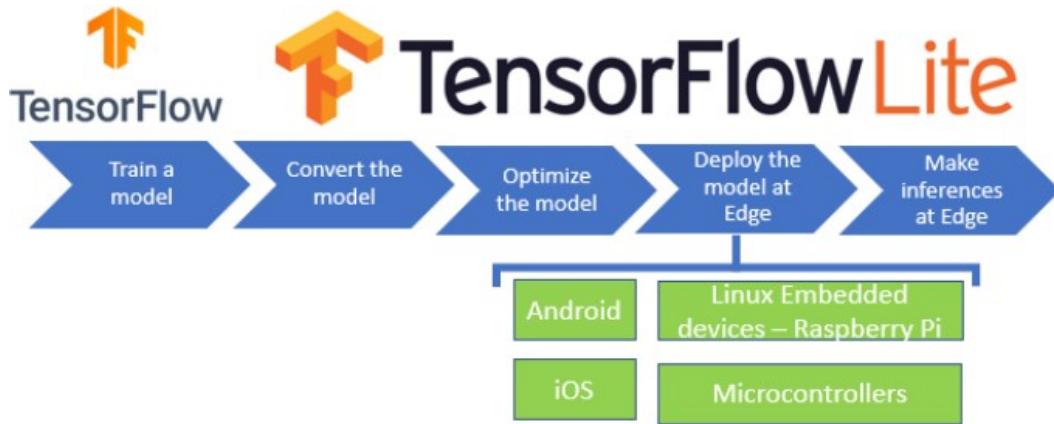


Figura 21: Suite de TFLite