Capítulo 1

Introducción y conceptos básicos

1.1. Introducción

La Visión Artificial es una disciplina científica que se ocupa de captación, el análisis y la comprensión de imágenes y vídeos. Se relaciona estrechamente con otros campos científicos y tecnológicos tales como la inteligencia artificial, el aprendizaje máquina, el procesamiento por computador, el procesamiento de señal, gráficos por computador, robótica e incluso con psicología y neurociencias.

Con frecuencia el término Visión Artificial se utiliza para referirse a cuatro disciplinas íntimamente relacionadas: Visión por Computador, Visión Máquina, Tratamiento de Imágenes y Visión Artificial propiamente dicha. La Visión por Computador (*Computer Vision* en la literatura anglosajona) se ocupa de la utilización de los computadores para alcanzar un conocimiento de las imágenes adecuado para desarrollar tareas similares a las que realizamos los humanos por medio de nuestro sistema visual. La Visión Máquina (*Machine Vision*) se orienta específicamente a la automatización de tareas típicamente industriales por medio de imágenes, como la inspección, el control de procesos y el guiado de robots. El Tratamiento de Imágenes (*Image Processing*) se refiere genéricamente al procesamiento de las mismas por medio de computadores de cara su mejora y análisis. Por último, la Visión Artificial propiamente dicha se relaciona con reproducir el comportamiento del sistema visual humano. Por supuesto estas cuatro disciplinas comparten algoritmos y tecnologías, por lo que es habitual referirse al conjunto de ellas genéricamente como Visión por Computador o, como lo haremos en este libro, Visión Artificial.

La Visión Artificial, entendida en este sentido amplio, comienza su andadura a mediados de los años 60, cuando por primera vez se aborda la conexión de una cámara a un computador (1966, M. Minsky). Surgen también los primeros algoritmos para la detección y seguimiento de bordes. En la década de los 70 asistimos al desarrollo de numerosos algoritmos de Visión Artificial que

continúan vigentes hoy en día: mejora de la extracción de regiones y bordes, reconocimiento de formas geométricas, flujo óptico, estimación del movimiento. Los 80 vienen marcados por una creciente formalización matemática, lo que convierte a la Visión Artificial en una disciplina científica madura. Con esto, y también gracias al desarrollo de los computadores y la microelectrónica, queda plenamente establecido el *pipeline* clásico de Visión Artificial. Existen distintas formulaciones de este *pipeline* que básicamente cubren, de una forma u otra, las siguientes etapas:

- Adquisición de las imágenes. Obtención de la información visual de la escena por medio de cámaras que proporcionan imágenes individuales o secuencias de imágenes (vídeo).
- Filtrado. Se trata de una etapa fundamental en la que se cubren múltiples aspectos, desde la reducción del ruido de las imágenes hasta el resaltado de características de interés como líneas, bordes de regiones, regiones de distinta morfología o características de más alto nivel. Esta etapa puede ser determinante de cara al éxito de los tratamientos posteriores.
- Segmentación. Consiste en el particionado de la imagen en conjuntos de píxeles o *segmentos* (segmentos de información; no confundir con segmentos lineales) que comparten algunas propiedades. La etapa previa filtrado determina la mejor forma de operar, sobre la base del análisis de propiedades tales como los niveles de gris, color o textura, y sus variaciones a lo largo de la imagen, en muchos casos combinado con criterios de proximidad en la imagen.
- Extracción de características. Cálculo de descriptores geométricos y topológicos de las regiones encontradas tras la segmentación, tales como perímetro, área, compacidad, momentos de inercia de distinto orden, número de agujeros y de regiones conexas, etc.
- Reconocimiento de los objetos presentes en la escena. En esta etapa se emplean desde técnicas sencillas de umbralización directa de descriptores hasta otras más elaboradas, del ámbito de la Inteligencia Artificial.
- Interpretación de la escena. Se analizan objetos y sus interrelaciones para dar sentido a la escena en el contexto de la aplicación deseada. De nuevo las técnicas de Inteligencia Artificial pueden jugar un papel fundamental en esta etapa final.

Una aproximación alternativa al problema, utilizada desde antiguo en Visión Máquina y que obvia algunas de estas etapas, es lo que se conoce como comparación con plantillas o *Template Matching*. Consiste en recorrer la imagen en busca de regiones parecidas a una pequeña plantilla que contiene una imagen del elemento buscado. Además, si se adopta la correlación normalizada como medida de similitud, se consigue cierta insensibilidad ante variaciones de la iluminación. Esta técnica se emplea con frecuencia en visión máquina, pero su

robustez es limitada cuando nos enfrentamos a variabilidad en el aspecto visual de los objetos.

Por otra parte, a finales de los 90 se desarrollan los conocidos algoritmos *Scale-Invariant Feature Transform* (SIFT) y Speeded-Up Robust Features (SURF), y sus distintas variantes. Estos algoritmos permiten detectar y caracterizar puntos relevantes de los objetos de forma robusta frente a cambios de iluminación, rotación y escala. Por ello, resultan de interés en tareas como reconocimiento de objetos, incluso en presencia de oclusiones, *stitching* o pegado de imágenes, *matching* o emparejamiento en visión estéreo, *tracking* o seguimiento visual, etc. En 2001 se produce otro avance significativo de la mano del algoritmo de Viola-Jones. Se trata de un algoritmo capaz de detectar rostros u otros objetos en tiempo real, con un coste computacional asumible incluso para dispositivos con prestaciones modestas.

Más recientemente, en los 2000, asistimos al resurgimiento de las redes neuronales convolucionales. La idea original data de los años 80, pero ha sido necesario esperar casi 30 años para verla implementada de manera efectiva gracias a los modernos procesadores gráficos (GPUs). La red AlexNet, de finales de 2012, representa un hito destacable por ser una de las primeras en sacar provecho de este tipo de procesadores. A raíz de AlexNet hemos asistido a un desarrollo vertiginoso del campo de las redes convolucionales, donde nuevas arquitecturas, como los modelos ViT o las redes más avanzadas de convolución (CoAtNet, ConvNeXt, etc.) han visto la luz. A lo largo del libro profundizaremos en alguno de estos modelos.

El presente libro se inscribe en este contexto. No pretende profundizar en el *pipeline* clásico o en las técnicas de los años 90, para cuya descripción ya existen excelentes textos. En lugar de ello, se centra en cómo utilizar las redes neuronales para resolver distintos problemas de visión artificial. Además, se ha adoptado un enfoque esencialmente práctico con el fin de que sirva de guía de referencia a la que podamos acudir para resolver los problemas que nos encontremos.

1.2. Imagen y vídeo

En primera aproximación, una imagen digital es una composición de elementos, denominados píxeles, habitualmente organizados en forma de matriz o línea, cuyos valores numéricos representan la luminosidad o el color de los puntos de una imagen. En los sistemas de Visión, esta imagen digital proviene de la proyección óptica de la escena sobre un dispositivo sensor, normalmente plano y de naturaleza discreta, con sus elementos fotosensibles distribuidos de manera regular. El sensor muestrea espacialmente la información óptica y la convierte en señales eléctricas, que son amplificadas y cuantificadas en forma de niveles numéricos que se transfieren al computador. Por simplicidad, denominaremos genéricamente *cámara* al conjunto formado por la óptica empleada para la proyección, el sensor y la electrónica correspondiente.

Los niveles que proporciona la cámara son una forma de representar la intensidad que debería adoptar una fuente de luz, o varias en el caso de color,

para producir una sensación visual similar a la imagen óptica de partida. Estos niveles se representan usualmente mediante números enteros sin signo de 8 bits, dado que una diferencia de $1/2^8$ es apenas discernible para el ojo humano en el rango de luminosidades y colores que manejamos habitualmente. No obstante, por supuesto es posible una cuantización más precisa. Por ejemplo, resulta cada vez más frecuente recurrir a 10, 12, 14 o incluso 16 bits (por canal cromático, en su caso). Además, el procesamiento ulterior de las imágenes puede dar como resultado valores que se representan más adecuadamente mediante otros tipos de datos: enteros de otros tamaños, con o sin signo, valores lógicos o números reales. Por todo ello, en general no restringiremos el tipo de datos que puede contener una imagen.

En cuanto a la señal de vídeo, se puede entender como una secuencia de imágenes adquiridas en instantes de tiempo sucesivos. En el presente texto supondremos que cada imagen de la secuencia contiene toda la información correspondiente a un cierto instante, con independencia del tipo de obturador empleado por la cámara (común o rotativo) y del modo de captura (progresivo o entrelazado). Además, asumiremos que el tiempo transcurrido entre dos imágenes consecutivas es siempre el mismo.

Por otra parte, para la transmisión y el almacenamiento de imágenes y vídeo se recurre con frecuencia a distintos contenedores y formatos. Estos vienen caracterizados por la organización de los valores de luminosidad o color que contienen, los metadatos que los acompañan y los algoritmos utilizados para la codificación de la información, con o sin pérdida. Algunos formatos de imagen populares son BMP, PNG , TIFF, EXIF y JPEG. Algunos formatos de vídeo comunes son AVI, MPEG, MOV y MP4. Una amplia discusión de estos y otros formatos puede encontrarse en textos especializados. En todo caso, de cara al procesamiento asumiremos que tanto las imágenes como las secuencias de vídeo han sido adecuadamente descodificadas hasta su representación en forma de matriz o secuencia temporal de matrices.

1.3. El color

Tal como hemos indicado, las cámaras proporcionan, por cada píxel, los niveles de intensidad que deberían adoptar las luces de un cierto dispositivo de visualización para producir en el observador humano una sensación visual análoga a la imagen capturada. Además, la mayoría de los sistemas actuales contempla luces de tres *colores primarios*, rojo, verde y azul, que suponen un compromiso adecuado entre complejidad de las cámaras y los dispositivos de visualización, y la gama de colores que permiten reproducir.

La elección de tres primarios descansa sobre el hecho de que el observador humano estándar, o promedio, utiliza tres tipos de células para percibir el color. Se trata de los denominados *conos*, sensibles a luz de longitudes de onda cortas, medias y largas, según su tipo. Un cuarto tipo de células, los *bastones*, perciben la cantidad total de luz. Las señales eléctricas que generan estos cuatro tipos de células son interpretadas por el cerebro como luz y color.

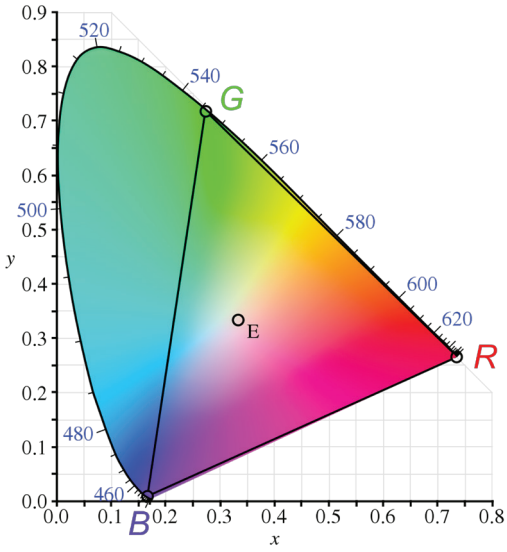


Figura 1.1: Diagrama cromático CIE 1931. Solo se visualizan correctamente los colores reproducibles a partir de las (pocas) tintas empleadas en la impresión de este documento. (Adaptado de Wikipedia, CIE 1931 color space).

En los años 30 se realizaron una serie de experimentos encaminados a relacionar cuantitativamente la radiación electromagnética con la percepción visual. Fruto de ello se propuso el denominado espacio color CIE 1931 XYZ, que asigna a cada color 3 números positivos, de forma unívoca. Estos números pueden entenderse como las coordenadas de cada color en un espacio cromático tridimensional denominado XYZ. Por conveniencia, el espacio XYZ fue definido de forma que Y representa la luminosidad, es decir la cantidad total de luz, mientras que X y Z se relacionan con el color propiamente dicho.

Los *diagramas cromáticos* son una forma adecuada de manejar este espacio cromático (véase la Figura 1.1). Se trata de secciones planas del espacio XYZ en las que se representa la gama de colores que puede percibir el observador humano promedio, para un determinado nivel de luminosidad (un cierto valor de Y). Los *colores espectrales*, es decir correspondientes a las radiaciones monocromáticas (de una única longitud de onda), se sitúan en el contorno de la gama, mientras que la denominada *línea de púrpuras* delimita la región por la parte inferior. Por construcción, todos los colores de la gama pueden generarse combinando luces de los (infinitos) colores espectrales, balanceadas en función del inverso de su distancia al color deseado medida sobre el diagrama. La característica forma de *lengua* o *herradura* de estos diagramas deriva de la fuerte correlación existente entre las respuestas espectrales de los distintos tipos de conos.

En general, un conjunto arbitrario de luces permite generar todos los colores presentes en el interior del polígono que circunscriben dentro del diagrama cromático (pero no los presentes en el exterior). A modo de ejemplo, el espacio

color estándar *CIE 1931 RGB* especifica tres luces monocromáticas de longitudes de onda 700 nm (rojo, marcado R en la Figura 1.1), 546.1 nm (verde, G) y 435.8 nm (azul, B) que, adecuadamente balanceadas, permiten producir todos los colores del interior del triángulo que delimitan. Obviamente no todos los colores pueden reproducirse por este medio. Por ejemplo, se pierden todos los verdeazulados situados a la izquierda de la línea BG y los púrpuras bajo la línea RB. Por lo general, la situación empeora en el caso de utilizar otra combinación de primarios monocromáticos, o bien de luces no monocromáticas como es el caso de muchos monitores o dispositivos de proyección. Desde luego, el uso de un mayor número de primarios, convenientemente seleccionados, permite ampliar la gama de colores reproducibles. Por ejemplo, algunos dispositivos de visualización trabajan con cuatro primarios (rojo, verde, azul y amarillo). Sin embargo, su uso suele resultar antieconómico, en especial si se tienen en cuenta las implicaciones respecto al diseño de cámaras.

En todo caso, lógicamente los niveles de color proporcionados por la cámara deben adaptarse a los primarios que utilice cada dispositivo de visualización. Para ello debe se requiere una transformación que, para tres primarios, adopta la forma

$$\begin{pmatrix} R' \\ G' \\ B' \end{pmatrix} = \mathbf{M} \begin{pmatrix} R \\ G \\ B \end{pmatrix} \ . \tag{1.1}$$

La naturaleza lineal de la transformación deriva del hecho de que, en definitiva, equivale a un cambio de ejes en el espacio XYZ. Adicionalmente se pueden incorporar otras transformaciones, lineales o no, como la conocida *corrección gamma* que compensa las no linealidades de las cámaras y los dispositivos de visualización, así como del propio sistema visual humano.

Existen formas alternativas de especificar la información cromática, no tanto orientadas a su reproducción mediante luces R, G y B, como a otros fines. Entre ellas destaca el sistema HSV (*hue* o matiz, *saturation* o saturación, *value* o valor), también conocido como HSB (*hue*, *saturation*, *brightness* o brillo). Este sistema se encuentra estrechamente relacionado con la forma en que los humanos percibimos y describimos el color. En este sistema, el matiz puede entenderse como una tonalidad específica dentro del espectro visible, descrita como su posición en un determinado *círculo cromático*. Se especifica habitualmente como un ángulo en el rango $[0, 360^{\text{o}})$. La saturación corresponde a la *cantidad* de ese color y se especifica típicamente en un rango de 0 %, para el gris, a 100 % para los colores puros. Finalmente, el brillo corresponde intuitivamente a la *cantidad de luz* que contiene el color, desde 0 % para negro hasta 100 % para blanco. Estos tres rangos se discretizan convenientemente, muchas veces en forma de enteros cortos o números reales, para su manejo eficiente por el computador. La transformación entre los espacios RGB y HSV es bien conocida, aunque no lineal.

Los espacios descritos, RGB y HSV, resultan adecuados para tareas de Visión. No obstante, existen espacios de color mejor adaptados a otros ámbitos.

Por ejemplo, el sistema HSL/HSI (*h*ue, *saturation*, *lightness* o *intensity*) es similar al HSV, pero se adapta a la reproducción de los colores mediante mezcla de pinturas en lugar de luces. Otros como el CIEL*a*b* y el propio CIEXYZ se usan en colorimetría por cuanto permiten expresar mediante números positivos todos los colores que puede percibir el observador humano, con independencia del dispositivo de visualización utilizado. Espacios como el CMY(K) (*cyan*, *magenta*, *yellow* (*black*)) se adecúan a la reproducción mediante tintas que reflejan estos tres colores primarios (y absorben, por tanto, todos los demás). Existen también sistemas más orientados a fotografía digital como el YCbCr. Una descripción más exhaustiva escapa al ámbito de este libro.

1.4. Operaciones básicas con imágenes.

Una vez asumido que las imágenes se representan en forma de matrices, se pueden contemplar una amplia variedad de operaciones útiles. Además, estas matrices pueden tener naturaleza meramente bidimensional, por ejemplo en el caso de imágenes de luminosidad, o bien tridimensional, como en el caso de imágenes con varios *canales* o *planos cromáticos*. En el primer caso se suele hablar de imágenes monocanal y en el segundo de imágenes multicanal. En una primera aproximación, cada canal de una imagen se puede procesar independientemente del resto. En otros casos, los resultados obtenidos en los distintos canales se combinan entre sí para dar lugar a una imagen con un número distinto de canales. A continuación, citamos simplemente a modo ilustrativo algunas operaciones típicas. Para una discusión más sistemática y extensa nos remitimos a los cualquiera de los excelentes textos que ya existen sobre el tema.

- Suma de imágenes. Se utiliza por ejemplo para promediar los niveles de varias imágenes sucesivas en el tiempo, con el fin de reducir el ruido.
- Resta de imágenes. Es de uso común en tareas como detectar movimiento en secuencias de imágenes o modelar el fondo estático de una escena con relación a los objetos móviles (aunque para esto existen técnicas más elaboradas, basadas en la caracterización estadística de la variación de los niveles de cada píxel a lo largo del tiempo).
- Multiplicación de imágenes elemento a elemento. Se destina habitualmente a poner a nivel 0 ciertos puntos de una imagen, según un patrón expresado en forma de imagen o *máscara* de naturaleza lógica.
- Operaciones con escalares. La multiplicación con escalares y la suma o resta de un cierto valor a cada punto son útiles para modificar el rango dinámico de los niveles de la imagen, por ejemplo para el desplazamiento y estiramiento del histograma. Combinadas con suma de imágenes se usan para mezclado o *blending* de imágenes.
- Corrección *gamma*. Corrección logarítmica que se utiliza para aumentar el contraste en las zonas de baja luminosidad de las imágenes para adaptarlo a las características del sistema visual humano.

- Umbralización. Operación para el etiquetado de los píxeles de la imagen por comparación de su valor contra un cierto valor umbral. El umbral puede fijarse *a priori*, calcularse a partir de un análisis global de la imagen u obtenerse a partir del análisis de los píxeles en una cierta vecindad. En este último caso se habla de *umbralización adaptativa*. La idea se generaliza fácilmente a más de un umbral y de dos niveles de salida, en lo que se conoce como umbralización multinivel.
- Operaciones lógicas sobre imágenes binarias. Se emplean por ejemplo para la mejora del resultado tras una umbralización.

A efectos prácticos es importante tener en cuenta que, tal como ya ha sido mencionado, el resultado de las operaciones con imágenes pueden ser matrices de valores lógicos, enteros de distinto tamaño, números reales etc. Esto tiene implicaciones obvias respecto al tipo de variable más adecuado para almacenar el resultado de la operación. Además, para visualizar el resultado en forma de imagen es preciso traducir los valores a niveles manejables por el dispositivo de visualización: típicamente tres enteros positivos, por ejemplo de un byte, para representar las intensidades de rojo, verde y azul. Esta traducción se realiza mediante tablas de consulta (*Look up tables*) asociadas al dispositivo. Es importante que estas tablas se configuren adecuadamente en cada caso, con el fin de evitar una mala interpretación del resultado de las operaciones.

1.5. El producto de convolución

Además de las operaciones básicas, el denominado *producto de convolución* o simplemente *convolución* constituye una operación de enorme interés debido a sus excelentes propiedades matemáticas y su versatilidad. Se suele expresar matemáticamente como una correlación,

$$J(x,y) = W(x,y) \circledast I(x,y) = \sum_{s=-M/2}^{M/2} \sum_{t=-N/2}^{N/2} W(s,t)\ I(x+s, y+t)\ . \quad (1.2)$$

Aquí, I es la imagen de entrada o fuente, J es la imagen de salida o destino y W una matriz N filas x M columnas, habitualmente de pequeño tamaño con relación a imagen, que denominaremos genéricamente *filtro* o, más específicamente según el contexto, *núcleo*, *kernel* o *máscara* de la convolución. Es importante remarcar que esta ecuación corresponde a una correlación, y no a una convolución. La diferencia entre ambas radica en que esta última conlleva un giro de π radianes del núcleo. Sin embargo, en Visión resulta frecuente utilizar el término *convolución* como sinónimo de correlación, en particular cuando los valores del núcleo se obtienen por aprendizaje, y así lo haremos en este libro salvo que se indique lo contrario.

La convolución, además de ser lineal, tiene las propiedades conmutativa y asociativa. Esto permite, entre otras cosas, calcular la convolución de una imagen

con una secuencia de filtros como su convolución con un único filtro, resultante de convolucionar aquellos entre sí. De esta manera se simplifica la manipulación matemática y se reduce el coste computacional.

$$W_2(x,y) \circledast (W_1(x,y) \circledast I(x,y)) = (W_2(x,y) \circledast W_1(x,y)) \circledast I(x,y)\ . \quad (1.3)$$

El número de operaciones necesario para completar una convolución crece cuadráticamente con el tamaño del filtro. Afortunadamente muchos filtros útiles son *separables*, esto es, resultan de convolucionar dos filtros monodimensionales entre sí. Aquí la propiedad asociativa permite convolucionar la imagen con uno de ellos, y el resultado con el otro, lo que conlleva una importante reducción del número de operaciones.

Por lo demás, los principales aspectos a definir de cara a realizar una convolución son los siguientes:

- Rellenado (o *padding*).
- Paso (o *stride*).
- Los pesos y el tamaño del núcleo.

El rellenado deriva de que los cálculos no pueden ser completados en los puntos cuya distancia a los límites de la imagen sea inferior a la mitad del tamaño del filtro en la dimensión correspondiente. En este caso existen dos alternativas. La primera consiste en asumir que la imagen destino tendrá un tamaño inferior al de la imagen fuente, con la consecuente reducción de las necesidades de cómputo y almacenamiento. La segunda alternativa consiste en rellenar, por ejemplo con ceros, los datos faltantes alrededor de la imagen. De esta forma se pueden completar los cálculos, aun asumiendo que el resultado será erróneo en los puntos de la periferia. La imagen de salida tendrá el mismo tamaño que la de entrada lo que simplifica la programación y la electrónica.

En cuanto al paso, es la distancia que avanza el núcleo entre una posición y la siguiente. La definición formal dada por la ecuación 1.2 contempla que el núcleo recorra la imagen de entrada avanzando una posición cada vez, esto es, con un *paso* de 1. Otras veces se prefiere un paso mayor, por ejemplo del mismo tamaño que el núcleo, para acelerar los cálculos y reducir el tamaño de la imagen de salida.

Por lo demás, los coeficientes o pesos del filtro, junto el tamaño de este, determinan el tipo de filtrado que se obtiene con la convolución. Así, existen filtros basa bajos, de realzado, de derivación y un largo etcétera. Generalmente, los filtros cuyos pesos suman más de 1 tienden a aclarar la imagen y los que suman menos tienden a oscurecerla. Cuando los pesos suman 1, la luminosidad general no se ve afectada. Para las operaciones de derivada primera y segunda es habitual recurrir a filtros cuyos pesos suman cero.

El filtro de la media de $3x3$ es un ejemplo clásico de filtro de convolución,

$$\frac{1}{9}\begin{pmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{pmatrix} . \tag{1.4}$$

Se utiliza para reducir el ruido en las imágenes, a costa de disminuir el contraste. La intensidad del filtrado puede aumentarse incrementando el tamaño del filtro. Además, es frecuente recurrir a filtros de forma cuadrada, al menos cuando la relación de aspecto de la distancia entre píxeles está próxima a 1:1. Existen alternativas para el filtrado de ruido que no son de convolución, como el popular filtro de la mediana que se comporta como un excelente pasa bajos. Sin embargo, conlleva un elevado coste computacional y presenta peores propiedades matemáticas que el filtro de la media, por lo que su uso está menos extendido.

Volviendo a la convolución, tradicionalmente se han venido utilizando núcleos de pequeño tamaño para ciertas operaciones básicas, el filtrado de ruido o el cálculo de gradientes de luminosidad en la imagen. El filtro de Prewit es un buen ejemplo. Utiliza dos núcleos que se obtienen convolucionando el filtro de la media con las aproximaciones centrales de la derivada en las direcciones horizontal y vertical, $(-1, 0, 1)$ y $(-1, 0, 1)^t$:

$$\begin{pmatrix}-1 & 0 & 1\\ -1 & 0 & 1\\ -1 & 0 & 1\end{pmatrix} \quad , \quad \begin{pmatrix}-1 & -1 & -1\\ 0 & 0 & 0\\ 1 & 1 & 1\end{pmatrix} . \tag{1.5}$$

Otro filtro popular es el de Sobel, que utiliza dos núcleos de convolución cuyos pesos resultan de añadir, a las componentes horizontal y vertical de la derivada en cada punto, la contribución de las derivadas a 45^{o} proyectadas sobre la dirección en cuestión:

$$\begin{pmatrix}-1 & 0 & 1\\ -2 & 0 & 2\\ -1 & 0 & 1\end{pmatrix} \quad , \quad \begin{pmatrix}-1 & -2 & -1\\ 0 & 0 & 0\\ 1 & 2 & 1\end{pmatrix} . \tag{1.6}$$

La Figura 1.2 muestra un ejemplo de imagen de prueba, en este caso de un solo canal cromático, y la Figura 1.3 el resultado de aplicar el citado filtro. Los valores mínimos (más negativos) se muestran en negro y los máximos en blanco. El resto se visualizan en tonalidades intermedias de gris.

Los filtros de Prewitt y Sobel permiten obtener fácilmente el módulo del vector gradiente y su dirección en cada punto de la imagen. Los máximos locales del módulo a lo largo de dicha dirección son buenos candidatos a puntos de borde de los objetos presentes en la imagen. El también popular filtro de Canny proporciona una conectividad mejorada entre estos puntos a costa de un mayor esfuerzo computacional, sobre la base de conectar puntos de gradiente elevado a través puntos con gradiente más reducido.

Filtros similares se pueden usar para resaltar bordes de regiones en direcciones específicas de la imagen, como por ejemplo:

Figura 1.2: Imagen de prueba.

(a) Componente horizontal.

(b) Componente vertical.

Figura 1.3: Filtrados de Sobel de la imagen de prueba.

$$\begin{pmatrix} -1 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \quad y \quad \begin{pmatrix} 0 & 1 & 1 \\ -1 & 0 & 1 \\ -1 & -1 & 0 \end{pmatrix} . \tag{1.7}$$

Los filtros basados en derivada segunda constituyen un interesante complemento a los anteriores. El operador *divergencia del gradiente*, también conocido como *laplaciana*, permite detectar zonas de la imagen donde la luminosidad alcanza máximos o mínimos locales. Algunas discretizaciones comunes de este operador son

$$\begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix} \quad y \quad \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix} . \tag{1.8}$$

Los pasos por cero de la laplaciana de una imagen en los puntos donde el gradiente es significativo sirven para detectar de forma precisa los puntos de borde. Por su parte los máximos y mínimos corresponden a puntos pertenecientes a líneas de mayor o menor nivel que el entorno, respectivamente.

Los filtros descritos hasta aquí se utilizan en ciertas aplicaciones prácticas por cuanto representan un cierto compromiso entre complejidad, coste computacional y calidad de los resultados. Sin embargo, existen alternativas ventajosas para tratar con las distintas frecuencias espaciales que suelen aparecer en las imágenes. Muchas de estas alternativas se basan en el denominado filtro *gaussiano*, cuyos pesos siguen una distribución gaussiana bidimensional caracterizada por su desviación típica σ,

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} . \tag{1.9}$$

El filtro gaussiamo proporciona un suavizado más natural que el de la media. Además, el grado de suavizado se controla de forma precisa a través del parámetro σ. Es también separable, como el de la media, pero presenta unas excelentes propiedades matemáticas que facilitan su manipulación analítica. Por otra parte, hay que tener en cuenta que se trata de un filtro de respuesta impulsional infinita (*Infinite Impulse Response*, IIR), es decir, tiene un número infinito de coeficientes no nulos. Sin embargo, a efectos prácticos se puede aproximar satisfactoriamente por un filtro de respuesta impulsional finita (*Finite Impulse Response*, FIR) de radio 3 σ. Un ejemplo de filtro gaussiano, para $\sigma = 1$, es

$$\begin{pmatrix} 0,00 & 0,01 & 0,02 & 0,01 & 0,00 \\ 0,01 & 0,06 & 0,10 & 0,06 & 0,01 \\ 0,02 & 0,10 & 0,16 & 0,10 & 0,02 \\ 0,01 & 0,06 & 0,10 & 0,06 & 0,01 \\ 0,00 & 0,01 & 0,02 & 0,01 & 0,00 \end{pmatrix} . \tag{1.10}$$

La figura figura 1.4 representa de forma visual la distribución espacial de estos pesos. Este tipo de representación será utilizada con frecuencia en el presente texto. Además, salvo donde se indique lo contrario, asumiremos que los

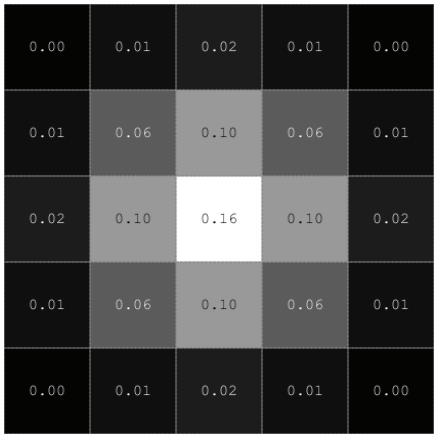


Figura 1.4: Filtro gaussiano de $\sigma = 1$.

niveles de visualización se han ajustado de forma lineal entre negro para el valor mínimo del filtro (aquí, 0,00), y blanco para el valor máximo (aquí, 0,16).

La figura 1.5 muestra el resultado de filtrar una imagen con filtros gaussianos de distinta σ. En este caso se trata de una imagen con tres canales cromáticos. Cada uno ha sido procesado por separado, si bien existen otras alternativas que discutiremos más adelante.

El filtro gaussiano se utiliza con frecuencia en combinación con operadores de derivada primera y segunda. La Figura 1.6 muestra un ejemplo de filtro tipo gradiente de gaussiana, que permite resaltar los bordes de regiones más claras o bien más obscuras que el fondo.

La Figura 1.7 muestra un ejemplo de aplicación de este filtro. En comparación con el filtro de Sobel, los bordes se encuentran mejor definidos. Además, el parámetro σ permite ajustar el comportamiento del filtro.

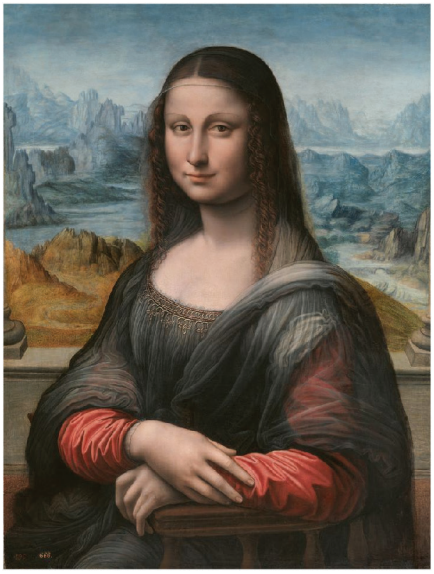
Por supuesto estos filtros pueden ser escalados en forma anisotrópica y rotados, de forma que presenten una mejor respuesta a tipos específicos de bordes, orientados en distintas direcciones. La Figura 1.8 muestra algunos ejemplos.

Los rasgos lineales, es decir grupos de píxeles con diferente valor que los del entorno, alineados en cierta dirección de la imagen, pueden ser resaltados mediante filtros de derivada segunda. Una primera alternativa consiste aplicar un nuevo gradiente a cada componente del gradiente obtenido mediante un filtro de primer orden. La Figura 1.9 muestra, en la parte superior, dos filtros de este tipo, orientados a lo largo de las dos direcciones principales de la imagen; y en la parte inferior el gradiente cruzado, de interés para ciertos algoritmos.

El resultado de aplicar los dos primeros filtros mostrados en esta Figura 1.9 a la imagen de prueba se muestra en la Figura 1.10.

Como en el caso de los filtros de derivada primera, estos filtros también pueden ser escalados y rotados, como en la Figura 1.11, con el fin de resaltar líneas de cierto espesor y con una orientación específica

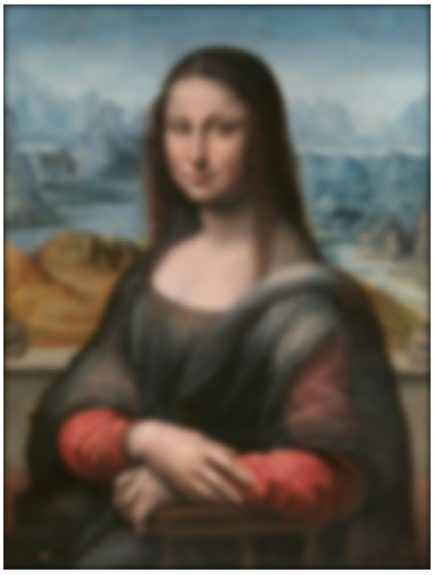
Una segunda alternativa de segundo orden consiste en calcular la divergencia



(a) Imagen original, sin aplicar filtrado.

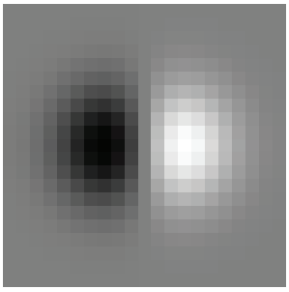
(b) Filtrado con una gaussiana de $\sigma = 2$.

(c) Filtrado con una gaussiana de $\sigma = 4$.

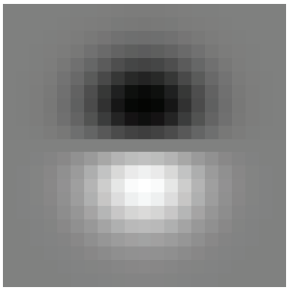


(d) Filtrado con una gaussiana de $\sigma = 6$.

Figura 1.5: Imagen de 954 x 720 píxeles filtrada con filtros gaussianos de distinta σ.



(a) Componente horizontal.



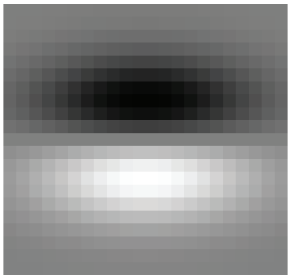
(b) Componente vertical.

Figura 1.6: Filtro de gradiente de gaussiana de $\sigma = 3$.

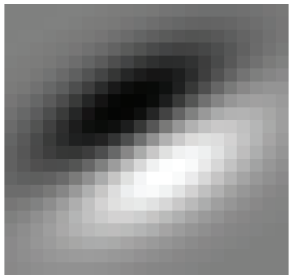
(a) Componente horizontal.

(b) Componente vertical.

Figura 1.7: Resultado de un filtrado de gradiente de gaussiana de $\sigma = 3$.

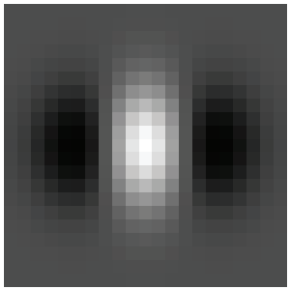


(a) Gradiente vertical escalado anisotrópicamente.

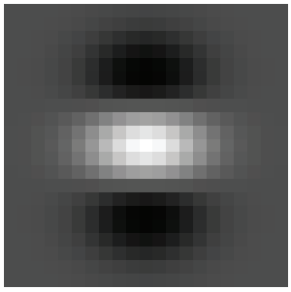


(b) El mismo gradiente, rotado.

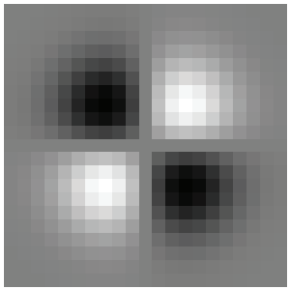
Figura 1.8: Filtro de gradiente de gaussiana.



(a) Gradiente horizontal de segundo orden.



(b) Gradiente vertical de segundo orden.



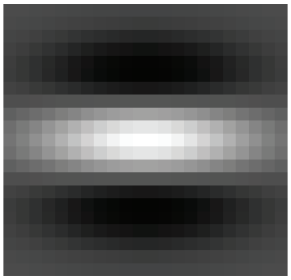
(c) Gradiente cruzado.

Figura 1.9: Filtros de gradiente de un gradiente de gaussiana: (a) gradiente horizontal de la componente horizontal, (b) gradiente vertical de la componente vertical, y (c) un gradiente cruzado (igual al otro por la conmutatividad de la convolución).

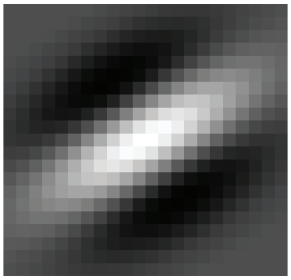
(a) Gradiente horizontal de segundo orden.

(b) Gradiente vertical de segundo orden.

Figura 1.10: Resultados de un gradiente de gradiente de gaussiana.

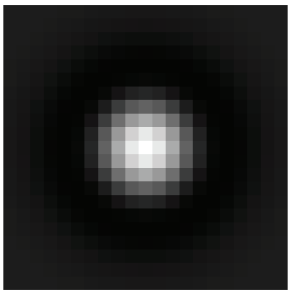


(a) Gradiente vertical de segundo orden, escalado.

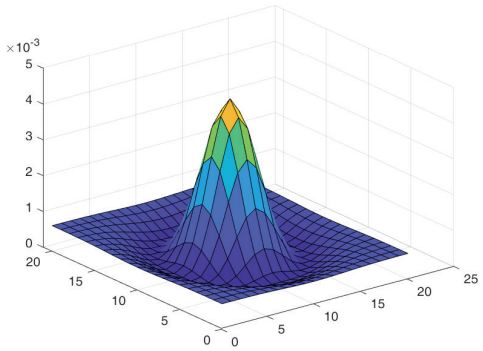


(b) El mismo gradiente, rotado.

Figura 1.11: Filtro de gradiente de un gradiente de gaussiana, escalado anisotrópicamente y rotado.



(a) Laplaciana de gaussiana.



(b) Visualización gráfica de los pesos.

Figura 1.12: Laplaciana de una gaussiana de $\sigma = 3$.

del gradiente de gaussiana, más conocida como *laplaciana de gaussiana* (LOG). La Figura 1.12 muestra un ejemplo de este filtro junto con su representación en forma de gráfico tridimensional. En esta se aprecia la distribución espacial de pesos, descrita habitualmente como de *sombrero mejicano*.

El filtro LOG es más popular que el gradiente de gradiente de gaussiana por resultar más simple y proporcionar resultados comparables, como se ve en la Figura 1.13, si bien se pierde la información de direccionalidad.

Por otra parte, una propiedad interesante de la LOG es que, debido a su mencionada forma de sombrero mejicano, presenta una fuerte respuesta a *blobs*, entendidos como grupos de píxeles próximos, más oscuros o claros que el entorno. El parámetro σ controla el tamaño de los *blobs* que se detectarán. En concreto, la respuesta máxima se obtiene para *blobs* con un radio en torno a $\sqrt{2}\sigma$. La Figura 1.14 muestra un ejemplo. Además, el filtro LOG también puede ser escalado anisotrópicamente y rotado, como en el caso de los filtros discutidos anteriormente, para detectar blobs de distinto tamaño, forma y orientación.

Por otra parte, una LOG se puede aproximar por la Diferencia entre dos Gaussianas (DOG) de distinta σ, o incluso por la diferencia entre una imagen y su filtrado gaussiano. De hecho, las células ganglionares de la retina realizan esta operación para favorecer la detección de los objetos. Así operan también ciertos algoritmos de procesamiento de imágenes como los populares SIFT y SURF, para detectar y caracterizar zonas de interés de la imagen a distintas escalas, así como ciertos algoritmos para la construcción de pirámides multiescala utilizados en trasmisión de imágenes y reconocimiento de objetos.

Sobre ideas análogas a las anteriores se construyen otros tipos de filtros, como filtros de derivada de orden mayor o, por ejemplo, los filtros de Gabor. Estos se basan en senoides de frecuencia y fase ajustable, moduladas por gaussianas centradas en las zonas de la imagen bajo análisis, como ilustra la Figura 1.15.

En todo caso, sí es importante notar que, más allá de estos operadores generales, y teniendo en cuenta que en definitiva estamos calculando correlaciones, se pueden diseñar filtros específicos para detectar rasgos particulares. En la Figura

Figura 1.13: Resultado de aplicar una laplaciana de gaussiana de $\sigma = 3$.

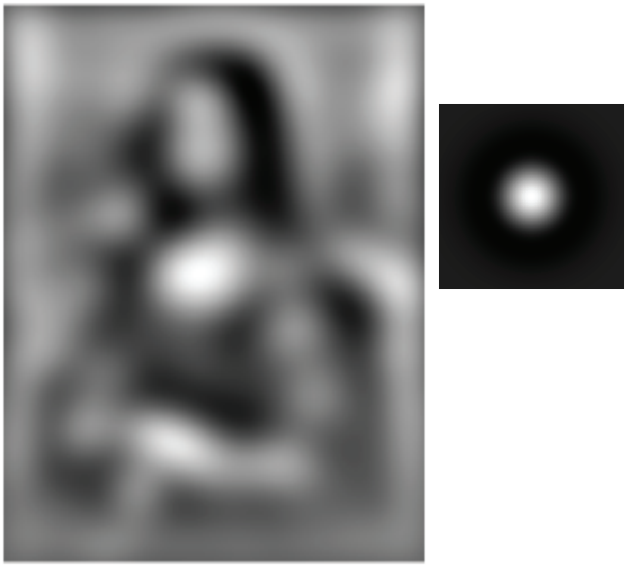


Figura 1.14: Resultado de aplicar una laplaciana de una gaussiana de $\sigma = 45$ (mostrada a la derecha).

Figura 1.15: Un ejemplo de filtro de Gabor.
Fuente: Chabacano, CC BY-SA 3.0, via Wikimedia Commons

1.16 se muestran algunos ejemplos de filtros calculados por la red convolucional VGG-16 [1] en distintas capas del modelo. Estos filtros se obtienen durante el entrenamiento y permiten la detección de líneas, bordes de regiones, círculos, patrones repetitivos, etc. en distintas orientaciones del plano de la imagen.

El problema a la hora de aplicar todos estos filtros en la práctica consiste en dar con el filtro adecuado en cada caso y ajustar sus pesos convenientemente. Esto puede abordarse sobre la base de hipótesis formuladas *a priori* acerca de los modelos matemáticos de los rasgos que se desea resaltar en las imágenes. Sin embargo, habitualmente se precisará un ajuste posterior por ensayo y error. En todo caso, por lo general estos filtros proveen un mecanismo para resaltar primitivas de bajo nivel, pero se requiere un esfuerzo adicional considerable para detectar formas más complejas. Esto puede abordarse por agrupamiento de las primitivas de bajo nivel, pero la forma de hacerlo dista de ser evidente. Las redes neuronales convolucionales, que constituyen el objeto de gran parte del presente libro, representan una solución adecuada a ambos problemas: proveen un mecanismo para el aprendizaje de los filtros y sus pesos, y permiten agrupar primitivas de bajo nivel sobre la base de un esquema piramidal de resolución decreciente.

Por otra parte, existen aproximaciones alternativas a las planteadas en esta sección tanto sobre la base de filtros con pesos predefinidos como aprendidos. Un ejemplo notable son las denominadas *características de Haar*, núcleos rectangulares estructurados como agrupación de núcleos más pequeños con pesos 1 y -1. El número y la forma de las características que resultan útiles en cada zona de la imagen se aprenden mediante un algoritmo de *boosting* (Figura 1.17). Esta solución se caracteriza por presentar un coste computacional de operación reducido cuando se implementa sobre la base de *imágenes integrales*, por lo que se aplica en electrónica de consumo. Sin embargo, por lo general se asume que presenta un campo de aplicación más restringido.

Por último, cabe notar que el carácter espacial de los filtros descritos permite su manejo directo e intuitivo en muchas aplicaciones prácticas. No obstante, cuando la imagen presenta patrones aproximadamente repetitivos (p.ej. texturas o ruido periódico) puede resultar ventajoso realizar el filtrado en el dominio frecuencial ($pixeles^{-1}$). En este dominio, las distintas frecuencias espaciales de

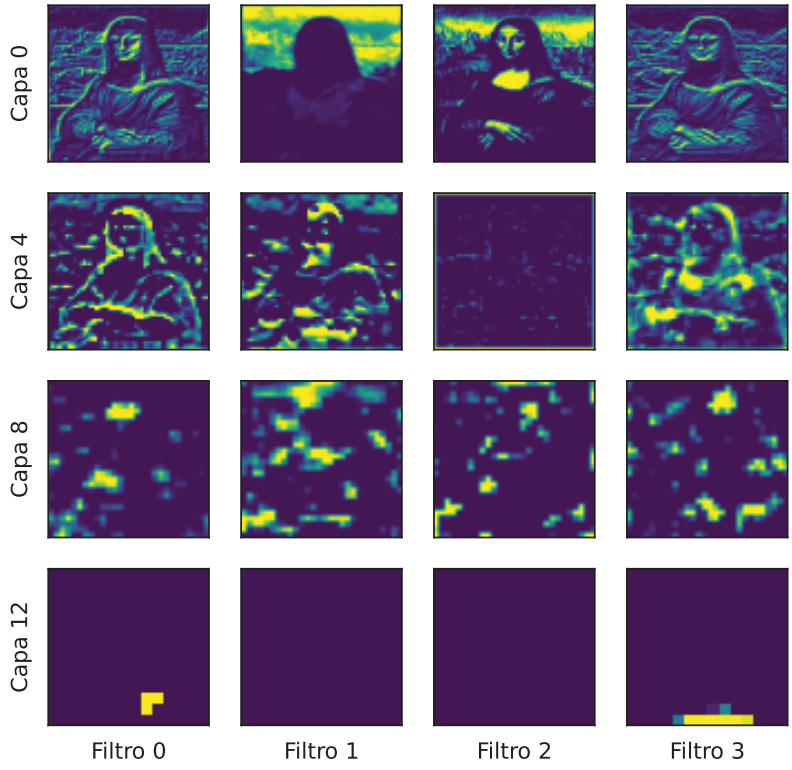


Figura 1.16: Filtros de la red convolucional VGG-16.

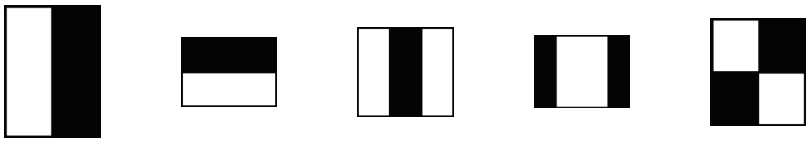


Figura 1.17: Ejemplos de características de Haar.

la imagen se localizan en zonas específicas del plano frecuencial. Además, los productos de convolución se convierten, en este plano, en productos punto a punto entre las transformadas de Fourier de la imagen y de la máscara espacial de convolución. Esta circunstancia puede ser aprovechada para acelerar la convolución con máscaras de tamaño elevado, mediante la transformación previa al dominio frecuencial, convolución en este y antitransformación del resultado.

1.6. Convolución multicanal

La discusión precedente se ha centrado en la convolución monocanal. La operación se puede aplicar también a imágenes multicanal, sin más que procesar cada canal por separado, tal como hicimos en el ejemplo de la Figura 1.5. El resultado es una imagen con el mismo número de canales que la imagen de partida.

Un tipo diferente de convolución es lo que se conoce como *convolución multicanal* y corresponde a generalizar la idea a tres dimensiones. Existen distintas formas de abordar esta generalización y la diferencia entre ellas es sutil. Además, la terminología no es estándar ni siempre consistente. Todo ello obliga a cierta reflexión previa que planteamos a continuación.

Partiremos de un concepto generalizado de imagen formada por N filas, M columnas y C *capas*. Cada capa puede estar estructurada, a su vez, como un conjunto de planos o canales P. Por ejemplo, una imagen de grises constará de una sola capa con un solo canal. Una imagen color RGB estará estructurada como una matriz de una capa con tres canales, R, G, y B. Un fragmento de vídeo con n imágenes RGB consistirá en una matriz de n capas, cada una integrada por tres canales. Por otra parte, no asumiremos ninguna restricción *a priori* sobre el tipo valores que contendrá una imagen. Estos podrán estar relacionados directamente con luminosidad o color, o bien ser el resultado de filtrados previos encaminados a resaltar ciertas características, tales como líneas, bordes, blobs etc. Cuando queramos referirnos específicamente a uno u otro tipo de imágenes utilizaremos el término *imagen óptica* o *imagen de características*, según corresponda. En lo que concierne a los filtros, habitualmente usaremos los términos *filtro* para referirnos a una matriz tridimensional y *núcleo* para cada uno de los planos que conforman un filtro. Sin embargo, tal como se ha indicado no se trata de una terminología estándar, por lo será utilizada con cierta flexibilidad.

Volviendo al tema de discusión, una primera forma de generalizar la convolución monocanal es lo que denominaremos *convolución bidimensional multicanal*. Se parte de una imagen con una o más capas de P canales cada una, y de un filtro integrado por P núcleos. Cada canal de una capa de la imagen se convoluciona con uno de tales núcleos, y los resultados obtenidos se suman entre sí. De ello resulta, por cada capa de entrada de P canales, una capa de salida con un único canal. El mismo proceso se repite para las demás capas. Debe notarse que el núcleo recorre cada capa de partida en forma bidimensional, es decir únicamente a lo largo de filas y columnas, de ahí la denominación convolución

Figura 1.18: Convolución multicanal con tres máscaras de $5x5x3$, con sus 25 pesos iguales a 0,2126, 0,7152 y 0,0722 respectivamente.

bidimensional multicanal.

Un ejemplo sencillo, útil en no pocas ocasiones, consiste en convolucionar una imagen de P canales con un núcleo de tamaño $1x1xP$. Por ejemplo, la convolución de una imagen RGB con un núcleo de tamaño $1x1x3$ con pesos 0,2126, 0,7152 y 0,0722 proporciona una imagen de grises que preserva la sensación perceptual de luminosidad de la imagen color original (para las tres fuentes de luz definidas por el estándar sRGB y asumiendo que no se aplica corrección gamma). La Figura 1.2 que venimos utilizando como ejemplo fue obtenida de esta manera.

La Figura 1.18 muestra otro ejemplo, esta vez correspondiente a aplicar un filtro conformado por tres núcleos de 5 x 5, con sus 25 valores igualados a 0,2126, 0,7152 y 0,0722 respectivamente. Su aspecto es similar al de la Figura 1.2 pero suavizada por efecto del filtrado de la media. Por otra parte, por supuesto nada impide utilizar núcleos distintos para cada canal y veremos ejemplos de ello en capítulos posteriores.

Una segunda generalización posible de la convolución monocanal es lo que se conoce como *convolución tridimensional*, propiamente dicha. En este caso se asume que la imagen de partida es una matriz tridimensional, como por ejemplo las que proporcionan los equipos de tomografía o resonancia magnética. El filtro es también una matriz tridimensional, habitualmente con un número de filas, columnas y planos reducido en comparación con el de la imagen de partida. La operación es la generalización directa de la descrita en la Ecuación (1.2), a saber:

$$J(x,y,z) = \sum_{s=-M/2}^{M/2} \sum_{t=-N/2}^{N/2} \sum_{u=-P/2}^{P/2} W(s,t,u)\ I(x+s, y+t, z+u)\ . \quad (1.11)$$

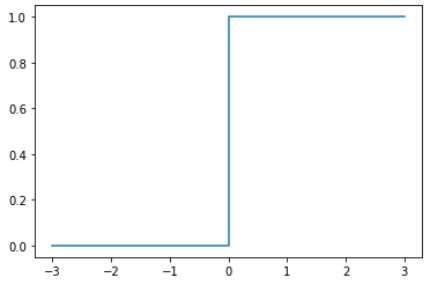


Figura 1.19: Función escalón.

El filtro recorre la imagen fuente a lo largo de filas, columnas y planos, dando como resultado una imagen tridimensional de tamaño comparable a la original. Este tipo de procesamiento se utiliza por ejemplo para la mejora y segmentación de imágenes de resonancia, tomografía, rayos X y similares.

1.7. Procesamiento del resultado del filtrado

Una vez completados los filtrados previos, el *pipeline* tradicional de procesamiento continúa con la segmentación. El método más simple consiste en umbralizar directamente el resultado del filtrado mediante $u(J(x, y) - th)$, donde u es la función escalón (Figura 1.19), $J(x, y)$ el nivel en el punto (x, y) resultante de los filtrados precedentes y th el umbral elegido.

Por supuesto esta técnica sencilla admite un sin fin de perfeccionamientos, tales como la umbralización multinivel, el cálculo adaptativo de umbrales, la consideración de criterios de vecindad y conectividad, consideraciones semánticas de más alto nivel, etc. Algunas ideas se han apuntado ya en la sección de operaciones con imágenes, o cuando se mencionaron los filtros de Sobel y Canny, aunque existen muchas otras técnicas cuya discusión escapa al ámbito del presente libro.

Por otra parte, en el contexto del aprendizaje profundo se recurre también a alternativas a la función escalón tales como la función sigmoidea (o la similar, tangente hiperbólica) o la función *ReLu*. La primera puede entenderse como una aproximación derivable de la función escalón. Esto resulta de trascendencia cuando se busca optimizar automáticamente los parámetros del filtrado mediante técnicas de derivada, por ejemplo de gradiente descendente como veremos en capítulos posteriores. Por otra parte, cuando el filtrado transcurre a través de varias etapas en cascada, suele resultar ventajoso utilizar la mencionada función *ReLu* o alguna aproximación derivable. Esta función proporcionan 0 como valor se salida donde el resultado de una convolución ha sido negativo (es decir, donde se ha encontrado una correlación negativa con el filtro), o el propio valor de la entrada en otro caso. Ello permite centrar las etapas subsiguientes del procesamiento en los rasgos para los que se ha detectado cierta correlación, aún pequeña, entre la imagen y los filtros empleados. Todos estos temas se relacionan

íntimamente con las redes neuronales, las redes convolucionales y el aprendizaje profundo, por lo cual serán tratados con detalle en capítulos posteriores.

Por último, las etapas finales de extracción de características y reconocimiento de objetos para la interpretación de las escenas quedan también fuera del ámbito del presente libro y para su estudio nos remitimos a los textos especializados sobre el tema. En todo caso, las técnicas que describimos en los siguientes capítulos permiten obviar gran parte de estas etapas, constituyendo así un camino alternativo hacia la comprensión de las información visual.

1.8. Transformaciones geométricas

Con independencia de las operaciones descritas hasta ahora, numerosas tareas de Visión comportan la realización transformaciones geométricas de las imágenes. Las más comunes son la traslación, el escalado, la rotación y la inclinación o cizalladura. Se suelen expresar en *forma homogénea* como:

$$\begin{pmatrix} x_T \\ y_T \\ 1 \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{1.12}$$

donde (x, y) son las coordenadas originales y (x_T, y_T) las resultantes de la transformación. En general, asumiremos que todas las coordenadas vienen dadas en unidades de píxel, es decir, como (*columna*, *línea*). La tercera ecuación no aporta información adicional pero facilita el encadenamiento y la inversión de transformaciones.

Distintos valores de los parámetros c_{ij} dan lugar a transformaciones diferentes. Por ejemplo,

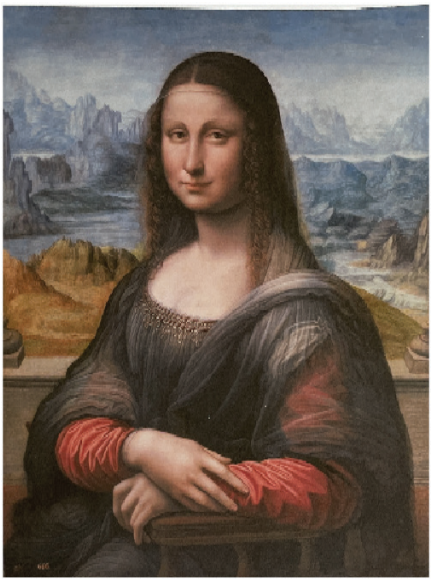
$$\begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & -c_x & 0 \\ -c_y & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{1.13}$$

expresan, respectivamente, una traslación de vector (t_x, t_y), un escalado (isotrópico en el caso $S_x = S_y$), un giro de ángulo θ en torno al origen (que asumiremos en la esquina superior izquierda de la imagen), y una inclinación de parámetros c_x, c_y. Desde un punto de vista práctico es importante notar que suele resultar ventajoso programar estas transformaciones mediante bucles que recorren la imagen transformada en lugar de la imagen original. Así, el nivel en cada posición (x_T, y_T) de la imagen transformada se obtienen a partir del nivel en la posición (x, y) de la imagen original mediante:

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ 0 & 0 & 1 \end{pmatrix}^{-1} \begin{pmatrix} x_T \\ y_T \\ 1 \end{pmatrix} . \tag{1.14}$$

Además, por lo general las coordenadas obtenidas de esta forma, (x, y), no adoptan valores enteros. Por ello, el nivel que corresponderá a (x_T, y_T) puede

(a) Fotografía original del objeto.



(b) Transformación homográfica.

Figura 1.20: Resultado de aplicar una homografía, en este caso encaminada a obtener una vista frontal (a invertir la homografía correspondiente a la captura de la imagen).

obtenerse a partir de las coordenadas redondeadas al vecino más próximo a (x, y) o bien, de forma más elaborada, mediante una interpolación bilineal a partir de los 4 vecinos más próximos a (x, y) o bicúbica a partir de 16 vecinos, con el fin de reducir el *pixelado* de la imagen transformada.

Las transformaciones geométricas descritas constituyen un caso particular de las denominadas genéricamente *transformaciones homográficas*,

$$\begin{pmatrix} x_T\ n \\ y_T\ n \\ n \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} , \tag{1.15}$$

que representan una proyección entre dos planos. Aquí, n es el *parámetro homogéneo* cuyo valor, no nulo, viene dado por la tercera ecuación. Las coordenadas transformadas (x_T, y_T) se calculan dividiendo $(x_T\ n,\ y_T\ n)$ entre n. La Figura 1.20 muestra un ejemplo en el que se ha invertido la transformación homográfica correspondiente a una captura de una imagen con el fin de obtener una perspectiva frontal. Los parámetros se han determinado mediante un ajuste de mínimos cuadrados a partir de cuatro puntos.

La transformación homográfica es, a su vez, un caso particular de la *transformación proyectiva* entre un espacio tridimensional y un plano de proyección,

$$\begin{pmatrix} x_T\ n \\ y_T\ n \\ n \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & c_{1z} & c_{13} \\ c_{21} & c_{22} & c_{2z} & c_{23} \\ c_{31} & c_{32} & c_{3z} & c_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} . \tag{1.16}$$

La captación de imágenes de una escena tridimensional por medio de una cámara es un buen ejemplo de transformación proyectiva.

1.9. Remuestreo y pirámides de imágenes

El denominado *remuestreo* es una operación común en Visión. Permite modificar el tamaño de las imágenes de cara a su manejo ulterior. Este tamaño se especifica habitualmente en forma de *resolución*, entendida como número de píxeles que conforman la imagen. El remuestreo puede aplicarse en dos sentidos: submuestreo (*subsampling*) y sobremuestreo (*upsampling*).

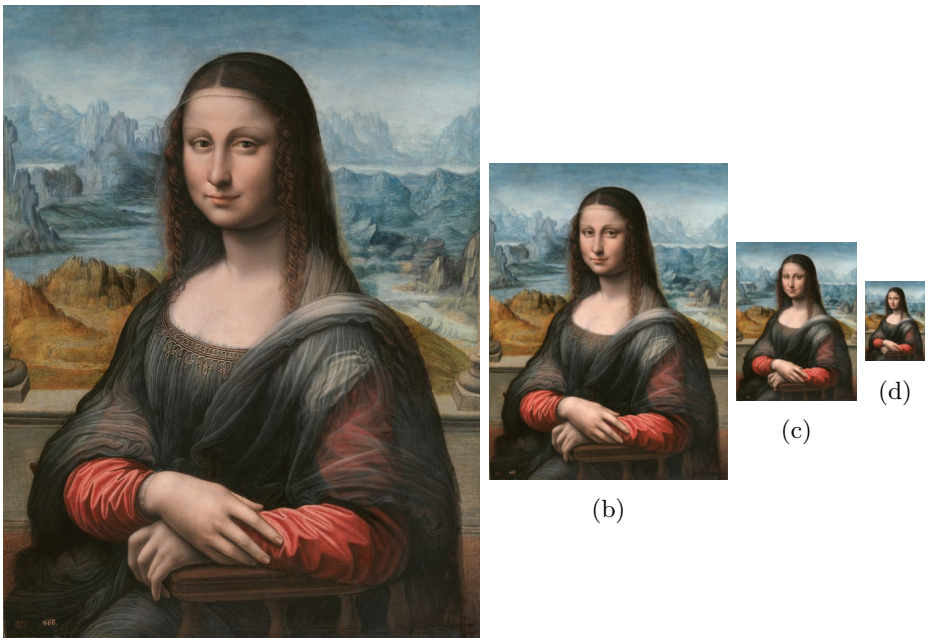
El submuestreo consiste en remuestrear la imagen a una frecuencia espacial inferior, es decir un periodo de muestreo superior. Esto resulta útil para reducir los requerimientos computacionales en tareas como el reconocimiento de objetos o el almacenamiento y la transmisión de imágenes. La mecánica es análoga a la convolución: se define una ventana de análisis que recorre la imagen a lo largo de sus filas y columnas, con un cierto paso (mayor que 1). Para cada posición se calcula el valor que se asignará al píxel correspondiente de la imagen de salida, siguiendo alguna de las alternativas siguientes:

- Selección de la media (*average pooling*)
- Selección gaussiana (*gaussian pooling*)
- Selección del máximo (*average pooling*):

En el primer caso la ventana de análisis implementa un filtro de la media y en el segundo un filtro gaussiano. Se trata de filtros pasa-bajos que permiten reducir los eventuales fenómenos de *aliassing*. La Figura 1.21 muestra un ejemplo: el resultado de un submuestrear una imagen a frecuencias decrecientes en un factor 0,5 , para construir lo que denomina una pirámide de imágenes. (Por supuesto podría haberse empleado otro factor). Como filtro pasa bajos se ha recurrido a una aproximación de una gaussiana de $\sigma \approx 1.082$, de uso común en muchas bibliotecas de procesamiento de imágenes.

En otras ocasiones, en particular cuando se procesan imágenes de características, se persigue específicamente que la imagen remuestreada conserve los rasgos más sobresalientes de la imagen de partida. En tales condiciones resulta preferible recurrir a una selección del máximo valor en la vecindad definida por la ventana de análisis. Esta técnica es de uso común en redes convolucionales por lo que será retomada en capítulos posteriores.

El sobremuestreo, por su parte, consiste en remuestrear la imagen de partida a una frecuencia espacial superior a la de entrada, es decir con un periodo de



(a) (b) (c) (d)

Figura 1.21: Ejemplo de pirámide decreciente imágenes. De izquierda a derecha: imagen original y resultado de su remuestreo gaussiano a octavas decrecientes sucesivas (frecuencias 1/2, 1/4 y 1/8 $pixeles^{-1}$).

(a) (b) (c) (d)

Figura 1.22: Ejemplo de pirámide creciente de imágenes. De izquierda a derecha: imagen de partida (tomada de la figura 1.21(d)), y resultado de su remuestreo a octavas crecientes sucesivas (frecuencias 2, 4 y 8 $pixeles^{-1}$).

muestreo inferior. La resolución de salida será superior a la de entrada. Esto conlleva realizar interpolaciones bilineales o bicúbicas, como las mencionadas anteriormente, o incluso, en ocasiones, a interpolaciones guiadas por aprendizaje. La Figura 1.22 muestra un ejemplo, en el que se ha partido de la imagen correspondiente al nivel 1/8 de la Figura 1.21. En general, la operación de submuestreo no es reversible dado que comporta pérdida de información. Por ello, las imágenes de las Figuras 1.21(a) y 1.22(d) no son iguales (la segunda, de 954 x 720 píxeles, se ha generado a partir de la primera, de tan solo 120 x 90 píxeles). Por otra parte, la combinación submuestreo-sobremuestreo se utiliza con frecuencia en arquitecturas tipo codificador-decodificador como veremos en capítulos posteriores, por ejemplo para reducir el coste computacional de reconocer un objeto en una imagen y reubicarlo luego, de forma aproximada, en la imagen original.

Capítulo 2

Frameworks de visión y deep learning

Durante los últimos años han aparecido diversas plataformas y marcos de trabajo, o *frameworks*, para facilitar el desarrollo de aplicaciones de aprendizaje automático. Este tipo de aplicaciones permiten desarrollar modelos de visión e inteligencia artificial. A lo largo del libro utilizaremos los frameworks Tensorflow y Pytorch. Sin embargo, aparte de estos frameworks, existen otras plataformas de alto y bajo nivel que se pueden utilizar para el aprendizaje automático y la visión artificial. Las plataformas de alto nivel son aquellas que prácticamente no requieren un conocimiento de programación y que se limitan a ofrecer un conjunto de servicios determinados, como puede ser una red neuronal de clasificación de imágenes donde nosotros solo necesitamos conocer dónde dejar las imágenes de entrenamiento. Las plataformas de bajo nivel, como Tensorflow o Pytorch, requieren un conocimiento de programación para poder diseñar los modelos. En muchas ocasiones, es posible utilizar modelos ya existentes.

Dentro de las plataformas de alto nivel podemos encontrar:

- Amazon Machine Learning: Dentro de AWS, Amazon ha tratado de ofrecer servicios de aprendizaje automático integrados con sus soluciones de almacenamiento y procesamiento. Se conecta a los datos almacenados en Amazon S3, Redshift o RDS, y puede realizar una clasificación binaria, una categorización multiclase o una regresión en esos datos para crear un modelo. Una desventaja de esta solución es que se dependen de los datos almacenados en Amazon, los modelos resultantes no pueden importarse ni exportarse, y los conjuntos de datos para el entrenamiento de los modelos no pueden superar los 100 GB.

- Microsoft Azure ML Studio: Microsoft ha dotado a Azure de su propio servicio de aprendizaje automático de pago, Azure ML Studio, con versiones mensuales, por horas y continuas. Azure ML Studio permite a los usuarios crear y entrenar modelos, y luego convertirlos en APIs que pue-

den ser consumidas por otros servicios. Los usuarios pueden obtener hasta 10 GB de almacenamiento por cuenta de modelo de datos, aunque el propio almacenamiento de Azure también puede conectarse al servicio para modelos más grandes. Existe una amplia gama de algoritmos, tanto de Microsoft como de terceros.

- Google Cloud AutoML: Proporciona modelos pre-entrenados para que los usuarios puedan crear diversos servicios. Por ejemplo, reconocimiento de texto, reconocimiento de voz, etc. Google Cloud AutoML se ha hecho muy popular entre las empresas que utilizan las plataformas de Google debido a la facilidad de aplicar la inteligencia artificial en todos los sectores, y más teniendo en cuenta la dificultad de encontrar profesionales de Machine Learning en frameworks de bajo nivel.
- IBM Watson Maching Learning: La solución que ofrece IBM para utilizar algoritmos de IA en la nube, utilizando datos de diversas fuentes: IBM Cloud, AWS, Azure, Google o su propia plataforma de cloud privado. Watson se aplica en distintos campos, como el aprendizaje automático o la extracción de información. Es parte de IBM Watson Studio y ayuda a los profesionales en el despliegue de modelos de ML en IBM Cloud Pak for Data. Se pueden desplegar modelos de machine learning y deep learning y modelos de optimización de decisiones, volver a entrenar modelos dinámicamente con el aprendizaje continuo, generar automáticamente APIs para crear aplicaciones basadas en IA a través de DevOps, o gestionar y supervisar modelos en busca de desviaciones, sesgos y riesgos en modelos.

Dentro de los frameworks y plataformas de bajo nivel podemos encontrar:

- TensorFlow-Keras: Plataforma de código abierto para machine learning. Tiene un amplio ecosistema de herramientas, bibliotecas y recursos de la comunidad que permiten construir y desplegar fácilmente aplicaciones de aprendizaje automático. Es un framework diseñado para escalar a través de múltiples nodos. Al igual que Kubernetes de Google, fue construido para resolver problemas internos de Google, pero finalmente fue liberarlo como un producto de código abierto. TensorFlow implementa lo que se denomina diagramas de flujo de datos, donde los arrays de datos (tensores) pueden ser procesados por una serie de algoritmos que se describen mediante grafos. Los grafos pueden ensamblarse con C++, Python o Java, y pueden procesarse en CPUs o GPUs. Dentro de Tensorflow actualmente se incluye Keras, que es una biblioteca de código abierto escrita en Python. Es capaz de ejecutarse sobre TensorFlow, Microsoft Cognitive Toolkit o Theano. Está especialmente diseñada para posibilitar la experimentación en más o menos poco tiempo con redes neuronales, incluyendo las de aprendizaje profundo. Es un complemento modular y extensible que además es fácil de aprender para el usuario y facilita el desarrollo y experimentación con modelos. Lo veremos posteriormente con más detalle al ser uno de los frameworks utilizados en el libro.

- Pytorch: Framework de código abierto desarrollado por el equipo de investigación de Facebook. Permite implementar modelos de deep learning en Python, ofreciendo una gran cantidad de funcionalidades. PyTorch permite añadir varios módulos como torchvision, torchaudio o torchtext, que son muy flexibles para trabajar en visión por ordenador o procesamiento de lenguaje natural. Sus tensores permiten la auto diferenciación, utilizando grafos acíclicos dirigidos DAG durante el proceso de retro-propagación para almacenar gradientes. Esto ayuda a acelerar el proceso de entrenamiento. Pytorch suele ser muy utilizado en la comunidad científica. Lo veremos posteriormente con más detalle al ser uno de los frameworks utilizados en el libro.

- Caffe: El entorno de deep learning Caffe (*Convolutional Architecture for Fast Feature Embedding*) está implementado pensando en la velocidad y la modularidad. Aunque fue inicialmente desarrollado por Yangqing Jia para proyectos de visión artificial, se ha ampliado desde entonces para incluir otras aplicaciones de inteligencia artificial, como puede ser el procesamiento del lenguaje natural (del inglés *Natural Language Processing* (NLP). Caffe es muy rápido al estar escrito íntegramente en C++ con soporte para aceleración CUDA, aunque se puede alternar entre el procesamiento en la CPU y en la GPU según sea necesario. La mayoría de los desarrolladores utilizan Caffe por su velocidad, y puede procesar 60 millones de imágenes al día con una sola GPU NVIDIA K40. Caffe cuenta con muchos colaboradores que actualizan y mantienen los frameworks. La distribución incluye un conjunto de modelos de referencia gratuitos y de código abierto para trabajos de clasificación comunes, con otros modelos creados por la comunidad de usuarios de Caffe. Según algunas comparativas, su velocidad puede superar a Tensorflow en torno a un 50 %. Recientemente, Caffe2 ha permitido aprovechar el Cloud Computing. Como desventajas podemos decir que: su curva de aprendizaje y desarrollo es tan complejo que pocos proyectos se llevan a cabo en Caffe; no tiene una API de alto nivel, por lo que es difícil hacer experimentos; y, finalmente, para desplegar un modelo en Caffe necesitamos compilar los códigos fuente si programamos en C++. Sin embargo, el framework Caffe también se puede utilizar desde Python, reduciendo algunas de las desventajas mencionadas.

- Scikit-Learn: Paquete de aprendizaje automático de código abierto y enfocado a múltiples propósitos. Ayuda en la resolución de problemas de regresión, agrupación o clustering, clasificación, reducción de la dimensionalidad y preprocesamiento. Scikit-Learn está construido sobre las tres principales bibliotecas de Python, NumPy, Matplotlib y SciPy. Algunos de los algoritmos que podemos utilizar son, por ejemplo, Principal Component Analysis (PCA) o Support Vector Machine (SVM).

- Microsoft Computational Network Toolkit (CNTK) y Distributed Machine Learning Toolkit (DMTK): Kit de herramientas de aprendizaje automático distribuido de Microsoft. El framework DMTK aborda el problema de

la distribución de varios tipos de trabajos de aprendizaje automático en un clúster de sistemas. DMTK está clasificado como un entorno más que como una solución completa, por lo que el número de algoritmos que incluye es reducido. Poco después del lanzamiento del DMTK, Microsoft presentó otro kit de herramientas de aprendizaje automático, el CNTK. CNTK es similar a TensorFlow de Google, ya que permite a los usuarios crear redes neuronales a través de grafos dirigidos. Microsoft también lo considera comparable a proyectos como Caffe, Theano y Torch. Su principal ventaja frente a esos marcos es la velocidad, concretamente la capacidad de explotar múltiples CPUs y múltiples GPUs en paralelo. Según Microsoft, el uso de CNTK junto con los clústeres de GPU en Azure acelera el entrenamiento del reconocimiento de voz de Cortana.

- Apache Singa: Es un framework de código abierto destinado a facilitar el entrenamiento de modelos de aprendizaje profundo en grandes volúmenes de datos. Singa proporciona un modelo de programación sencillo para el entrenamiento de redes de deep learning en un clúster de máquinas, y soporta los trabajos de entrenamiento más comunes: redes neuronales convolucionales, máquinas de Boltzmann restringidas o redes neuronales recurrentes. Los modelos se pueden entrenar de forma síncrona (uno tras otro) o asíncrona (uno al lado del otro), dependiendo de lo que funcione mejor para el problema en cuestión. Singa también simplifica la configuración del clúster con Apache Zookeeper. También dentro de Apache tenemos Mahout, que es una plataforma de código abierto basada en Hadoop. Mahout permite aprendizaje automático o minería de datos, incluyendo técnicas de regresión, clasificación o clustering.

- DarkNet: Es un framework de redes neuronales de código abierto escrito en C y CUDA. Tiene especial interés en la comunidad científica al haberse utilizado para crear algunas versiones originales de YOLO, el conocido detector que veremos en capítulos posteriores del libro. DarkNet es rápido, fácil de instalar y admite cálculo en CPU y GPU. El código fuente, así como la guía de utilización, se pueden encontrar en `https://pjreddie.com/darknet/`.

- MediaPipe: Es un framework diseñado para ser eficiente y requerir mínimos recursos, lo que le ha llevado a ser utilizado en numerosos dispositivos de IoT. Inicialmente se desarrolló para el análisis en tiempo real de vídeo y audio en YouTube, y ha sido ampliamente utilizado por Google en muchos de sus productos, incluyendo la detección de objetos en Google Lens o la implementación de la API de Cloud Vision. En su web, `https://mediapipe.dev/`, ofrecen numerosos ejemplos de cómo utilizar sus modelos, que permiten detectar esqueletos de personas, detectar puntos característicos de manos y cara, segmentación de imágenes, detección de objetos en 3D a partir de imágenes 2D y muchas otras aplicaciones. Para su instalación, basta con lanzar el comando *pip install mediapipe*, lo que hace que sea un framework muy accesible. Como inconveniente, po-

demos indicar que está pensado en ofrecer soluciones hechas, no pensado para entrenar modelos propios.

2.1. Introducción a OpenCV

OpenCV (Open Source Computer Vision Library) es la biblioteca de visión artificial más utilizada en la actualizad. La biblioteca cuenta con más de 2.500 algoritmos optimizados, que incluyen un amplio conjunto de algoritmos de visión por ordenador y aprendizaje automático, tanto clásicos como de última generación. Fue una biblioteca creada para satisfacer las necesidades de las aplicaciones de visión y de percepción artificial en productos comerciales. Al ser un producto con licencia Apache 2, OpenCV facilita a los profesionales la utilización y modificación del código. OpenCV se puede utilizar con Python, C++, Java o Matlab.

Incluye algoritmos que pueden utilizarse para detectar y reconocer caras, identificar objetos, clasificar acciones humanas en vídeos, rastrear movimientos de cámara, seguir objetos en movimiento, extraer modelos 3D de objetos, producir nubes de puntos 3D, obtener una imagen de alta resolución a partir de varias imágenes de una escena, buscar imágenes similares a partir de una base de datos de imágenes, etc. A mayores, incluye infinidad de algoritmos de visión clásica para realizar todo tipo de transformaciones y procesamientos de imágenes. El módulo DNN de OpenCV permite llevar a cabo la inferencia de una red neuronal sobre imágenes y vídeos. Aunque no permite el entrenamiento de las redes, sí nos permite trabajar con modelos que hemos entrenado en otros frameworks como Tensorflow, Pytorch, DarkNet o Caffe.

La instalación de OpenCV se puede realizar utilizando pip, con el siguiente comando: *pip install opencv-python*

Hay que tener en cuenta que este tipo de instalación no incluye el soporte para GPU. Para ciertos algoritmos o los modelos de Deep Learning, el uso de GPU es altamente recomendable. Pero para utilizar OpenCV con GPU es necesario compilarlo. El proceso de compilación de OpenCV suele llevar bastante tiempo. El Script 2.1 muestra un ejemplo de cómo podría llevarse a cabo dicha compilación. Es necesario identificar el tipo de arquitectura de la GPU a utilizar para que la compilación y la posterior ejecución sean satisfactorias.

Código 2.1: Compilación de OpenCV para uso con GPU

```
# Primero: Incluir las siguientes rutas a /etc/apt/sources.list
#   deb http://archive.ubuntu.com/ubuntu focal-updates main
#   deb http://es.archive.ubuntu.com/ubuntu/ focal main
    restricted
#   deb http://es.archive.ubuntu.com/ubuntu/ focal universe
#   deb http://es.archive.ubuntu.com/ubuntu/ focal multiverse
#   deb http://security.ubuntu.com/ubuntu focal-security
    restricted main
#   deb http://security.ubuntu.com/ubuntu focal-security universe
#   deb http://security.ubuntu.com/ubuntu focal-security
    multiverse
```

```
#   deb http://archive.ubuntu.com/ubuntu focal main universe
    restricted multiverse

sudo apt update
sudo apt upgrade

sudo apt install build-essential cmake pkg-config unzip yasm git
    checkinstall
sudo apt install libjpeg-dev libpng-dev libtiff-dev
sudo apt install libavcodec-dev libavformat-dev libswscale-dev
    libavresample-dev

sudo apt install libgstreamer1.0-dev libgstreamer-plugins-base1
    .0-dev
sudo apt install libxvidcore-dev x264 libx264-dev libfaac-dev
    libmp3lame-dev libtheora-dev

sudo apt install libfaac-dev libmp3lame-dev libvorbis-dev

sudo apt install libopencore-amrnb-dev libopencore-amrwb-dev

sudo apt-get install libdc1394-22 libdc1394-22-dev libxine2-dev
    libv4l-dev v4l-utils
cd /usr/include/linux
sudo ln -s -f ../libv4l1-videodev.h videodev.h
cd ~

sudo apt-get install libgtk-3-dev

# Python
sudo apt-get install python3-dev python3-pip
sudo -H pip3 install -U pip numpy
sudo apt install python3-testresources

sudo apt-get install libtbb-dev
sudo apt-get install libatlas-base-dev gfortran

wget -O opencv.zip https://github.com/opencv/opencv/archive/refs/
    tags/4.5.2.zip

wget -O opencv_contrib.zip https://github.com/opencv/
    opencv_contrib/archive/refs/tags/4.5.2.zip

unzip opencv.zip
unzip opencv_contrib.zip

cd opencv-4.5.2
mkdir build
cd build

cmake -D CMAKE_BUILD_TYPE=RELEASE \
-D CMAKE_INSTALL_PREFIX=/usr/local \
-D WITH_TBB=ON \
-D ENABLE_FAST_MATH=1 \
-D CUDA_FAST_MATH=1 \
-D WITH_CUBLAS=1 \
-D WITH_CUDA=ON \
```

```
-D BUILD_opencv_cudacodec=OFF \
-D WITH_CUDNN=ON \
-D OPENCV_DNN_CUDA=ON \
-DCUDNN_LIBRARY=/usr/lib/cuda/lib64/libcudnn.so.8.0.5 \
-DCUDNN_INCLUDE_DIR=/usr/lib/cuda/include \
-D CUDNN_VERSION='8.0' \
-D CUDA_ARCH_BIN=7.5 \
-D CUDA_TOOLKIT_ROOT_DIR=/usr/local/cuda \
-D WITH_V4L=ON \
-D WITH_QT=OFF \
-D WITH_OPENGL=ON \
-D WITH_GSTREAMER=ON \
-D OPENCV_GENERATE_PKGCONFIG=ON \
-D OPENCV_PC_FILE_NAME=opencv.pc \
-D OPENCV_ENABLE_NONFREE=ON \
-D OPENCV_PYTHON3_INSTALL_PATH=/usr/local/lib/python3.8/dist-
    packages \
-D PYTHON_EXECUTABLE=/usr/bin/python3 \
-D OPENCV_EXTRA_MODULES_PATH=/home/roasis/src/opencv_contrib
    -4.5.2/modules \
-D INSTALL_PYTHON_EXAMPLES=OFF \
-D INSTALL_C_EXAMPLES=OFF \
-D BUILD_EXAMPLES=OFF ..

nproc
make -j8
sudo make install

sudo /bin/bash -c 'echo "/usr/local/lib" >> /etc/ld.so.conf.d/
    opencv.conf'
sudo ldconfig

# Añadir a .bashrc:

echo 'export PYTHONPATH=/usr/local/lib/python3.8/dist-packages/:
    $PYTHONPATH' >> ~/.bashrc
```

El Código 2.2 muestra un ejemplo de programa Python que nos mostraría el número de GPUs disponibles para OpenCV.

Código 2.2: Verificar OpenCV con GPU

```
import cv2
numeroGPUs = cv2.cuda.getCudaEnabledDeviceCount()
print("GPUs disponibles: ", numeroGPUs)
```

2.2. Pillow y DLIB

Más allá de OpenCV, existen numerosas librerías que permiten operar con imágenes en Python. Es interesante conocerlas, por lo menos a nivel superficial, a la hora de poder leer código desarrollado por otras personas. En esta sección exploraremos dos de las librerías más importantes.

2.2.1. Pillow

La actual librería Pillow, originada a partir de *Python Imaging Library* (PIL), añade capacidades de procesamiento de imágenes y proporciona un amplio soporte de formatos de archivo, con una representación interna eficiente de las imágenes. A diferencia de OpenCV, programado en C y C++, PIL ha sido desarrollado en Python y C. Por este motivo, es algo más lenta que OpenCV en muchas operaciones. Aunque no implementa internamente modelos de aprendizaje automático ni algoritmos tan complejos como OpenCV o DLIB, es una librería exclusiva de Python y es muy utilizada en los proyectos de aprendizaje automático, con Pytorch y Tensorflow. Una de las razones por las que se utiliza más con este tipo de proyectos es que utiliza una representación interna de la información que permite realizar transformaciones más fácilmente que con OpenCV, que internamente utiliza arrays. Además, PIL trabaja por defecto en RGB mientras que OpenCV lo hace en BGR. Finalmente, es más fácil distribuir Pillow que OpenCV, lo que nos reduce problemas de dependencias a la hora de integrarlo con Tensorflow y Pytorch. Es una librería de código abierto que se instala en Python directamente con PIP: *pip install Pillow*. También es posible instalarla utilizando Conda, si por ejemplo utilizamos Anaconda para nuestros proyectos: *conda install -c anaconda pillow*. En la web `https://pypi.org/project/Pillow/` podemos encontrar toda la documentación de la librería.

2.2.2. DLIB

DLIB es una librería que contiene numerosos algoritmos de aprendizaje automático escritos en C++ pero que pueden ser utilizados con Python para resolver diferentes tipos de problemas, muchos de visión artificial, como pueden ser reconocimiento facial, seguimiento de objetos en vídeo o segmentación de objetos. DLIB también incluye numerosas funciones de procesado de imágenes. Es una librería de código abierto que se instala en Python directamente con PIP: *pip install dlib*. También es posible instalarla utilizando Conda, si por ejemplo utilizamos Anaconda para nuestros proyectos: *conda install -c conda-forge dlib*.

En muchos proyectos es habitual ver la utilización de varias librerías simultáneamente, como PIL y DLIB, ya que puede interesarnos realizar ciertas funciones con una u otra librería. También, muchas operaciones de procesado de imágenes y muchos modelos se encuentran disponibles en las diferentes bibliotecas, OpenCV, DLIB o PIL, por lo que podremos elegir la que más nos convenga en función de nuestros conocimientos o necesidades. Además, ciertos problemas pueden ser resueltos de una manera más o menos favorable a nuestros intereses. Por ejemplo, el reconocimiento de caras puede ser más rápido con DLIB, pero a lo mejor no somos capaces de detectar tantas caras pequeñas como detecta OpenCV. En la web de DLIB, `dlib.net`, podemos encontrar una completa descripción de todas las funcionalidades.

Al igual que ocurría con PIL, DLIB utiliza una representación diferente de las imágenes de manera interna. También utiliza arrays y permite formatos RGB

y BGR. Si utilizamos las dos librerías para operar sobre una misma imagen deberemos tener en cuenta la distribución de canales.

El Código 2.3 muestra la utilización de PIL y DLIB de manera simultánea. Mientras que utilizamos DLIB para buscar una cara en una imagen mediante el detector de HOG (*Histogram of Oriented Gradients*) y SVM, utilizamos PIL para pintar un cuadro rojo sobre la cara (ver Figura 2.1).

Código 2.3: Detección de caras en imagen con DLIB y PIL

```
########################################
# Detección de caras con DLIB y PIL
########################################
import dlib
from PIL import Image, ImageDraw

# Imagen de entrada y fichero de salida
fichero = 'imagen.jpg'
resultado = 'resultado.jpg'

# Abrir imagen con DLIB
img = dlib.load_rgb_image(fichero)

# Detectar caras con DLIB
# HOG (Histogram of Oriented Gradients) y SVM
face_detector = dlib.get_frontal_face_detector()
detected_faces = face_detector(img, 1)
print('%d caras detectadas' % (len(detected_faces)))

# Cargamos de nuevo imagen con PIL
imagen = Image.open(fichero)

# Para cada cara detectada pintamos un rectángulo
for i, face_rect in enumerate(detected_faces):
    width = face_rect.right() - face_rect.left()
    height = face_rect.bottom() - face_rect.top()

    # Creamos la región de la cara a partir de los datos de DLIB
    region_cara = [(face_rect.left(), face_rect.top()), (
        face_rect.right(), face_rect.bottom())]

    # Dibujamos un rectángulo
    imagenD = ImageDraw.Draw(imagen)
    imagenD.rectangle(region_cara, outline ='red', width=15)

# Mostramos la imagen resultado
imagen.show()
imagen.save(resultado)
```

2.3. Introducción a Tensorflow

TensorFlow es uno de los frameworks de aprendizaje automático más utilizados. Es una plataforma de código abierto que cuenta con un amplio ecosistema

Figura 2.1: Resultado de la detección de caras con PIL y DLIB

de herramientas, bibliotecas y recursos comunitarios que permiten crear e implantar fácilmente aplicaciones de *Machine Learning* (ML).

Fue desarrollado por Google y está diseñado para escalar a través de múltiples nodos. Al igual que Kubernetes, plataforma para automatizar la implementación, el escalado y la administración de aplicaciones en contenedores, se creó para resolver problemas internos de Google. Finalmente, Google optó por publicarlo como producto de código abierto.

TensorFlow implementa diagramas de flujo de datos (grafos), donde los datos son manejados con *tensores* que pueden ser procesados por una serie de algoritmos descritos por un grafo. El movimiento de datos a través del sistema se denomina flujo. A diferencia de un array multidimensional de Python de tipo NumPy, donde los valores deben pertenecer a un mismo tipo numérico, los tensores pueden ser vistos también como arrays extendidos. Al igual que los arrays, los tensores pueden ser unidimensionales y multidimensionales. Un tensor con eje cero contiene un único valor y se denomina *escalar*. Un tensor con un eje contiene una lista de valores y se denomina *vector*. Un tensor con 2 o más ejes se denomina *matriz*. En el caso de los tensores multidimensionales, cada elemento debe tener el mismo tamaño, al igual que las matrices. Sin embargo, algunos tipos de tensores pueden manejar diferentes formas de elementos, como son los tensores irregulares o los tensores dispersos.

Una de las principales diferencias con los arrays es que los tensores están preparados para la diferenciación automática, mecanismo que les permite ajus-

tar los parámetros de una red neuronal para ayudar a la red a ajustarse a los resultados esperados. La diferenciación automática es útil para ciertos algoritmos, como la retropropagación. Para diferenciar automáticamente, TensorFlow necesita recordar qué operaciones suceden y en qué orden durante el paso hacia adelante. A continuación, durante la retropropagación o paso hacia atrás, TensorFlow recorre esta lista de operaciones en orden inverso para calcular los gradientes. Estos gradientes serán utilizados para ajustar los pesos del flujo del grafo en el paso hacia adelante. Los tensores además permiten la paralelización para una computación más rápida mediante el uso de unidades de proceso gráfico, del inglés *Graphics Processing Unit* (GPU), o unidades de proceso tensorial, del inglés *Tensor Processing Unit* (TPU).

El framework Tensorflow se puede utilizar con lenguajes como Python, C++ o incluso Java. Para poder trabajar con Tensorflow, con GPU, es necesario instalar CUDA (Compute Unified Device Architecture). Esta librería incluye un compilador y un conjunto de herramientas desarrolladas por Nvidia para llevar a cabo computación en paralelo, principalmente utilizando GPUs. Si tenemos GPU, de cara a elegir una versión CUDA compatible, tenemos que verificar qué arquitectura o capacidad de cómputo tiene nuestra GPU (eso se puede localizar en la web `https://developer.nvidia.com/cuda-gpus`). Hay distintas arquitecturas, como Turing, Pascal o Ampere, que llevan asociado un indicador denominado capacidad de cómputo. Así por ejemplo, una RTX3090 utiliza la arquitectura Ampere y tiene una capacidad de cómputo 7.5. A partir de dicho valor, podemos seleccionar una versión CUDA apropiada. Normalmente, la última versión será compatible con las GPU más recientes (URL: `https://developer.nvidia.com/cuda-downloads`). Tensorflow requiere adicionalmente de cuDNN, que es una biblioteca de primitivas para redes neuronales profundas. cuDNN proporciona implementaciones altamente ajustadas para rutinas estándar como la convolución hacia delante y hacia atrás, la agrupación, la normalización y las capas de activación. cuDNN forma parte del SDK de aprendizaje profundo de NVIDIA (URL: `https://developer.nvidia.com/cudnn`).

Hasta la versión 1.15 de Tensorflow (TF) era necesario especificar si queríamos soporte de GPU o de CPU. A partir de dicha versión, la instalación se ha simplificado, siendo la misma versión para los dos. Es importante revisar bien las variables de entorno (LD_LIBRARY_PATH, PATH) de CUDA y cuDNN para que Tensorflow las detecte correctamente y pueda trabajar con GPU. Si no encuentra estas variables, TF utilizará CPU y la velocidad de procesamiento de los modelos que veremos en el libro se reducirá notablemente. Toda la información sobre la configuración correcta de CUDA y cuDNN para su uso con Tensorflow se puede localizar en `https://www.tensorflow.org/install/gpu`. Hay que tener en cuenta que, si Tensorflow no ha sido compilado para una versión concreta de CUDA y cuDNN que utilizamos, es posible que no funcione correctamente. En dichos casos es posible compilar Tensorflow, pero dicho proceso es laborioso y costoso en tiempo, y queda fuera del alcance de este libro.

Una vez instalado CUDA y cuDNN, la instalación de Tensorflow se realiza en Windows y Linux directamente con alguno de los comandos mostrados en el Script 2.4. Dependiendo de si estamos instalando directamente con *conda* o con

pip, tendremos que seleccionar uno u otro. Recordemos que *conda* es el gestor de paquetes utilizado por defecto en Anaconda, que también nos facilita enormemente la instalación de Tensorflow desde la propia interfaz de la aplicación (Environments → Search Packages).

Código 2.4: Instalación de Pytorch

```
% En caso de CONDA
conda install -c conda-forge tensorflow

% En caso de PIP
pip install tensorflow
```

Si al instalar Tensorflow no nos detecta correctamente la GPU, es posible que no exista compatibilidad entre el TF instalado y las librerías CUDA y cuDNN. En dicho caso, tendremos que buscar una versión compatible en la web: `https://www.tensorflow.org/install/source?hl=es-419#gpu`

Una vez concluida la instalación, podemos verificar si todo funciona correctamente ejecutando el Código Python 2.5. Este código verificará primeramente qué versión de Tensorflow existe y mostrará las GPUs disponibles. A mayores, crearemos un tensor de ejemplo.

Código 2.5: Primer ejemplo de verificación de Tensorflow

```
import tensorflow as tf
# Datos de Tensorflow
print('Versión TF: ', tf.__version__)
# Versión de CUDA
sys_details = tf.sysconfig.get_build_info()
cuda_version = sys_details['cuda_version']
print('Versión de CUDA: ', cuda_version)
print('Dispositivos GPU: ', tf.config.list_physical_devices('GPU'
    ))
# Ejemplo de tensor
T = tf.constant([[1.0, 2.0], [3.0, 4.0]])
print(T)
```

El resultado debería ser parecido al mostrado en la Figura 2.2. A partir de este momento, ya podremos probar distintos programas que utilicen Tensorflow.

```
Versión TF:  2.10.0
Versión de CUDA:  64_112
Dispositivos GPU:  [PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
2022-12-19 15:16:54.305960: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1616]
Created device /job:localhost/replica:0/task:0/device:GPU:0 with 8861 MB memory:
-> device: 0, name: NVIDIA GeForce RTX 2080 Ti, pci bus id:
0000:01:00.0, compute capability: 7.5
tf.Tensor(
[[1. 2.]
 [3. 4.]], shape=(2, 2), dtype=float32)
```

Figura 2.2: Resultado de la ejecución del primer programa Tensorflow